本系列教程目标：使初学者了解富网络应用概念，理解并掌握以下四种架构方法。

1. Flex + BlazeDS + Spring（< 2.5.6） + iBATIS + Cairngorm
2. Flex + BlazeDS + Spring BlazeDS Integration + Spring（>= 2.5.6） + iBATIS + Cairngorm
3. Flex + BlazeDS + Spring（< 2.5.6） + iBATIS + pureMVC
4. Flex + BlazeDS + Spring BlazeDS Integration + Spring（>= 2.5.6） + iBATIS + pureMVC

RIA 是什么？

RIA 是富网络应用（Rich Internet Application）的缩写，也即丰富互联网应用程序。它只是一种技术形式而不是具体的技术。

RIA 出现的背景

在 RIA 出现之前，软件开发都是基于 C/S（Client/Server）或 B/S（Browser/Server）架构，但两者各有缺点。

C/S 的主要缺点：

1. 开发、部署成本高
   传统 B/S 结构的软件需要针对不同 OS 开发对应的版本，且软件更新换代的速度越来越快自然成本会很高。
2. 维护成本高
   服务器和客户端都需要维护管理，工作量较大且技术支持复杂。

B/S 的主要缺点：

1. 受限于 HTML 技术，很难像 C/S 那样产生丰富，个性的客户端界面；
2. 存在浏览器兼容性差问题；
3. Server 端负荷较重，响应速度慢；
   绝大多数处理都集中在 Server 端，并且每次响应都要刷新页面（利用 Ajax 技术会有所缓解）。

随着软件的飞速发展，此时需要出现一种能够摒弃上诉缺点的新的技术形式 - RIA 出现了。

目前比较流行的 RIA 技术

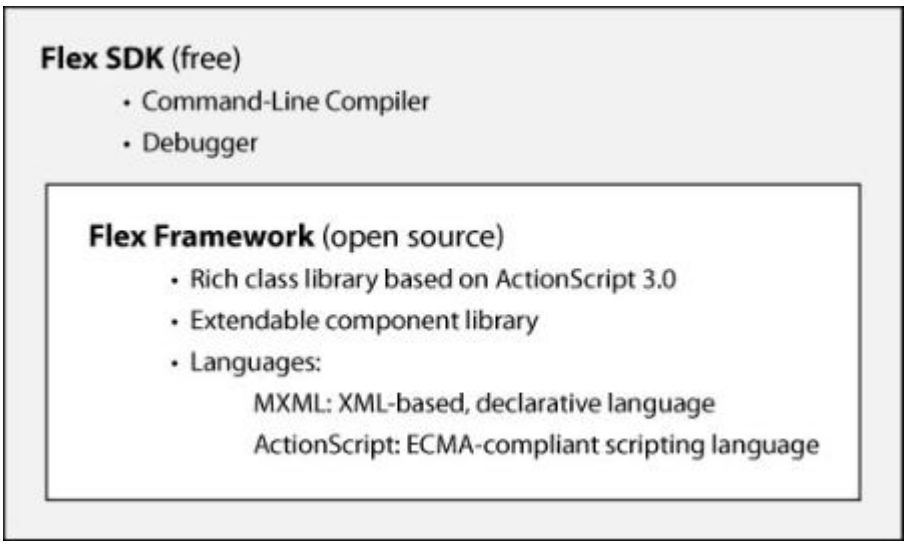
- Adobe 的 Flex
- 微软的 Silverlight
- Sun 的 JavaFX

以上三种技术各有优势，本教程只关注目前应用较广泛的 Flex。

Flex 和 Flex SDK 是什么？

Flex 是一个开源、免费的框架，用于构建在 Adobe® Flash® Player 或 Adobe AIR® runtimes 环境内运行的跨浏览器、桌面和操作系统的富网络应用。

Flex SDK（Flex Software Development Kit）除了包括 Flex 框架以外还包括 compilers（编译器）和 debugger（调试器）等开发工具。（这也意味着没有 Flash Builder 等 IDE 同样可以开发 Flex 应用，但效率会很低。） 三



授权

Mozilla Public License, version 1.1 (MPL)

开发语言

Flex Framework ： Action Script 3.0

开发者

Adobe Systems Incorporated

Flex 应用运行环境 - Adobe® Flash® Player 和 Adobe AIR® Runtimes

两者都是运行环境，前者基于浏览器，后者基于桌面。

可基于这两个环境开发 Flex 应用，但 Adobe® Flash® Player 已非常普及所以现有 Flex 应用绝大多数都是基于 Adobe® Flash® Player 开发。（ Flex 3 要求 Flash Player 9 以上，Flex 4 要求 Flash Player 10 以上）

本系列教程也只针对 Adobe® Flash® Player。

Flex 与 Flash

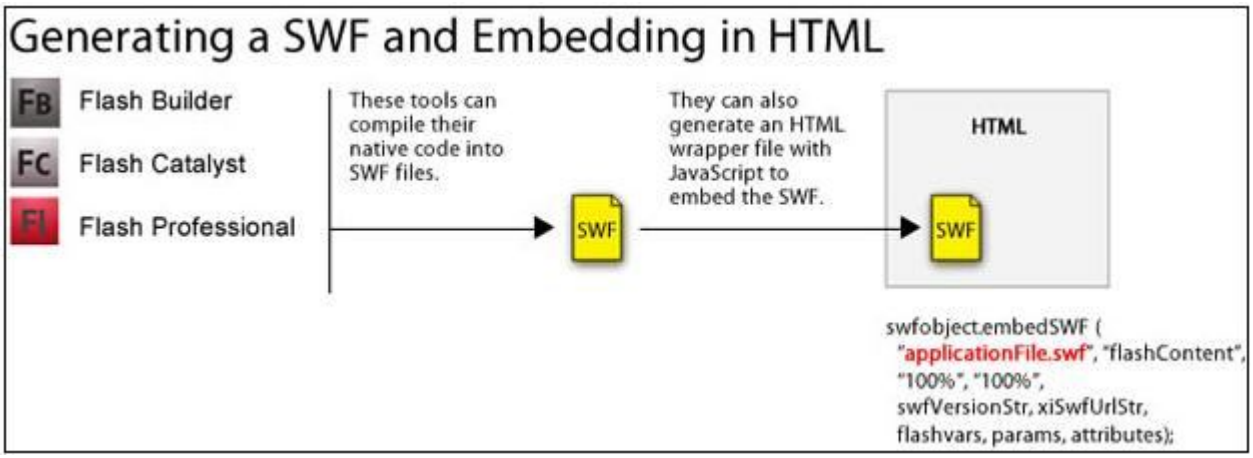
在我们的印象里 Flash 只是设计师用来制作动画的工具，但实际上 Flash 也可以构建富网络应用的，但比较复杂。程序员并不习惯使用画图工具，时间轴和可视化面板等来开发富网络应用，Flex 的出现解决了这一问题。有了 Flex，程序员可以使用 Action Script 和 MXML 编程语言快速开发富网络应用。

Flex 对开发者更具吸引力，而 Flash 更多的是吸引设计人员。

Flex 原理

当你编译一个 Flash 程序时，Flash 开发环境把所有的可视化元素，时间轴指令和 ActionScript 中的业务逻辑编译为 SWF 文件。

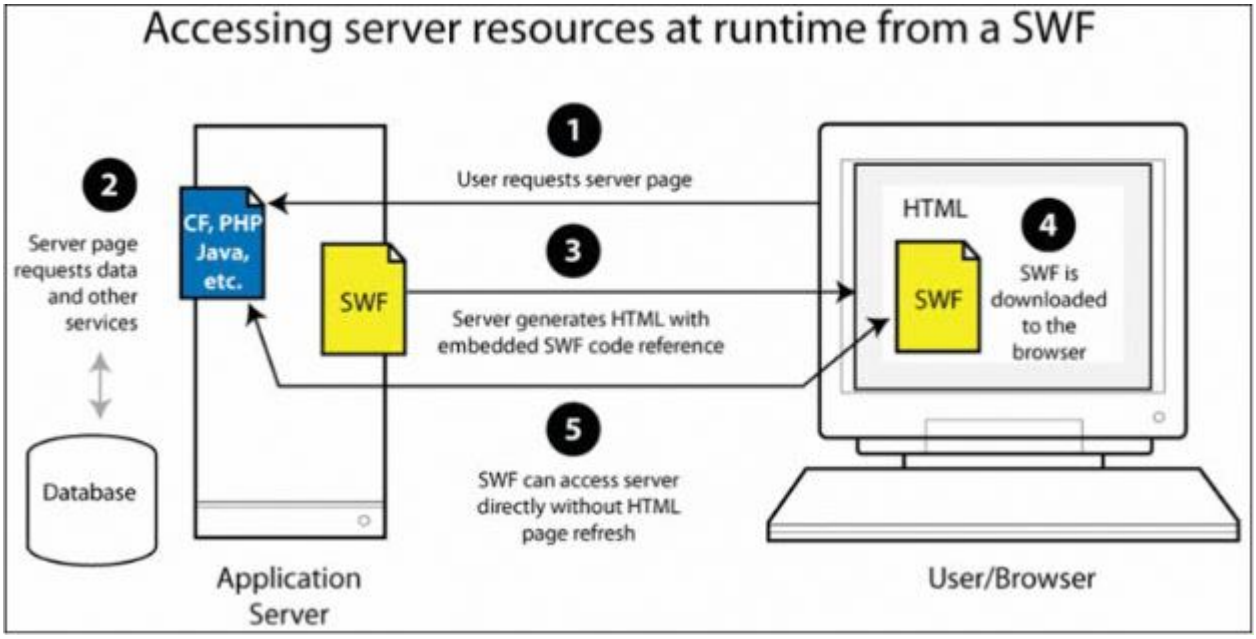
同样地，Flex 程序中的 MXML 和 ActionScript 代码首先全部被转换为 ActionScript 然后编译为 SWF 文件。当你把 SWF 文件部署到服务器上时，使用者可以从服务器获取到这个程序。



Flex 应用的构建形式

尽管用 Flex 开发 RIA 有多种形式，但现在主流的架构是：Flex 作为 Client（客户端），Java、PHP、Asp、Ruby 等技术作为 Server（服务器端）。

本教程之后的内容主要介绍 Flex 与 Java 技术的整合。



Flex 访问服务器端数据的 3 种方式

既然 Flex 可以和 Java 等 Sever 端技术整合，那么它们之间怎样实现通信的呢？Flex 通过 HTTPService，WebService 和 RemoteObject 这 3 个组件实现与 Server 端的通信。

- *HTTPService 组件*

  HTTPService 组件允许你与 HTTP 服务交互，可以是接收 HTTP 请求和发送 HTTP 响应的任何 HTTP URI。

  你可以通过 HTTPService 组件调用任何类型的 Server 端技术，包括 PHP pages, ColdFusion Pages, JavaServer Pages, Java servlets, Ruby on Rails 和 ASP pages。

  HTTPService 组件允许你发送 HTTP GET、POST、HEAD、OPTIONS、PUT、TRACE 和 DELETE 请求，并典型的以 XML 形式返回。

- *WebService 组件*

  WebService 组件允许你访问 WEB 服务。不了解 WEB 服务吗？

- *RemoteObject 组件*（最灵活、最常用的方式）
  RemoteObject 组件允许你访问 Server 端对象的方法，例如 ColdFusion components (CFCs), Java objects, PHP objects 和 .NET objects, 并且不需要把对象配置为 WEB 服务。
  但这种方式与其他 2 种方式不同，它需要中间件（下一节要讲的内容），此时应用和 Server 端对象之间通过 AMF（Action Message Format） 二进制形式传递数据。

Flex 视频教程

一周学会 Flex3 应用开发视频培训（简体中文字幕）

一周学会 Flex4 应用开发视频培训（英文）

Flex 参考文档

Adobe® Flex® 4 Beta 语言参考（简体中文）

Using Flex 4（英文）

Accessing Data with Flex 4（英文）

ADOBE® FLEX® 4 Tutorials（英文）

ActionScript 3.0 Reference for the Adobe Platform（英文）

Tour de Flex

Flex 相关下载

所有 Flex4 文档（约 60 M）

Flex4 SDK

中间件是什么？为什么需要中间件？

上节中我们谈到 Flex 通过 HTTPService，WebService 和 RemoteObject 三个组件与 Server 端技术通信，并且如果用 RemoteObject 那么应用和 Server 端对象之间通过 AMF 二进制形式传递数据。因此就需要额外的软件实现 AMF 协议，这样的软件就是我们所说的中间件。根据不同的 Server 端技术你需要选择不同的中间件。

中间件类型

PHP 中间件

- Zend Framework（开源，免费） 中的 Zend_Amf
- AMFPHP（开源，免费）
- SabreAMF（开源，免费）
- WebORB for PHP（开源，免费）

.NET 中间件

- WebORB for .NET（社区版免费，企业版收费）

Rails 中间件

- WebORB for Rails（开源，免费）

Java 中间件

- WebORB for Java（社区版免费，企业版收费）
- Adobe LiveCycle Data Services ES2（收费）
- Adobe BlazeDS（开源，免费）

对比 Adobe LiveCycle Data Services ES2 和 Adobe BlazeDS

BlazeDS 应用广泛

BlazeDS 是发布于 LGPL v3 许可下的开源，免费项目。在采用 Java 作为 Server 端技术的 Flex 构架中得到越来越多的应用。在之后的教程中也采用它作为中间件。

BlazeDS 文档

BlazeDS 4.0 Installation Guide

BlazeDS 4.0 Javadoc

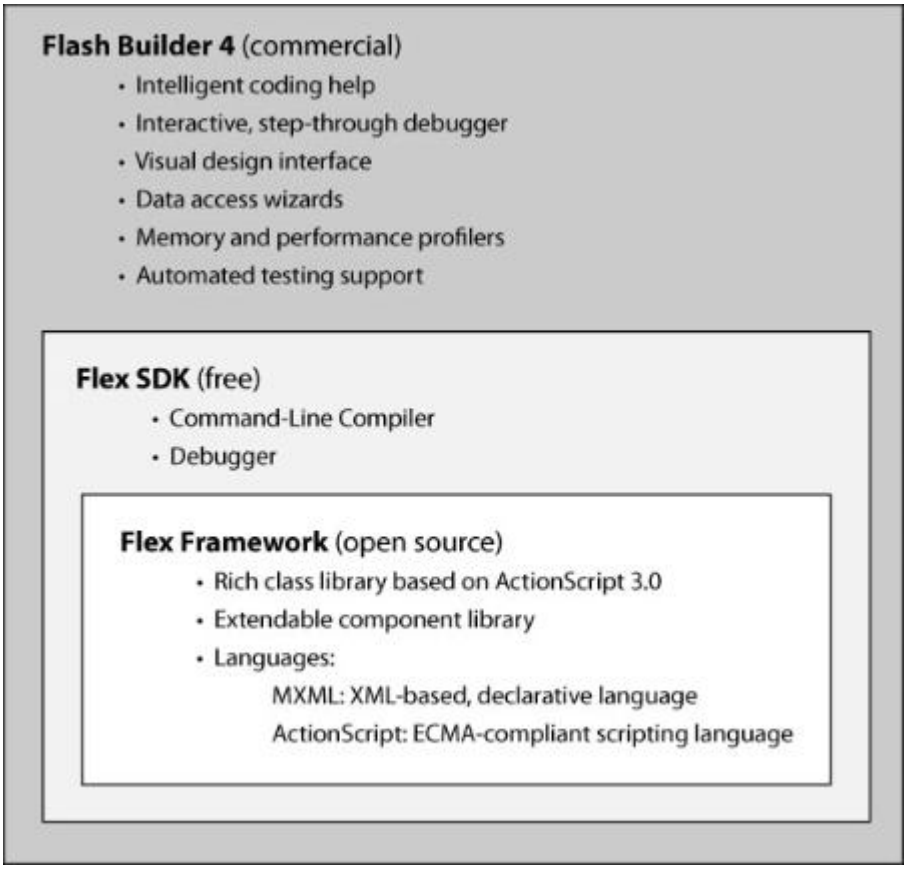
BlazeDS 相关下载

BlazeDS

BlazeDS source code

Flash Builder4 是什么？

Flash Builder4 是一个 Eclipse 插件，版本 4 之前称为 Flex Builder。用于帮助开发者使用 Flex 框架快速开发跨平台的富网络应用。正如下图描述的那样， Flash Builder4 已经集成了 Flex SDK，你不需要再额外下载安装它。



Flash Builder4 版本及授权

有 4 个版本，分别是：

Flash Builder 4 高级版

Flash Builder 4 标准版

Flash Builder 4 标准教育版

Flash Builder 4 高级教育版

其中“Flash Builder 4 标准教育版“注册（需要提供教师资格的相关证明）后可免费下载使用，其他都是收费的（但有 60 天的试用期）。

Flash Builder4 开发者

Adobe Systems Incorporated

Flash Builder4 主要特征

- 强大的编码工具
  借助功能强大、基于 Eclipse™ 的 IDE 进行开发，它包含针对 MXML、ActionScript® 语言和 CSS 的编辑器以及语法颜色、语句完成、代码折叠、交互式点进调试和自动生成常用代码。
- 丰富的可视布局
  使用一个丰富的内建组件库以可视方式设计和预览用户界面布局、外观和行为。扩展内建 Flex 框架组件或根据需要创建新组件。导入使用 Adobe Flash Catalyst™ 交互式设计工具创建的功能性应用程序 UI。
- 以数据为中心的开发
  检查 Java™、PHP、Adobe ColdFusion®、REST 和 SOAP 服务，在新的“Data/Service”（数据/服务）资源管理器中显示方法和属性。使用简单的拖放方法将方法绑定到 UI 组件。
- 交互式数据可视化
  只需使用 Flex Charting 库拖放图表类型并将它链接到数据源，即可创建数据仪表板和交互式数据分析。使用功能强大的 Advanced Datagrid 使用户能浏览复杂数据。
- 外观与样式设计
  使用 CSS 和图形属性编辑器自定义应用程序外观。快速设置最常用的属性，并在“Design”（设计）视图中预览结果。使用新的 Theme Browser（主题浏览器）浏览可用主题，并将它们应用于您的项目。
- 与 Adobe Creative Suite 设计工具集成
  导入使用 Adobe Flash Professional、Illustrator®、Photoshop® 或 Fireworks® 软件创建的设计资源，或导入使用 Flash Catalyst 创建的整个应用程序用户界面。Flash Professional 与 Flash Builder 之间的新工作流程简化了自定义 Flex 组件的导入和更新。
- 对 Adobe AIR 的本机支持
  使用 Flash Builder 4（包括构建、调试、打包和签署 AIR 应用程序所需的全部工具）为 Adobe AIR® 运行时创建应用程序。Adobe AIR 允许您使用与构建浏览器 RIA 相同的技能和代码库快速开发桌面 RIA。
- 代码重构
  通常重命名对类、方法或变量的所有引用，在代码中实现快速导航或对它进行重构。Flash Builder 4 增加了移动重构。
- 功能强大的测试工具（仅限高级版）
  借助内存和性能概要分析器提高应用程序性能，它们可以监视和分析内存消耗情况以及 CPU 周期。还提供对 HP QuickTest Professional 等自动化功能测试工具的支持。
- Network Monitor（网络监视器）（仅限高级版）
  为本地 Flex 应用程序与后端之间通过的全部数据生成一个详细的审计追踪，为调试和性能调试提供协助。
- 高级数据服务
  使用开放源 BlazeDS 添加二进制、高性能、基于 HTTP 的数据传输，或增加 Adobe LiveCycle® Data Services ES2 模块以实现实时数据推送及 pub/sub 消息传递。
- 命令行构建（仅限高级版）
  使用新的命令行构建功能实现构建流程自动化。
- Flex 单元测试集成（仅限高级版）
  使用 Flex 单元测试框架实现功能测试自动化。
- ASDoc 支持
  使用 ASDoc 在 MXML 和 ActionScript 编辑器中显示注释。

对比 Flash Builder4 标准版，Flash Builder4 高级版，Flex Builder3 及 Flex4 SDK 的功能

Flash Builder4 的系统要求（软件）

- 操作系统

  Microsoft® Windows® XP with Service Pack 3

  Windows Vista® Ultimate or Enterprise (32 or 64 bit running in 32-bit mode)

  Windows Server® 2008 (32 bit)

  Windows 7 (32 or 64 bit running in 32-bit mode)
- Java™ 虚拟机 (32 位)

  IBM® JRE 1.5

  Sun™ JRE 1.5

  IBM JRE 1.6

  Sun JRE 1.6
- Eclipse 3.4.2 或 3.5 (插件安装)

在 Windows 操作系统上安装 Flash Builder4

Flash Builder4 安装文件有两种形式：”独立安装文件“（即，安装文件已经包含 Eclipse）和“插件安装文件”（不包含 Eclipse）。以下只介绍插件形式的安装。

第一步：下载相关软件

1. 下载 JDK 6（76.67 MB）
2. 下载 Eclipse IDE for Java EE Developers （基于 Eclipse 3.5 SR2，190 MB）
3. 下载 Flash Builder 4 高级版

- 免费创建一个 Adobe 账号
- 创建账号成功后会显示“Download Adobe Flash Builder 4 Premium”页面
- 在下拉菜单中选择“English | Eclipse Plug-in Windows | 403.3 MB”
- 点击“Download”按钮下载

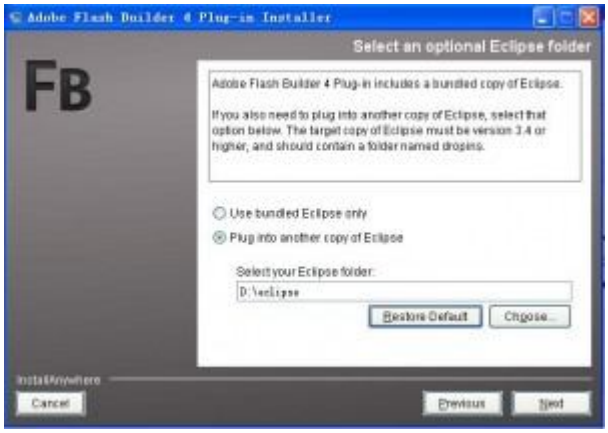
第二步：安装

1. 安装 JDK；
2. 解压 Eclipse 到指定目录，确保 Eclipse 能正常启动；
3. 安装 Flash Builder 插件之前关闭 Eclipse 和所有浏览器窗口；
4. 运行 Flash Builder 插件；

- 选择安装前的解压目录

- 选择语言（这只是安装向导的语言，安装后 IDE 中菜单的语言依据操作系统自动识别）

- 之后是介绍信息，点“Next”
- 之后是许可协议，选择同意后点“Next”
- 之后选择安装路径，点“Next”
- 指定上面安装的 Eclipse 位置，点“Next”



- 待执行完启动 Eclipse ，在新建项目弹出窗口中会有“Flash Builder”一项，至此安装完毕。

在继续本教程之前你需要准备好以下事项：

- 已安装 JDK（截稿时最新版 JDK 6 Update 20），并设定好 JAVA_HOME 环境变量（Tomcat 启动需要）；
- 下载 Tomcat（截稿时最新版 Tomcat 6.0.26）解压到适当目录，确保 Tomcat 启动正常；
- 已在“Eclipse IDE for Java EE Developers“（截稿时最新版基于 Eclipse 3.5）基础上正确安装了 Flash Builder 4 插件（可试用 60 天）；
- 下载最新版 BlazeDS（截稿时最新版 blazeds 4.0.0.14931），解压备用；
- 已对 Flex 基本了解。

第一步：添加 Apache Tomcat 运行时

1. 从 Window 菜单选择 **Preferences**
2. 在 Preferences 对话框中展开 **Server**，然后选择 **Runtime Environments**
3. 在 Server Runtime Environments 页点击 **Add**，打开 New Server Runtime Environment 对话框
4. 在 New Server Runtime Environment 页展开 **Apache**
5. 从下面支持的 Apache Tomcat 服务器中选择一个（我用的是 Apache Tomcat v6.0）：
   - Apache Tomcat v3.2
   - Apache Tomcat v4.0

- Apache Tomcat v4.1
- Apache Tomcat v5.0
- Apache Tomcat v5.5
- Apache Tomcat v6.0

6. 当你添加一个 Server Runtime Environment 时，默认会创建一个 Server 并作为实体添加在 Servers 视图(View)内。如果你只想添加 server runtime environment 而不想在 Servers 视图内创建 Server，那么清除 **Create a new local server** 前的多选框（我们选上这个多选框免得之后手动创建 Server）
7. 当你点击 **Next** 时会打开 Tomcat Server 页
8. 在 **Tomcat installation directory** 项选择 Apache Tomcat 目录（例如：D:\apache-tomcat-6.0.26）
9. 点击 Tomcat Server 页的 **Finish**
10. 点击 Server Runtime Environment 页的 **OK**（如 6 所述，此时 Servers 视图内会显示一个 Server）

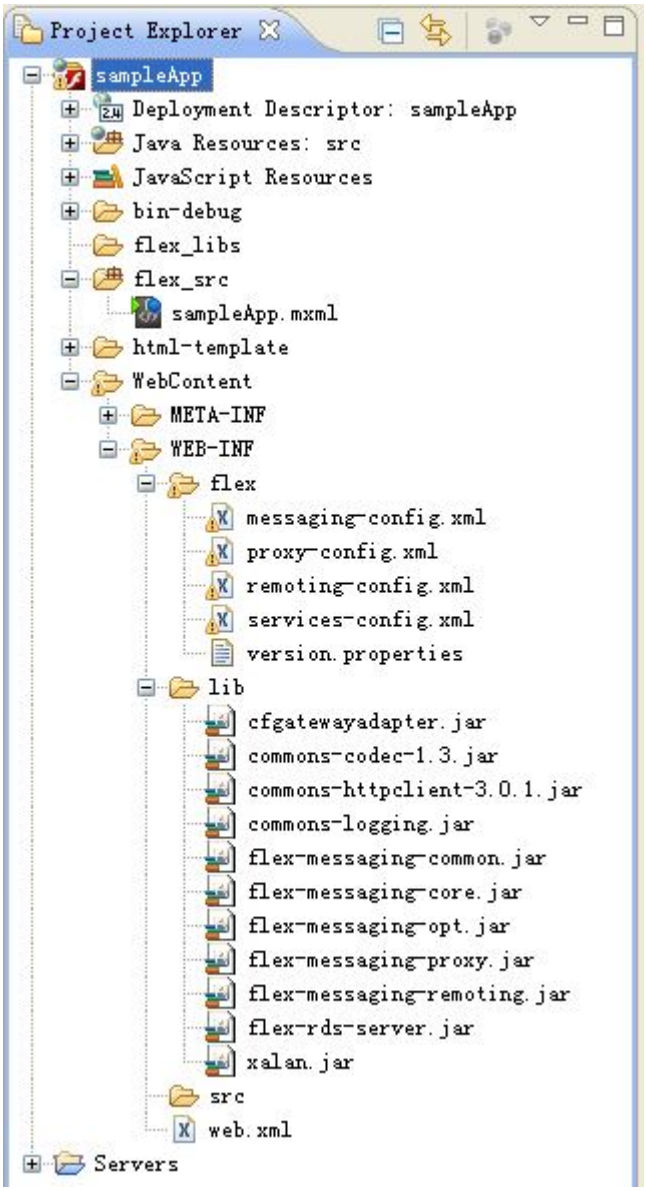
第二步：使用 WTP 创建 Java/Flex 组合项目

1. 切换到 Java EE 视图（perspective）
2. 在 Project Explorer 视图（View）内点击右键，选择 **New** 项
3. 选择子菜单中的 **Project…** 项，打开 New Project 对话框
4. 展开 **Flash Builder**，选择“ **Flex 项目**“（因为我是中文系统所以 Flash Builder 的菜单项都显示为中文，尽管我的 Eclipse 为英文）
5. 点击 **Next**
6. 在“新建 Flex 项目”对话框中对应以下几项：
   - 项目名：sampleApp
   - 项目位置：默认即可
   - 应用程序类型：**Web**
   - Flex SDK 版本：默认即可
   - 应用程序服务器类型：J2EE
   - 远程对象访问服务：选择 **BlazeDS**
   - 使用 WTP 创建 Java/Flex 组合项目：**选上**
7. 点击 **Next**
8. 在“配置 J2EE 服务器”页对应以下两项，其他项默认即可
   - 目标运行时：Apache Tomcat v6.0
   - BlazeDS WAR 文件：选择上面准备好的 blazeds.war
9. 点击 **Next**

   - 输出文件夹 URL：http://localhost:8080/sampleApp

10. 点击 **Finish**
11. 按提示切换到 Flash 视图（perspective），向 sampleApp.mxml 中拖入 DataGrid 控件以备后用

项目结构图：



第三步：运行 sampleApp 项目

1. 重新切换到 Java EE 视图（perspective）
2. 在 Servers 视图（View）中的 Server 内添加 sampleApp 项目
3. 启动此 Server
4. 以“**Web 应用程序**”的方式运行项目
5. 如果看到刚才拖入的表格，恭喜你成功了 

第四步：使 Flex 以 RemoteObject 的方式与 Java 交互

是不是觉得表格太空洞了？下面我们用它显示雇员信息，借此演示 Flex 与 Java 的交互过程。

1. 创建 com.sample 包
2. 在包内创建两个类： 雇员类 Employee，雇员的 Service 类 EmployeeService

```
3. package com.sample;
4.
5. public class Employee {
```

```
    private String name;
    private int age;
    private String email;

    public Employee(String name, int age, String email) {
     this.name = name;
     this.age = age;
     this.email = email;
    }

    public void setName(String name) {
     this.name = name;
    }

    public String getName() {
     return name;
    }

    public void setAge(int age) {
     this.age = age;
    }

    public int getAge() {
     return age;
    }

    public void setEmail(String email) {
     this.email = email;
    }

    public String getEmail() {
     return email;
    }
}
package com.sample;

import java.util.ArrayList;

public class EmployeeService {
    public ArrayList getList() {
     ArrayList tempList = new ArrayList();

     for (int i = 1; i <= 30; i++) {
```

```
            tempList.add(new Employee("Smith"+i, 20+i,
"smith"+i+"@test.com"));
        }

        return tempList;
    }
}
```

39. 在 remoting-config.xml 文件中定义 EmployeeService 对应的 destination

```
<destination id="employeeServiceDest">
    <properties>
        <source>com.sample.EmployeeService</source>
    </properties>
</destination>
```

44. 在 sampleApp.mxml 中通过 employeeServiceDest 调用 EmployeeService 的 getList() 方法
    - 定义显示雇员信息的表格

```
<mx:DataGrid x="32" y="25" width="400"
dataProvider="{employeeList}">
    <mx:columns>
        <mx:DataGridColumn headerText="Name" dataField="name"/>
        <mx:DataGridColumn headerText="Age" dataField="age"/>
        <mx:DataGridColumn headerText="Email"
dataField="email"/>
    </mx:columns>
</mx:DataGrid>
```

    - 定义 RemoteObject 组件

```
<fx:Declarations>
    <mx:RemoteObject id="employeeServiceRO"
destination="employeeServiceDest"
                     result="resultHandler(event);"
                     fault="faultHandler(event);"/>
</fx:Declarations>
```

    - 定义相关函数

```
<fx:Script>
    <![CDATA[
        import mx.controls.Alert;
        import mx.rpc.events.ResultEvent;
        import mx.rpc.events.FaultEvent;

        [Bindable]
        private var employeeList:Object;
```

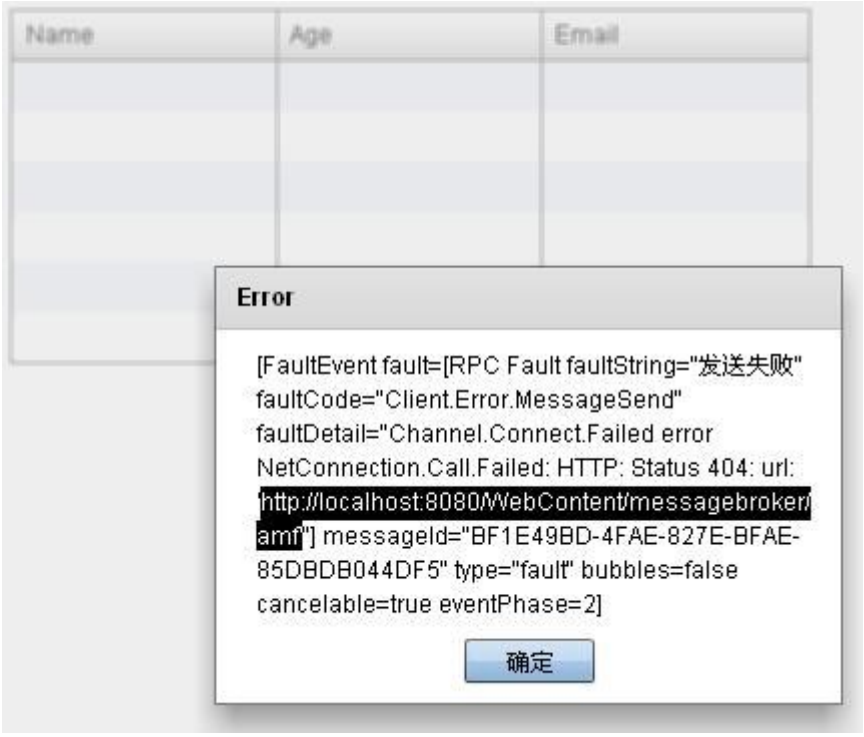
```
        private function init():void {
            employeeServiceRO.getList();
        }

        private function resultHandler(event:ResultEvent):void {
            employeeList = event.result;
        }

        private function faultHandler(event:FaultEvent):void {
            //Alert.show(event.fault.faultString, 'Error');
            Alert.show(event.toString(), 'Error');
        }
    ]]>
</fx:Script>
```

当 Application 完成构建后立即触发 init() 方法，以实现对 Server 端 Java 的调用

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               minWidth="955" minHeight="600"
               creationComplete="init();">
```

第五步：重新运行 sampleApp 项目

很不幸  ，RPC 过程失败了(Adobe Flash Builder 的 Bug 吗？)



注意到上图用黑色背景标注的内容了吧？本应该是 sampleApp，但现在却成了 WebContent。

我们需要处理一下：

打开项目根文件夹下的 **.flexProperties** 文件，更改其中的 serverContextRoot=”/WebContent” 为 **serverContextRoot=”/sampleApp”**。

OK，再运行试试吧（别忘了刷新项目）。

附件： sampleApp-basic.7z

Spring BlazeDS Integration 是什么？

Spring BlazeDS Integration 是 SpringSource 的开源项目，用于整合 Spring 与 BlazeDS。

为什么需要 Spring BlazeDS Integration？

正如“Flex4 系列教程之六”介绍的：不使用 Spring BlazeDS Integration 同样可以整合 Spring 与 BlazeDS。但这种整合方式不自然，需要额外维护一个 BlazeDS 配置文件，Spring BlazeDS Integration 会改善这种处境。

Spring BlazeDS Integration 需要的软件环境：

- Java 5 或更高
- Spring 2.5.6 或更高
- BlazeDS 3.2 或更高

Spring BlazeDS Integration 特征

- MessageBroker（BlazeDS 的核心组件）被配置为 Spring 管理的 Bean
- Flex 客户端发出的 HTTP 消息通过 Spring 的 DispatcherServlet 路由给 MessageBroker
- Remote objects 以 Spring 的方式配置在 Spring 配置文件内

注意事项：

以下内容基于“Flex4 系列教程之五”中创建的 sampleApp 项目。

在继续本教程之前你需要准备好以下事项：

- 下载 Spring Framework（截稿时最新版 spring-framework 3.0.2），解压备用
- 下载 Spring Framework dependencies（截稿时最新版 spring-framework 3.0.2 dependencies），解压备用
- 下载 Spring BlazeDS Integration（截稿时最新版 spring-flex 1.0.3），解压备用

第一步：准备所需 jar 包

将以下 3 部分 jar 包拷贝到 sampleApp 项目的 lib 下

1. Spring Framework

   org.springframework.aop-3.0.2.RELEASE.jar

   org.springframework.asm-3.0.2.RELEASE.jar

   org.springframework.beans-3.0.2.RELEASE.jar

   org.springframework.context-3.0.2.RELEASE.jar

   org.springframework.core-3.0.2.RELEASE.jar

   org.springframework.expression-3.0.2.RELEASE.jar

   org.springframework.web.servlet-3.0.2.RELEASE.jar

   org.springframework.web-3.0.2.RELEASE.jar

2. Spring Framework dependencies

   org.aopalliance 内的 com.springsource.org.aopalliance-1.0.0.jar

   edu.emory.mathcs.backport 内的 com.springsource.edu.emory.mathcs.backport-3.0.0.jar

   net.sourceforge.cglib 内的 com.springsource.net.sf.cglib-2.2.0.jar

   [注：]Spring 3 的依赖包用 Ivy 或 Maven 管理会很方便，完成本系列教程后我会单独整理这部分。暂且手动拷贝吧

3. Spring BlazeDS Integration
   org.springframework.flex-1.0.3.RELEASE.jar

第二步：修改 web.xml 文件

将 web.xml 内所有 Flex 相关配置删除掉，添加以下内容（改用 Spring web 应用的前端控制器处理所有应用请求）

```
<servlet>
  <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/web-application-config.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
  <url-pattern>/messagebroker/*</url-pattern>
</servlet-mapping>
```

第三步：配置 web-application-config.xml

1. 创建应用上下文配置文件 web-application-config.xml

```
2. <?xml version="1.0" encoding="UTF-8"?>
3. <beans xmlns="http://www.springframework.org/schema/beans"
4.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.
   xsi:schemaLocation="http://www.springframework.org/schema/beans
6.
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7.
8. </beans>
```

9. 为了使用 Spring BlazeDS Integration 的 tag，增加命名空间

```
10.<?xml version="1.0" encoding="UTF-8"?>
11.<beans xmlns="http://www.springframework.org/schema/beans"
12.        xmlns:flex="http://www.springframework.org/schema/flex"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

   xsi:schemaLocation="http://www.springframework.org/schema/beans
13.
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/flex
```

```
    http://www.springframework.org/schema/flex/spring-flex-1.0.xsd">
14.
    </beans>
```

15. 为了把请求路由给 MessageBroker，添加以下 tag

```
    <flex:message-broker />
```

16. 定义 Bean，并用 remoting-destination tag 把它暴露给 Flex

```
17.<bean id="employeeServiceDest" class="com.sample.EmployeeService">
18.   <flex:remoting-destination />
    </bean>
```

第四步：删除多余的 Flex 配置文件

删除 services-config.xml 以外的所有 Flex 配置文件（你认为它们还有必要保留吗？ ）。但千万别忘记在 services-config.xml 内重新定义默认 channel（原来定义在 remoting-config.xml 内）：

修改 services-config.xml，替换

```
<services>
  <service-include file-path="remoting-config.xml" />
  <service-include file-path="proxy-config.xml" />
  <service-include file-path="messaging-config.xml" />
</services>
```

为

```
<services>
  <default-channels>
    <channel ref="my-amf"/>
  </default-channels>
</services>
```

第五步：重新运行 sampleApp 项目

运行结果与整合之前相同吧

附件： sampleApp-7.7z

Spring BlazeDS Integration 是什么？

Spring BlazeDS Integration 是 SpringSource 的开源项目，用于整合 Spring 与 BlazeDS。

为什么需要 Spring BlazeDS Integration?

正如“Flex4 系列教程之六”介绍的：不使用 Spring BlazeDS Integration 同样可以整合 Spring 与 BlazeDS。但这种整合方式不自然，需要额外维护一个 BlazeDS 配置文件，Spring BlazeDS Integration 会改善这种处境。

Spring BlazeDS Integration 需要的软件环境：

- Java 5 或更高
- Spring 2.5.6 或更高
- BlazeDS 3.2 或更高

Spring BlazeDS Integration 特征

- MessageBroker（BlazeDS 的核心组件）被配置为 Spring 管理的 Bean
- Flex 客户端发出的 HTTP 消息通过 Spring 的 DispatcherServlet 路由给 MessageBroker

- Remote objects 以 Spring 的方式配置在 Spring 配置文件内

注意事项：

以下内容基于“Flex4 系列教程之五”中创建的 sampleApp 项目。

在继续本教程之前你需要准备好以下事项：

- 下载 Spring Framework（截稿时最新版 spring-framework 3.0.2），解压备用
- 下载 Spring Framework dependencies（截稿时最新版 spring-framework 3.0.2 dependencies），解压备用
- 下载 Spring BlazeDS Integration（截稿时最新版 spring-flex 1.0.3），解压备用

第一步：准备所需 jar 包

将以下 3 部分 jar 包拷贝到 sampleApp 项目的 lib 下

1. Spring Framework

   org.springframework.aop-3.0.2.RELEASE.jar

   org.springframework.asm-3.0.2.RELEASE.jar

   org.springframework.beans-3.0.2.RELEASE.jar

   org.springframework.context-3.0.2.RELEASE.jar

   org.springframework.core-3.0.2.RELEASE.jar

   org.springframework.expression-3.0.2.RELEASE.jar

   org.springframework.web.servlet-3.0.2.RELEASE.jar

   org.springframework.web-3.0.2.RELEASE.jar

2. Spring Framework dependencies

   org.aopalliance 内的 com.springsource.org.aopalliance-1.0.0.jar

   edu.emory.mathcs.backport 内的 com.springsource.edu.emory.mathcs.backport-3.0.0.jar

   net.sourceforge.cglib 内的 com.springsource.net.sf.cglib-2.2.0.jar

   [注：]Spring 3 的依赖包用 Ivy 或 Maven 管理会很方便，完成本系列教程后我会单独整理这部分。暂且手动拷贝吧 

3. Spring BlazeDS Integration

   org.springframework.flex-1.0.3.RELEASE.jar

第二步：修改 web.xml 文件

将 web.xml 内所有 Flex 相关配置删除掉，添加以下内容（改用 Spring web 应用的前端控制器处理所有应用请求）

```
<servlet>
  <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</ser
vlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/web-application-config.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
```

```
  <url-pattern>/messagebroker/*</url-pattern>
</servlet-mapping>
```

第三步：配置 web-application-config.xml

1. 创建应用上下文配置文件 web-application-config.xml

```
2. <?xml version="1.0" encoding="UTF-8"?>
3. <beans xmlns="http://www.springframework.org/schema/beans"
4.        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.
   xsi:schemaLocation="http://www.springframework.org/schema/beans
6.
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7.
8. </beans>
```

9. 为了使用 Spring BlazeDS Integration 的 tag，增加命名空间

```
10.<?xml version="1.0" encoding="UTF-8"?>
11.<beans xmlns="http://www.springframework.org/schema/beans"
12.       xmlns:flex="http://www.springframework.org/schema/flex"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

   xsi:schemaLocation="http://www.springframework.org/schema/beans
13.
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
          http://www.springframework.org/schema/flex

   http://www.springframework.org/schema/flex/spring-flex-1.0.xsd">
14.
   </beans>
```

15. 为了把请求路由给 MessageBroker，添加以下 tag

```
   <flex:message-broker />
```

16. 定义 Bean，并用 remoting-destination tag 把它暴露给 Flex

```
17.<bean id="employeeServiceDest" class="com.sample.EmployeeService">
18.  <flex:remoting-destination />
   </bean>
```

第四步：删除多余的 Flex 配置文件

删除 services-config.xml 以外的所有 Flex 配置文件（你认为它们还有必要保留吗？ ）。但千万别忘记在 services-config.xml 内重新定义默认 channel（原来定义在 remoting-config.xml 内）：

修改 services-config.xml，替换

```
<services>
  <service-include file-path="remoting-config.xml" />
```

```
  <service-include file-path="proxy-config.xml" />
  <service-include file-path="messaging-config.xml" />
</services>
```

为

```
<services>
  <default-channels>
    <channel ref="my-amf"/>
  </default-channels>
</services>
```

第五步：重新运行 sampleApp 项目

运行结果与整合之前相同吧 

附件：  sampleApp-7.7z

注意事项：

以下内容基于“Flex4 系列教程之七”中最后形成的 sampleApp 项目。Spring 2.5.6 之前版本的整合方式与本篇基本相同，不再重复。

配置数据源

是时候改用 DB 存储 sampleApp 中的雇员（Employee）信息了。我们采用 Mysql，并假定你已安装它（截稿时最新版 MySQL Community Server 5.1.47）。

1. 准备数据库
   - 创建数据库 sample
   - 创建表 employees

```
CREATE TABLE IF NOT EXISTS `employees` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(20) COLLATE utf8_unicode_ci NOT NULL DEFAULT '',
  `age` int(2) NOT NULL DEFAULT '0',
  `email` varchar(100) COLLATE utf8_unicode_ci NOT NULL DEFAULT '',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

   - 插入一些模拟数据
2. 准备所需组件
   - 下载以下组件解压备用

     Commons DBCP（截稿时最新版 commons-dbcp 1.4）

     Commons Pool（截稿时最新版 commons-pool 1.5.4）

     Connector/J（截稿时最新版 mysql-connector-java 5.1.12）
   - 拷贝 jar 包

     将解压后的 mysql-connector-java-5.1.12-bin.jar，commons-pool-1.5.4.jar 和 commons-dbcp-1.4.jar 拷贝到 sampleApp 的 lib 下
3. 使用单独文件存储 DB 驱动等信息
   - 创建 resources 包
   - 在包下创建 jdbc.properties 文件，输入你的 DB 信息

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://域名或 IP:端口/sample
jdbc.username=用户名
jdbc.password=密码
```

4. 修改 web-application-config.xml 文件
   - 增加命名空间

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:flex="http://www.springframework.org/schema/flex"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/flex
http://www.springframework.org/schema/flex/spring-flex-1.0.xsd
       http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
```

   - 引入 jdbc.properties 文件

```
<context:property-placeholder
location="classpath:resources/jdbc.properties"/>
```

   - 配置数据源

```
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
  <property name="driverClassName"
value="${jdbc.driverClassName}"/>
  <property name="url" value="${jdbc.url}"/>
  <property name="username" value="${jdbc.username}"/>
  <property name="password" value="${jdbc.password}"/>
</bean>
```

   - 向 EmployeeService 中注入 dataSource

```
<bean id="employeeServiceDest">
  <flex:remoting-destination />
  <property name="dataSource" ref="dataSource"/>
```

```
</bean>
```

5. 修改 EmployeeService
   - 追加 dataSource 属性

```
private BasicDataSource dataSource;

public void setDataSource(BasicDataSource dataSource) {
  this.dataSource = dataSource;
}

public BasicDataSource getDataSource() {
  return dataSource;
}
```

   - 修改 getList 方法

```
public ArrayList<Employee> getList() {
    ArrayList<Employee> tempList = new ArrayList<Employee>();

    try {
        Connection conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement("SELECT * FROM `employees`");
        ResultSet rs = ps.executeQuery();

        while (rs.next()) {
            Employee employee = new Employee();
            employee.setName(rs.getString("name"));
            employee.setAge(rs.getInt("age"));
            employee.setEmail(rs.getString("email"));
            tempList.add(employee);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return tempList;
}
```

   [注：]以上创建 Employee 实例的方法需要在 Employee 中追加默认构造器。

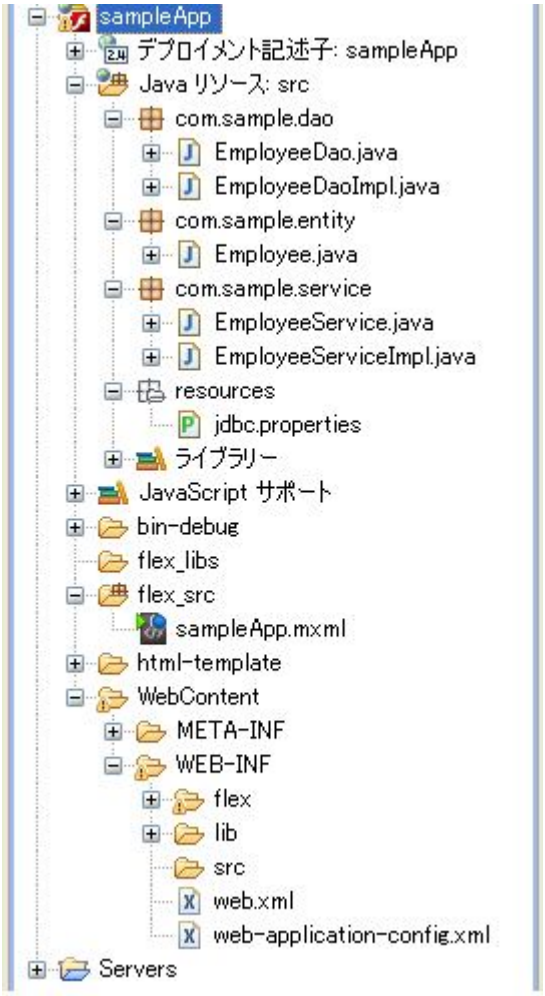
6. 运行 sampleApp

重构 server 端

你是否觉得目前的 server 端比较混乱？是的，我们需要重构它。

1. 按以下原则重构 server 端
   - 面向接口编成
   - 分离业务逻辑层和持久层

重构后结构图（代码请参照之后的附件）：



2. 重新配置 web-application-config.xml

将以下内容

```
<bean id="employeeServiceDest" class="com.sample.EmployeeService">
  <flex:remoting-destination />
  <property name="dataSource" ref="dataSource"/>
</bean>
```

更改为

```
<bean id="employeeServiceDest"
class="com.sample.service.EmployeeServiceImpl">
  <flex:remoting-destination />
  <property name="employeeDao" ref="employeeDao"/>
</bean>

<bean id="employeeDao" class="com.sample.dao.EmployeeDaoImpl">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

6. 运行 sampleApp

附件： sampleApp-8-1.7z

整合 iBATIS 2.3

spring-framework 3.0 目前只支持 iBATIS 2.x，期望它尽快支持 iBATIS 的高版本 - MyBatis

1. 准备所需组件
   - 下载解压 iBATIS 2.3.4，将 ibatis-2.3.4.726.jar 拷贝到 lib 下
   - 将以下 spring framework 的 jar 文件拷贝到 lib 下

     org.springframework.jdbc-3.0.2.RELEASE.jar

     org.springframework.orm-3.0.2.RELEASE.jar

     org.springframework.transaction-3.0.2.RELEASE.jar
2. 通过 Spring 管理 iBATIS
   - 追加以下代码

```
<bean id="sqlMapClient"
class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
  <property name="configLocation"
value="WEB-INF/sqlmap-config.xml"/>
  <property name="dataSource" ref="dataSource"/>
</bean>
```

   - 修改

```
<bean id="employeeDao" class="com.sample.dao.EmployeeDaoImpl">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

   为

```
<bean id="employeeDao" class="com.sample.dao.EmployeeDaoImpl">
  <property name="sqlMapClient" ref="sqlMapClient"/>
</bean>
```

3. 创建 iBATIS 映射文件
   - 创建包 com.sample.dao.ibatis
   - 创建映射文件 employees.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
    PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap>
  <select id="getEmployees"
resultClass="com.sample.entity.Employee">
    SELECT * FROM employees
  </select>
</sqlMap>
```

4. 创建 iBATIS 配置文件

   在 WEB-INF 下创建 sqlmap-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMapConfig
    PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">
<sqlMapConfig>
    <sqlMap resource="com/sample/dao/ibatis/employees.xml"/>
</sqlMapConfig>
```

11. 修改 Dao 的实现类 EmployeeDaoImpl

```
package com.sample.dao;

import java.util.ArrayList;

import
org.springframework.orm.ibatis.support.SqlMapClientDaoSupport;
import com.sample.entity.Employee;

public class EmployeeDaoImpl extends SqlMapClientDaoSupport
    implements EmployeeDao {
    public ArrayList<Employee> getList() {
        return (ArrayList<Employee>)getSqlMapClientTemplate().
                queryForList("getEmployees");
    }
}
```

25. 重构 Dao 和 Service 中 getList 方法的返回值类型
    由 ArrayList<Employee> 改为 List<Employee>
26. 重新运行 sampleApp
    没问题吧 

怎么样？现在整个架构感觉舒服多了吧  但是否感觉缺点什么？对，事务！我们下一篇搞定它。

附件：  sampleApp-8-2.7z

注意事项：

以下内容基于“Flex4 系列教程之八”中最后形成的 sampleApp 项目。

准备所需 jar 包

将以下 jar 包拷贝到 sampleApp 项目的 lib 下

1. Spring Framework dependencies
   org.aspectj 内的 com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

Spring Framework 的事务管理类型

综合性的事务支持是 Spring Framework 倍受欢迎的原因之一。Spring Framework 有两种事务管理方式：声明式事务管理和编程式事务管理。前者因为“对代码影响最小“和“非侵入性”而较为流行。

配置声明式事务

Spring Framework 的声明式事务通过 AOP 思想实现。

1. 制定事务管理规则

   常见的是对 Service 层进行事务管理，我们也不例外。我们约定对 Service 接口内定义的方法实行以下事务上下文语义：

   - 以 get 开头的方法：只读（read-only）
   - 以 insert 开头的方法：读写（read-write）
   - 以 update 开头的方法：读写（read-write）
   - 以 delete 开头的方法：读写（read-write）

2. 配置

   向 web-application-config.xml 文件追加以下内容：

   - 配置 PlatformTransactionManager bean，用于驱动事务

```
<bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

   - 配置 advice

     增加命名空间

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:flex="http://www.springframework.org/schema/flex"
xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/flex
http://www.springframework.org/schema/flex/spring-flex-1.0.xsd
       http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

     追加 advice

```
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
```

```
    <tx:method name="get*" read-only="true"/>
    <tx:method name="insert*"/>
    <tx:method name="update*"/>
    <tx:method name="delete*"/>
  </tx:attributes>
</tx:advice>
```

- 配置切入点

  增加命名空间

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:flex="http://www.springframework.org/schema/flex"

xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/flex

http://www.springframework.org/schema/flex/spring-flex-1.0.xsd
       http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
       http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

  追加切入点配置

```
<aop:config>
  <aop:pointcut id="serviceOperation" expression="execution(* *..*Service.*(..))"/>
  <aop:advisor advice-ref="txAdvice"
pointcut-ref="serviceOperation"/>
</aop:config>
```

[注：]加粗部分为 AspectJ 切入点表达式，我会在本系列教程之后详细介绍。

3. 上面的配置实际上做了什么？

   它们被用于围绕 Service 对象创建相应的事务代理，此代理会用 advice 配置。这样当 Service 中的方法在代理上执行时相应的事务也就启动了。

4. 运行 sampleApp

附件： sampleApp-9.7z

Cairngorm 2 概述

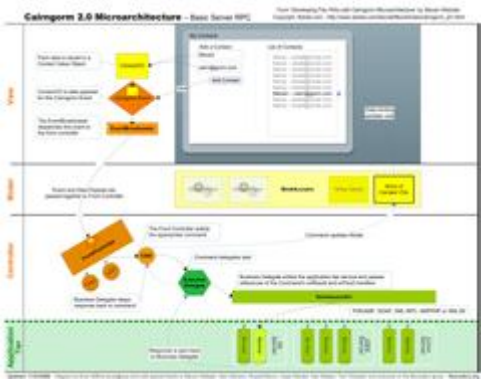
1. Cairngorm 2 是什么？

   Cairngorm 2 是一个简单规范的 MVC 模式框架。

2. Cairngorm 2 的两个版本

   Cairngorm 2 分为“普通版“和“企业版“，后者依赖于 LiveCycle Data Services ，所以我们只探讨普通版（因为我们用的是 BlazeDS ）。

3. Cairngorm 2 原理图示



4. Cairngorm 2 教程
   - Introduction to Cairngorm2
   - Developing Flex RIAs with Cairngorm microarchitecture

为什么不整合 Cairngorm 3 ？

Cairngorm 3 已经不是 Cairngorm 2 的升级。它由跨框架的“指导原则”、“工具”和“库”三部分组成，目的在于帮助开发者应用 Flex 和第三方框架。

开始整合

1. 注意事项：

   以下内容基于“Flex4 系列教程之九”中最后形成的 sampleApp 项目。

2. 准备所需组件

   下载 Cairngorm 2 普通版，将解压后的 Cairngorm.swc 拷贝到 flex_libs 文件夹。

3. 在 flex_src 下创建以下文件夹
   - business ：放置 Delegate 类和 ServiceLocator 文件
   - command ：放置 Command 类
   - event ：放置 Event 类
   - vo ：放置 VO 类
   - util ：放置工具类
   - model ：放置 Model 类
   - view ：放置视图文件（即 mxml 文件）
   - controller：放置 Controller 类

4. 在继续之前，还是回顾一下 Cairngorm 2 的原理吧（总觉得 Cairngorm 2 概述中的图示有点乱 ）

   记住一点： Cairngorm 2 是事件驱动的。

所以要显示存储在数据库中的职员信息需经过以下过程：

- 触发一个 Event；
- 控制器依据 Event ID 找到对应的 Command；
- Command 调用 Delegate(Delegate 又调用 Server 端对象)，并把返回的职员信息存储到 Model 中的某个属性；
- 把 Mxml 文件中的 DateGrid 组件与上述 Model 中的属性绑定。因为绑定是动态的，所以一旦属性值发生变化 DateGrid 内容会立即体现。

5. 在 business 下创建 Services.mxml，以统一管理远程对象

```
<?xml version="1.0" encoding="utf-8"?>
<cairngorm:ServiceLocator
            xmlns:mx="http://www.adobe.com/2006/mxml"
            xmlns:cairngorm="http://www.adobe.com/2006/cairngorm">

    <mx:RemoteObject id="employeeServiceRO"
destination="employeeServiceDest" />

</cairngorm:ServiceLocator>
```

13. 在 business 下创建 LoadEmployeesDelegate 代理，调用远程对象

```
package business
{
    import com.adobe.cairngorm.business.ServiceLocator;

    import mx.rpc.AsyncToken;
    import mx.rpc.IResponder;

    public class LoadEmployeesDelegate
    {
        private var responder:IResponder;
        private var service:Object;

        public function LoadEmployeesDelegate(responder:IResponder)
{
            this.responder = responder;
            this.service =
ServiceLocator.getInstance().getRemoteObject("employeeServiceRO");
        }

        public function load():void {
            var token:AsyncToken = service.getList();
            token.addResponder(responder);
```

```
        }
    }
}
```

在 model 下创建 EmployeesModelLocator（单例模式），用于存储返回的雇员信息

```
package model
{
    import com.adobe.cairngorm.model.IModelLocator;
    import com.adobe.cairngorm.CairngormMessageCodes;
    import com.adobe.cairngorm.CairngormError;

    import mx.collections.ArrayCollection;

    [Bindable]
    public class EmployeesModelLocator implements IModelLocator
    {
        public var employeesList:ArrayCollection;

        private static var _instance:EmployeesModelLocator;

        public function EmployeesModelLocator() {
            if (_instance != null ) {
                throw new
CairngormError(CairngormMessageCodes.SINGLETON_EXCEPTION,
                                            "EmployeesModelLocator");
            }

            _instance = this;
        }

        public static function getInstance():EmployeesModelLocator {
            if (_instance == null) {
                _instance = new EmployeesModelLocator();
            }

            return _instance;
        }
    }
}
```

在 event 下创建 LoadEmployeesEvent 事件

```
package event
{
    import com.adobe.cairngorm.control.CairngormEvent;
```

```
73.
74.     public class LoadEmployeesEvent extends CairngormEvent
75.     {
76.         static public var EVENT_ID:String = "loadEmployees";
77.
78.         public function LoadEmployeesEvent() {
79.             super(EVENT_ID);
80.         }
81.     }
    }
```

82. 在 command 下创建 BaseCommand，作为所有 Commmand 类的基类，以便统一处理 fault 事件。

```
83.package command
84.{
85.     import com.adobe.cairngorm.commands.ICommand;
86.     import com.adobe.cairngorm.control.CairngormEvent;
87.
88.     import mx.rpc.IResponder;
89.     import mx.controls.Alert;
90.
91.     public class BaseCommand implements ICommand, IResponder
92.     {
93.         public function execute(event:CairngormEvent):void {
94.         }
95.
96.         public function result(data:Object):void {
97.         }
98.
99.         public function fault(info:Object):void {
100.                 Alert.show("We are sorry, a system error has
    occurred.
101.                             Please try again later.");
102.            }
103.        }
    }
```

104.在 command 下创建 BaseCommand 的子类 LoadEmployeesCommand。调用 LoadEmployeesDelegate，并把取得的雇员信息保存到 EmployeesModelLocator。

```
105.     package command
106.     {
107.         import com.adobe.cairngorm.control.CairngormEvent;
108.
109.         import business.LoadEmployeesDelegate;
110.         import model.EmployeesModelLocator;
```

```
111.
112.        public class LoadEmployeesCommand extends BaseCommand
113.        {
114.            public override function
    execute(event:CairngormEvent):void {
115.                var delegate:LoadEmployeesDelegate = new
    LoadEmployeesDelegate(this);
116.                delegate.load();
117.            }
118.
119.            public override function result(data:Object):void {
120.                var employeesModelLocator:EmployeesModelLocator =
    EmployeesModelLocator.getInstance();
121.                employeesModelLocator.employeesList = data.result;
122.            }
123.        }
    }
```

124. 到目前你可能比较疑惑：LoadEmployeesEvent 和 LoadEmployeesCommand 是怎样关联上的呢？这就需要控制器了，在 control 下创建 FSController。

```
125.    package controller
126.    {
127.        import com.adobe.cairngorm.control.FrontController;
128.
129.        import event.LoadEmployeesEvent;
130.        import command.LoadEmployeesCommand;
131.
132.        public class FSController extends FrontController
133.        {
134.            public function FSController() {
135.                addCommand(LoadEmployeesEvent.EVENT_ID,
    LoadEmployeesCommand);
136.            }
137.        }
    }
```

138. 是不是觉得都 OK 了？呵呵，别高兴的太早。我们还需要把 Services.mxml 和 FSController.as 引入到主应用文件（即 <s:Application> 标签所在的文件 ）。

把 sampleApp.mxml 文件的 29~30 行替换为以下内容：

```
139.    <rds:Services xmlns:rds="business.*"/>
    <router:FSController xmlns:router="controller.*"/>
```

140. 触发 LoadEmployeesEvent 事件

替换 sampleApp.mxml 的 8~25 行：

```
141.    import event.LoadEmployeesEvent;
142.
```

```
143.     private function init():void {
144.         var loadEmployeesEvent:LoadEmployeesEvent = new
    LoadEmployeesEvent();
145.         loadEmployeesEvent.dispatch();
     }
```

146. 终于到最后一步了：绑定数据源

引入 EmployeesModelLocator

```
import model.EmployeesModelLocator;
```

把以下内容

```
dataProvider="{employeeList}"
```

替换为

```
dataProvider="{EmployeesModelLocator.getInstance().employeesList}"
```

147. 运行

附件：

sampleApp-10.7z

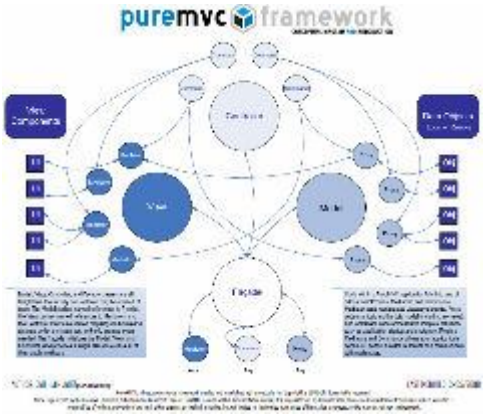
sampleApp(CRUD) Cairngorm-2 + Spring-3.0.2 + iBATIS-2.3.7z

PureMVC 概述

1. PureMVC 是什么？

   PureMVC 是一个定位于设计高性能 RIA 客户端的基于模式的框架。目前已经被移植到多种语言(AS2、AS3、C#、ColdFusion、Haxe、JavaScript、Java、Objective C、PHP、Python、Ruby) 和平台，包括服务器端环境。

2. PureMVC 原理图示



3. PureMVC 的两个版本

   PureMVC 分为标准(Standard)和多核(MultiCore)两个版本。后者目的在于进行模块化编程。[PureMVC - Multicore vs Standard / Singlecore]介绍了两个版本的本质区别。

4. PureMVC 教程

   PureMVC Framework Goals and Benefits

   PureMVC Framework Overview with UML

   PureMVC Implementation Idioms and Best Practices

   PureMVC Implementation Idioms and Best Practices 简体中文(感谢张泽远和 Tamt 的翻译工作)

开始整合

1. 注意事项：

   以下内容基于“Flex4 系列教程之九”中最后形成的 sampleApp 项目。

2. 准备所需组件

   下载 PureMVC(AS3) 多核版，将解压后的 PureMVC_AS3_MultiCore_1_0_5.swc 拷贝到 flex_libs 文件夹。

3. 在 flex_src 下创建以下文件夹

   employees

   employees/controller　　：放置 Command 类

   employees/model　　：放置 Proxy 类

   employees/view　　：放置 Mediator

   employees/view/components：放置视图文件（即 mxml 文件）

4. 在继续之前，还是回顾一下 PureMVC 的原理吧

   记住一点：PureMVC 的通信并不采用 Flash 的 EventDispatcher/Event，而是使用观察者模式以一种松耦合的方式来实现的。

   所以要显示存储在数据库中的职员信息需经过以下过程：

   - View Component 触发一个 Event；
   - Mediator 监听到此 Event，发送通知；
   - 控制器依据通知找到对应的 Command；
   - Command 调用 Proxy（Proxy 又调用 Server 端对象），Proxy 依据执行结果发送相应通知；
   - Mediator 接收到上诉通知，随即把通知中附带的雇员信息赋值给 DataGrid 组件。

5. 不难理解，我们之所以创建 employees 文件夹就是要把雇员信息相关机能放到此文件下。

   出于此目的我们把显示雇员信息的 DataGrid 组件从 sampleApp.mxml 中分离出来，命名为 EmployeesDataGrid，存储于 employees/view/components 下。

```
<?xml version="1.0" encoding="utf-8"?>
<s:VGroup xmlns:fx="http://ns.adobe.com/mxml/2009"
          xmlns:s="library://ns.adobe.com/flex/spark"
          xmlns:mx="library://ns.adobe.com/flex/mx"
          width="400" x="32" y="25">

    <mx:DataGrid id="employeesList" width="400">
        <mx:columns>
            <mx:DataGridColumn headerText="Name"
dataField="name"/>
            <mx:DataGridColumn headerText="Age"
dataField="age"/>
            <mx:DataGridColumn headerText="Email"
dataField="email"/>
        </mx:columns>
    </mx:DataGrid>
</s:VGroup>
```

19. 在 sampleApp 中引入 EmployeesDataGrid 组件

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
```

```
                xmlns:mx="library://ns.adobe.com/flex/mx"
                minWidth="955" minHeight="600"
                xmlns:view="employees.view.components.*">

    <view:EmployeesDataGrid id="employeesDataGrid"/>

</s:Application>
```

29. 在 employees 下创建 ApplicationFacade，作为此应用程序的 Facade

```
package employees
{
    import org.puremvc.as3.multicore.patterns.facade.Facade;
    import employees.controller.*;

    public class ApplicationFacade extends Facade
    {
        public static const STARTUP:String = 'startup';

        public function ApplicationFacade(key:String) {
            super(key);
        }

        public static function
getInstance(key:String):ApplicationFacade {
            if (instanceMap[key] == null)
                instanceMap[key] = new ApplicationFacade(key);

            return instanceMap[key] as ApplicationFacade;
        }

        override protected function initializeController():void {
            super.initializeController();

            registerCommand(STARTUP, StartupCommand);
        }

        public function startup(app:sampleApp):void {
            sendNotification(STARTUP, app);
        }
    }
}
```

[注：]看到上面的 StartupCommand 了吧，我们稍候创建它，该 Command 主要用于注册 Proxy 和 Mediator。

60. 在主应用中初始化 Facade，并调用 startup 方法（应该能理解调用此方法的意图吧？）

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               minWidth="955" minHeight="600"
               xmlns:view="employees.view.components.*"
               initialize="facade.startup(this);">
    <fx:Script>
    <![CDATA[
        import employees.ApplicationFacade;

        public static const NAME:String = 'sampleApp';
        private var facade:ApplicationFacade =
ApplicationFacade.getInstance(NAME);
    ]]>
    </fx:Script>

    <view:EmployeesDataGrid id="employeesDataGrid"/>

</s:Application>
```

79. 是时候创建 StartupCommand 了

```
package employees.controller
{
    import
org.puremvc.as3.multicore.patterns.command.SimpleCommand;
    import org.puremvc.as3.multicore.interfaces.INotification;

    public class StartupCommand extends SimpleCommand
    {
        override public function execute(note:INotification):void {
            // @TODO
        }
    }
}
```

91. 至此 pureMVC 已经整合完毕，是不是很简洁？:）接下来实现雇员信息输出。

92. 首先在 model 下创建 LoadEmployeesProxy，调用远程对象返回雇员信息

```
package employees.model
{
    import org.puremvc.as3.multicore.patterns.proxy.Proxy;
    import mx.rpc.remoting.RemoteObject;
```

```
97.     import mx.rpc.events.ResultEvent;
98.     import mx.rpc.events.FaultEvent;
99.
100.        public class LoadEmployeesProxy extends Proxy {
101.            public static const NAME:String = 'LoadEmployeesProxy';
102.            public static const LOAD_EMPLOYEES_SUCCESS:String =
   'loadEmployeesSuccess';
103.            public static const LOAD_EMPLOYEES_FAILED:String  =
   'loadEmployeesFailed';
104.
105.            private var employeeServiceRO:RemoteObject;
106.
107.            public function LoadEmployeesProxy() {
108.                super(NAME);
109.
110.                employeeServiceRO = new RemoteObject();
111.                employeeServiceRO.destination =
   "employeeServiceDest";
112.
   employeeServiceRO.addEventListener(ResultEvent.RESULT, onResult);
113.
   employeeServiceRO.addEventListener(FaultEvent.FAULT, onFault);
114.            }
115.
116.            public function load():void {
117.                employeeServiceRO.getList();
118.            }
119.
120.            private function onResult(event:ResultEvent):void {
121.                sendNotification(LOAD_EMPLOYEES_SUCCESS,
   event.result);
122.            }
123.
124.            private function onFault(event:FaultEvent):void {
125.                sendNotification(LOAD_EMPLOYEES_FAILED,
   event.fault.faultString);
126.            }
127.        }
   }
```

128.其次在 view 下创建管理 EmployeesDataGrid 的 Mediator — EmployeesDataGridMediator

```
129.    package employees.view
130.    {
```

```
131.        import
    org.puremvc.as3.multicore.patterns.mediator.Mediator;
132.        import org.puremvc.as3.multicore.interfaces.INotification;
133.
134.        import flash.events.Event;
135.        import mx.controls.Alert;
136.
137.        import employees.ApplicationFacade;
138.        import employees.model.LoadEmployeesProxy;
139.        import employees.view.components.EmployeesDataGrid;
140.
141.        public class EmployeesDataGridMediator extends Mediator
142.        {
143.            public static const NAME:String =
    'EmployeesListMediator';
144.
145.            public function
    EmployeesDataGridMediator(viewComponent:EmployeesDataGrid) {
146.                super(NAME, viewComponent);
147.
    employeesDataGrid.addEventListener(EmployeesDataGrid.LOAD_EMPLOYEE
    S,
148.
    onGetEmployees);
149.            }
150.
151.            protected function onGetEmployees(event:Event):void {
152.
    sendNotification(ApplicationFacade.LOAD_EMPLOYEES);
153.            }
154.
155.            override public function
    listNotificationInterests():Array {
156.                return [
157.                    LoadEmployeesProxy.LOAD_EMPLOYEES_SUCCESS,
158.                    LoadEmployeesProxy.LOAD_EMPLOYEES_FAILED
159.                ];
160.            }
161.
162.            override public function
    handleNotification(note:INotification):void {
163.                switch (note.getName()) {
164.                    case
    LoadEmployeesProxy.LOAD_EMPLOYEES_SUCCESS:
```

```
165.
   employeesDataGrid.employeesList.dataProvider = note.getBody();
166.                         break;
167.                     case LoadEmployeesProxy.LOAD_EMPLOYEES_FAILED:
168.                         Alert.show(note.getBody().toString(),
   'Error');
169.                         break;
170.                 }
171.             }
172.
173.             protected function get
   employeesDataGrid():EmployeesDataGrid {
174.                 return viewComponent as EmployeesDataGrid;
175.             }
176.         }
   }
```

177.把上面创建的 Proxy 和 Mediator 注册到 Model 和 View 中

```
178.    package employees.controller
179.    {
180.        import
   org.puremvc.as3.multicore.patterns.command.SimpleCommand;
181.        import org.puremvc.as3.multicore.interfaces.INotification;
182.
183.        import employees.model.LoadEmployeesProxy;
184.        import employees.view.EmployeesDataGridMediator;
185.
186.        public class StartupCommand extends SimpleCommand
187.        {
188.            override public function
   execute(note:INotification):void {
189.                facade.registerProxy(new LoadEmployeesProxy());
190.
191.                var app:sampleApp = note.getBody() as sampleApp;
192.                facade.registerMediator(new
   EmployeesDataGridMediator(app.employeesDataGrid));
193.            }
194.        }
   }
```

[注：]在注册 Mediator 的时候也就确定了它所管理的 Mxml 文件

195.在 controller 中创建 LoadEmployeesCommand，用于调用 LoadEmployeesProxy

```
196.    package employees.controller
197.    {
```

```
198.        import
    org.puremvc.as3.multicore.patterns.command.SimpleCommand;
199.        import org.puremvc.as3.multicore.interfaces.INotification;
200.
201.        import employees.model.LoadEmployeesProxy;
202.
203.        public class LoadEmployeesCommand extends SimpleCommand
204.        {
205.            override public function
    execute(note:INotification):void {
206.                var loadEmployeesProxy:LoadEmployeesProxy =
207.                    facade.retrieveProxy(LoadEmployeesProxy.NAME)
    as LoadEmployeesProxy;
208.                loadEmployeesProxy.load();
209.            }
210.        }
    }
```

211.把 LoadEmployeesCommand 与事件的对应关系追加到 ApplicationFacade 中

```
212.    package employees
213.    {
214.        import org.puremvc.as3.multicore.patterns.facade.Facade;
215.        import employees.controller.*;
216.
217.        public class ApplicationFacade extends Facade
218.        {
219.            public static const STARTUP:String = 'startup';
220.            public static const LOAD_EMPLOYEES:String =
    'loadEmployees';
221.
222.            public function ApplicationFacade(key:String) {
223.                super(key);
224.            }
225.
226.            public static function
    getInstance(key:String):ApplicationFacade {
227.                if (instanceMap[key] == null)
228.                instanceMap[key] = new ApplicationFacade(key);
229.
230.                return instanceMap[key] as ApplicationFacade;
231.            }
232.
233.            override protected function initializeController():void
    {
```

```
                super.initializeController();

                registerCommand(STARTUP, StartupCommand);
                registerCommand(LOAD_EMPLOYEES,
    LOADEmployeesCommand);
            }

            public function startup(app:sampleApp):void {
                sendNotification(STARTUP, app);
            }
        }
    }
```

244.万事俱备，只需要在 EmployeesDataGrid 创建完毕时触发相应事件

```
    <?xml version="1.0" encoding="utf-8"?>
    <s:VGroup xmlns:fx="http://ns.adobe.com/mxml/2009"
              xmlns:s="library://ns.adobe.com/flex/spark"
              xmlns:mx="library://ns.adobe.com/flex/mx"
              width="400" x="32" y="25"
              creationComplete="init();">

        <fx:Metadata>
            [Event('loadEmployees')]
        </fx:Metadata>

        <fx:Script>
        <![CDATA[
            public static const LOAD_EMPLOYEES:String =
    'loadEmployees';

            public function init():void {
                dispatchEvent(new Event(LOAD_EMPLOYEES, true));
            }
        ]]>
        </fx:Script>

        <mx:DataGrid id="employeesList" width="400">
            <mx:columns>
                <mx:DataGridColumn
    headerText="Name"  dataField="name"/>
                <mx:DataGridColumn
    headerText="Age"   dataField="age"/>
                <mx:DataGridColumn headerText="Email"
    dataField="email"/>
```

```
271.             </mx:columns>
272.         </mx:DataGrid>
    </s:VGroup>
```

273.运行试试吧 ：)

附件:

sampleApp-11.7z

sampleApp(CRUD)_pureMVC-MultiCore.1.0.5 + Spring-3.0.2 + iBATIS-2.3.7z