

# JFox 参考指南

V3.1

## 目 录

1	文档说明	4
1.1	内容范围	4
1.2	阅读对象	4
1.3	版权声明	4
1.4	历史记录	4
2	JFox 概述	5
2.1	2.1 功能特性	5
2.2	2.2 JFox 在企业应用中的角色	5
3	JFox 安装与配置指南	6
3.1	运行环境	6
3.2	资源下载	7
3.3	安装 Tomcat	7
3.4	安装 JFox	7
3.5	安装集成版	8
3.6	JFox 目录结构	8
3.7	主程序结构	8
3.8	JFox 模块结构	9
3.9	JFox 配置	9
3.10	web.xml 配置	10
3.11	数据源配置	10
4	第一个 EJB 应用	12
4.1	第一个 Session Bean	12
4.2	第一个 Entity Bean	14
4.3	第一个 Message Driven Bean	19
4.4	容器外运行与容器内运行	21
4.4.1	容器外运行	21
4.4.2	容器内运行	21
5	JFox 体系结构	22
5.1	JFox Java EE 容器	22
5.1.1	JNDI	22
5.1.2	EJB3 Container	23
5.1.2.1	依赖注入	23
5.1.2.2	Interceptors	24
5.1.2.3	Timer Service	25
5.1.2.4	在 EJB 中访问 HttpSession	27
5.1.2.5	JFox 3 EJB 容器的局限性	27
5.1.3	Security	28
5.1.3.1	JAASLoginService	30
5.1.3.2	CallbackHandler	31
5.1.4	Web Service	33
5.1.5	Transaction Manager	36
5.1.6	Message Service	36

5.1.7	JPA Container	39
1.1.1.1	@NamedNativeQuery	39
5.1.7.1	@Entity	40
5.1.7.2	@Column and @MappingColumn	42
5.1.7.3	DataSource 和 Cache	45
5.1.7.4	ID 生成器	47
5.1.7.5	DAO (Data Access Object)	48
5.1.7.6	如何分页	52
5.1.7.7	如何支持多数据库	53
5.1.7.8	了解 JFox JPA 局限	53
5.2	JFox 微内核	54
5.2.1	Component	54
5.2.2	ComponentContext	56
5.2.3	Annotation	56
5.2.4	事件和监听器	61
5.2.5	生命周期回调方法	62
5.2.6	全局占位符	63
5.3	JFox MVC 框架	64
5.3.1	web.xml	64
5.3.2	Action	67
5.3.3	Invocation	75
5.3.4	InvocationContext	81
5.3.5	依赖注入	84
5.3.6	文件上传	84
5.3.7	使用 JAAS 登录	85
5.3.8	访问 Action	86
5.4	JFox Module	87
5.4.1	模块结构	88
5.4.2	模块类路径	88
5.4.3	模块的配置	88
5.4.4	模块的部署	89
6	管理控制台	89
6.1	System Information	89
6.2	JNDI View	90
6.3	EJB Container	91
6.4	JPA Container	91
6.5	Modules	92
6.6	查看 Web Service	93
7	安装 JFox Petstore 模块	93
7.1	环境准备	93
7.2	配置 Petstore 数据源	94
7.3	导入初始化数据	94
7.4	访问 Petstore	95

# 1 文档说明

本文档是 JFox 3.0 开放源码应用服务器参考手册，主要介绍 JFox 3.0 应用服务器的安装、开发、部署、结构。

## 1.1 内容范围

本文档由如下章节组成：

- JFox 概述
- JFox 安装、目录结构、配置
- JFox 体系结构
- JFox 应用开发

## 1.2 阅读对象

本文档面向的主要对象是 JFox 应用服务器的用户、Java EE 开发人员、技术专家、广大师生、Java EE 技术爱好者。如果你有任何疑问，可以访问 JFox 网站 (<http://www.jfox.org.cn>) 获得帮助，或者发邮件至 [support@jfox.org.cn](mailto:support@jfox.org.cn)。

## 1.3 版权声明

Copyright © 2007 JFox Team.

JFox 遵循 LGPL (<http://www.gnu.org/licenses/gpl.txt>) 协议进行分发，复制和修改。

JFox trademarks are trademarks or registered trademarks of JFox Team.

## 1.4 历史记录

日期	内容	作者
2007-05-07	文档结构初始化	Peter Cheng, Young Yang
2007-05-19	完成主要章节	Young Yang, Peter Cheng
2007-05-20	全文 Review, 错误修订	Peter Cheng
2007-05-23	再次 Review, 错误修订	Young Yang
2007-10-08	升级到 3.1 完善@ActionMethod; 增加支持多数据库;	Young Yang

## 2 JFox 概述

JFox 应用被设计为轻量的、稳定的、高性能的 Java EE 应用服务器，从 3.0 开始，JFox 提供了支持模块化的 MVC 框架，以简化 EJB 以及 Web 应用的开发，以满足企业对快速化统一开发平台的迫切要求。

JFox 3 支持标准如下：

- 支持 EJB 规范 3.0
- 支持 JPA 规范 1.0
- 支持 JMS 规范 1.1
- 支持 JNDI 规范 1.2
- 支持 JDBC 规范 3.0
- 支持 JTA 规范 1.1
- 支持 Servlet 规范 2.5
- 支持 JSP 规范 2.1

可以从以下网站获得关于 JFox 的更多信息。

- JFox 官方网站：<http://www.jfox.org.cn>
- JFox 社区网站：<http://www.huihoo.org/jfox>
- JFox 项目网站：<http://code.google.com/p/jfox>

### 2.12.1 功能特性

JFox 3.0 在以前版本的基础上进行较大的重构和改进，在尽可能兼容 EJB 3 规范的同时，保持 JFox 一直以来坚持轻量、简单、高效、实用的目标。

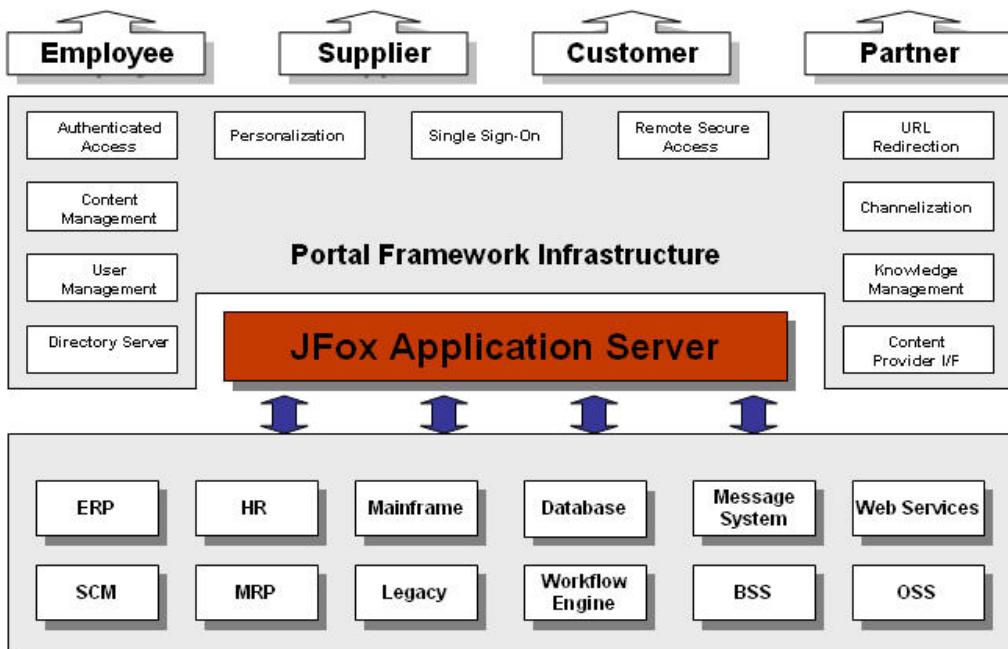
1. 重新设计的 IoC 微内核，融入 OSGi 模块化思想
2. 设计成嵌入式架构，能够和任何 Java Web Server 集成部署
3. 支持 EJB3, JPA 规范，支持容器内和容器外两种方式运行 EJB 和 JPA 组件
4. 支持 EJB 发布成 Web Service
5. 采用 JOTM(<http://jotm.objectweb.org/>) 提供事务处理，支持两阶段提交(2PC)
6. 采用 XAPool(<http://forge.objectweb.org/projects/xapool/>) 提供 XA DataSource，支持智能连接池管理
7. 内置 MVC 框架，实现自动 Form Mapping, Validator, Uploading 等功能，支持 JSP/Velocity/Freemarker 页面引擎，并支持直接在 Action 中注入 EJB
8. 支持多应用模块部署，让中大型应用充分享受模块化开发带来的优势
9. 提供 Manager 管理模块，可以查看和管理各种运行时参数
10. 提供根据 JFox 特色重写的 Petstore 应用模块

### 2.22.2 JFox 在企业应用中的角色

经济的高速发展，信息系统间的孤立成了最大的绊脚石，对于任何一家企业，信息间缺

少关联和通讯都在不时地阻碍企业的发展。解决这一难题的最佳方案就是中间件。JFox 是实现 Java EE 5.0 规范的轻量级应用服务器，作为企业信息化的中间件平台，它的应用范围非常广泛，包括电子政务、电子商务、电信、金融、环保、教育、ERP、CRM、人力资源管理等等与互联网相关的所有行业。JFox 作为企业信息化的核心基础平台，将在信息化进程中扮演重要的角色。如下图：

图 JFox 企业应用系统架构



### 3 JFox 安装与配置指南

请按以下要求和步骤安装 JFox 3.0 应用服务器。

#### 3.1 运行环境

表. 2-2 JFox 运行硬件环境最低配置

CPU	内存	硬盘
PIII 500 相当	128M	1G
SUN Ultra 10 相当	128M	1G

表. 2-3 JFox JDK 支持版本列表

供应商	版本号
Sun Microsystems	5.0 或更高

表. 2-4 JFox 支持操作系统列表

供应商	名称+版本号
-----	--------

Red Hat	RedHat 7.2, 8.0, 9.0, Enterprise Server 3.0
Novell	SuSe 8.0, 9.0
GNU	Debian 3.0
Sun Microsystems	Solaris 6, 8
Microsoft	Windows 2000, 2000Server, 2003, XP

表. 2-5 JFox 支持数据库列表

供应商	名称+版本号
MySQL	MySQL 3.x, 4.x
PostgreSQL	PostgreSQL 7.x, 8.x
Oracle	Oracle 8, 9i, 10g
IBM	DB2 7.1, 7.2, 8.1
Microsoft	MS SQL Server 2000
HSQL	HSQL 1.7.2

## 3.2 资源下载

下载 JDK, <http://java.sun.com>

下载 Tomcat, <http://tomcat.apache.org>

下载 JFox3, <http://code.google.com/p/jfox/downloads/list>

如果需要获得 JFox 3 源码, 请参考 <http://code.google.com/p/jfox/source>。

## 3.3 安装 Tomcat

详细过程请参考 Tomcat 安装指南, <http://tomcat.apache.org>。

实际上, JFox 3 可以部署在任何 Java EE Web 服务器上, 但是没有特殊情况, 我们仍然推荐采用 Tomcat 作为 Web 服务器。

## 3.4 安装 JFox

假设下载的是 JFox standalone 版本, 即 jfox.war (如果下载的是 jfox.war.20070520 这样的文件, 请更名, 去掉日期后缀) 文件, 请将下载的 jfox.war 拷贝到 %TOMCAT\_HOME%/webapps 下, 以便在 Tomcat 启动时加载。JFox 3 应用服务器是设计成嵌入式的, 作为一个标准的 Web Application 部署在 Tomcat 中。

启动 Tomcat, 默认配置下, 使用浏览器访问 <http://localhost:8080/jfox>, 如果看到 JFox 欢迎页面, 那么恭喜你, JFox 安装成功了。

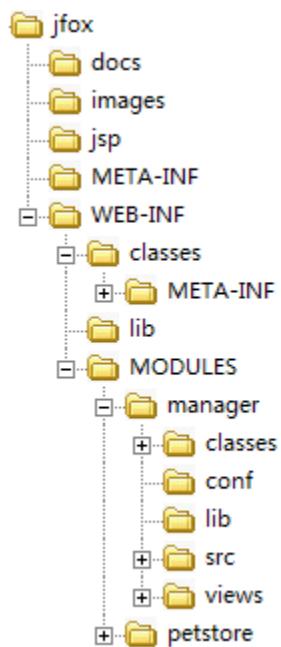
### 3.5 安装集成版

也可以下载已经集成了 Tomcat 的版本，如：“jfox-3.0-with-apache-tomcat-6.0.10.zip”（如果下载的是 jfox.war.20070520 这样的文件，请更名，去掉日期后缀），那么已经包含了 Tomcat 和 JFox，解压之后，直接启动 Tomcat 即可。实际上，这与手动将 JFox 作为 war 部署到 Tomcat 中没有任何区别，JFox 并不需要对 Tomcat 做特殊的配置，集成的版本只是让你的安装过程更简单而已。

### 3.6 JFox 目录结构

JFox 作为 Web Application 部署之后，和一般的 Web Application 目录结构并没有太大的区别，只是再 WEB-INF 下多出了 MODULES 目录，用来部署 jfox 的应用模块。

目录结构如下图：



JFox 的目录分为两部分：

1. JFox 主程序，包含 JFox Java EE 容器和 MVC 框架；
2. 应用模块，包含可以部署在 JFox 3 的各个模块，如：manager, petstore 等。

下面分别介绍。

### 3.7 主程序结构

上图中，除 jfox/WEB-INF/MODULES 目录之外，所有内容都属于 JFox 的主程序。

JFox 主程序结构和标准的 Web Application 没有任何的区别，WEB-INF 外的部分存放了可通过 Web 访问的内容，JFox 的重要文件放在 WEB-INF 目录中。WEB-INF/web.xml 配置了用来启动 JFox 的重要参数，下一节将专门讲解；WEB-INF/lib 下存放了所有 JFox 的 jar 包；WEB-INF/classes 下存放了 JFox 需要的配置文件：

来看一下 WEB-INF/classes 下的配置文件：

目录名	作用	备注
META-INF/persistence.xml	JPA 配置文件	用来配置数据源、以及 Cache 详情参见：JFox 配置章节
jndi.properties	配置 JNDI 属性	一般无需修改配置。 注：JFox 只支持 local JNDI
global.properties	用来配置全局占位符，以用于在 @Constat 注入外部变量	当前仅用来配置了 Transaction Timeout。 如下： jta_transaction_timeout=60 详情可以参见：JFox 内核-全局占位符章节
roles.properties	使用 EJB Security 时，用来配置 Application Role 到 EJB Role 的映射关系	
log4j.properties	Log4J 配置文件	
jotm.properties	JOTM 配置文件	一般无需修改
carol.properties	carol 配置文件	一般无需修改
velocity.properties	Velocity 配置文件	注意：默认采用的是 UTF-8 编码
jaas.conf	JAAS 配置文件	一般无需修改

### 3.8 JFox 模块结构

WEB-INF/MODULES 是 JFox 的一个重要目录，用来部署基于 JFox 模块化规范开发的应用模块，这也是和一般的 Web Application 的不同之处。默认可以看到 MODULES 下有一个 manager 目录或 manager.zip 文件，这是 JFox 内置的 manager 模块，用来展示各种基于 JFox 开发的样例程序，并提供管理控制台，通过 manager 模块，现在就可以了解一下部署在 JFox 之上的应用模块的结构

JFox 对部署之上的模块有一定的结构要求，就如同 Web 服务器对 war 包的结构有要求一样，只有提供了正确的结构，运行时 JFox 才能正确加载。先提示一下，JFox 为每个模块提供独立的类加载器。

关于 JFox 应用模块的更多的介绍将在后面做详细讲解。

### 3.9 JFox 配置

主要需要配置 web.xml 和 persistence.xml 文件。

### 3. 10 web.xml 配置

JFox 3 Web application 的配置文件，一般情况下，无需做修改。

下面的 XML 片段配置了用于启动 JFox 内核的 Listener:

```
1 <listener>
2   <listener-class>org.jfox.mvc.WebContextLoader</listener-class>
3 </listener>
```

下面的 XML 片段配置了用来访问 Web Service 的 Servlet:

```
1 <servlet>
2   <servlet-name>xfire_servlet</servlet-name>
3   <servlet-class>org.jfox.webservice.xfire.JFoxXFireServlet</servlet-class>
4   <load-on-startup>1</load-on-startup>
5 </servlet>
6 <servlet-mapping>
7   <servlet-name>xfire_servlet</servlet-name>
8   <url-pattern>/webservice/*</url-pattern>
9 </servlet-mapping>
```

还有一些其它的配置，均与 JFox MVC 相关，在 JFox MVC 一章中讲述。

### 3. 11 数据源配置

JPA 规范中，通过 persistence.xml 文件来配置存储单元，JFox 的数据源以及 Persistence Cache 也在这里配置，下面展示了 JFox 的 persistence.xml。

```
1 <persistence>
2
3   <persistence-unit name="default">
4     <jta-data-source>java:/DefaultMySqlDS</jta-data-source>
5     <properties>
6       <property name="driver" value="com.mysql.jdbc.Driver"/>
7       <property name="url" value="jdbc:mysql://localhost:3306/test"/>
8       <property name="username" value="root"/>
9       <property name="password" value="root"/>
10      <property name="minSize" value="1"/> <!-- min pool size -->
11      <property name="maxSize" value="200"/> <!-- max pool size -->
12      <property name="lifeTime" value="1800000"/> <!-- 3h, connection max idle time,
in ms -->
13      <property name="sleepTime" value="600000"/> <!-- PoolKeeper sleep time, in ms
-->
14      <property name="deadLockRetryWait" value="2000"/> <!-- retry time if no free
connection, in ms -->
```

```

15      <property name="deadLockMaxWait" value="60000"/> <!-- max wait time if no free
connection, in ms -->
16      <property name="checkLevelObject" value="4"/> <!-- check connection closed -->
17      <property name="cache.algorithm" value="LRU"/> <!-- algorithm for "default"
cache category, LRU, LFU, FIFO-->
18      <property name="cache.ttl" value="600000"/> <!-- ttl for "default", in ms-->
19      <property name="cache.maxIdleTime" value="300000"/> <!-- maxidletime for
"default", in ms-->
20      <property name="cache.maxSize" value="1000"/> <!-- max size for "default" -->
21      <property name="cache.maxMemorySize" value="100000000"/> <!-- max memory size
for "default", in bytes-->
22    </properties>
23  </persistence-unit>
24
25 </persistence>

```

详细的说明如下：

#### <persistence-unit name="default">

指明了 persistence unit 名称，unit 是 JPA 的概念，表示一个存储单元，JPA 的 EntityManager 都关于在存储单元上。使用 @PersistenceContext 注入 EntityManager 时，可以通过 unitName 指定使用的 unit，如果不指明，则会使用默认的 persistence unit，前提是只有一个 unit。

#### <jta-data-source>java:/DefaultMySqlDS</jta-data-source>

指明绑定的数据源 JNDI 的名称，JFox JPA 创建了数据源之后，将使用该名称将其绑定到 JNDI 上。

通过 properties 可以指定各种参数，这些参数是与特定的应用服务器相关的，JFox 使用的 property 说明如下：

名称	意义	备注
driver	数据库驱动	需要将数据库驱动程序拷贝到 WEB-INF/lib 目录中或者 Web Server 的 lib 中
url	数据库连接的 url	
username	用户名	
password	密码	
minSize	最小连接数	维持的最小的数据库连接数目，即使空闲也不会被释放
maxSize	最大连接数	如果实际连接数达到 maxSize，发生新的连接时，将尝试等待有连接释放，直至超时
lifeTime	连接保持时间	如果一个连接已经从创建到当前已经超过 lifeTime 时间，将会被断开以便重连
sleepTime	连接空闲时间	如果一个连接处于空闲，超过 sleepTime 时间之后，将会被释放
deadLockRetryTime	重连的等待时间	如果连接已满，重连之前的等待时间

deadLockMaxWait	最大重试时间	如果连接已满，在重试超过该时间之后，如果仍未连接成功，将抛出连接失败异常
checkObjectLevel	是否检查连接有效	设置是否每次获得连接之后，是否对连接进行有效性检查。如果数据库会出现重启现象，则必须设置有效值，否则只有重启应用服务器才能重新获得有效连接。 4 为总是检查。
cacheAlgorithm	Cache 清除策略的算法	LRU 为最近最少使用； LFU 为最近最常使用； FIFO 为先进先出； 一般配置为 LRU。
cacheTTL	cache 数据的最大生存时间	从创建到当前时间，超过最大生存时间的数据，将会被清除
cacheMaxIdleTime	最大空闲时间	空闲超过该时间的 cache 数据将被清除
cacheMaxSize	cache 的最大尺寸	如果 cache 中的数据的条目已经达到该值，且均有效时，新的数据将不被缓存
cacheMaxMemory	Cache 使用的最大内存数	如果 cache 的数据的体积已经达到最大可用内存数且均有效时，新的数据将不会被缓存

请保证 url、username、password 的设置是正确的，否则将无法成功连接数据库，关于 cache 的更多内容可以参考后面“JPA Container”章节。

## 4 第一个 EJB 应用

接下来，我们看一下如何编写 EJB 组件，旨在让大家对基于 JFox 开发 EJB 有个初步的了解，更多关于 EJB 组件的内容可以参考相关书籍和资料。

### 4.1 第一个 Session Bean

编写 EJB 接口：

```

1 package jfox.test.ejb3.stateless;
2
3 /**
4 * Calculator EJB interface
5 */
6 public interface Calculator {
7
8     int add(int x, int y);
9
10    int subtract(int x, int y);
11 }
```

12

编写 EJB 实现:

```
1 package jfox.test.ejb3.stateless;
2
3 import javax.ejb.Local;
4 import javax.ejb.Remote;
5 import javax.ejb.Stateless;
6
7 /**
8 * Calculator EJB Bean
9 */
10 @Stateless(name = "stateless.CalculatorBean")
11 @Remote
12 @Local
13 public class CalculatorBean implements Calculator {
14
15     public int add(int x, int y) {
16         return x + y;
17     }
18
19     public int subtract(int x, int y) {
20         return x - y;
21     }
22
23 }
24
```

编写用于测试的 EJB Client:

```
1 package jfox.test.ejb3.stateless;
2
3 import java.util.Hashtable;
4 import javax.naming.Context;
5 import javax.naming.InitialContext;
6
7 import org.jfox.framework.Framework;
8 import org.jfox.ejb3.naming.InitialContextFactoryImpl;
9 import org.jfox.ejb3.naming.url.javaURLContextFactory;
10
11 /**
12 * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
13 */
14 public class TestMain {
15
```

```

16  public static void main(String[] args) throws Exception {
17      // start Framework
18      Framework framework = new Framework();
19      framework.start();
20
21      // initialize JNDI
22      Hashtable<String, String> prop = new Hashtable<String, String>();
23      prop.put(Context.INITIAL_CONTEXT_FACTORY,
InitialContextFactoryImpl.class.getName());
24      prop.put(Context.OBJECT_FACTORIES, InitialContextFactoryImpl.class.getName());
25      prop.put(Context.URL_PKG_PREFIXES,
javaURLContextFactory.class.getPackage().getName());
26      prop.put(Context.PROVIDER_URL, "java://localhost");
27      Context context = new InitialContext(prop);
28
29      // lookup calculator then invoke add method
30      Calculator calculator =
(Calculator)context.lookup("stateless.CalculatorBean/remote");
31      int result = calculator.add(99, 1);
32      System.out.println("invoke calculator: 99+1=" + result);
33
34      // stop Framework
35      Thread.sleep(2000);
36      framework.stop();
37  }
38 }
39

```

将 JFox 的 WEB-INF/lib/\*.jar 和 WEB-INF/classes 加入到执行时的 classpath 中，执行 TestMain，将会看到如下输出：

```
invoke calculator: 99+1=100
```

## 4.2 第一个 Entity Bean

在 EJB3 中，Entity Bean 已经独立出来，形成了 JPA 规范，编写 JPA Entity 已经不再像从前那么复杂。首先用@Entity 描述在类上，指明这是一个 JPA Entity 类，然后用@Column 描述在 Field 上，指明对应的数据表的字段。

代码如下：

```

1 package jfox.test.jpa;
2
3 import javax.persistence.Column;
4 import javax.persistence.Entity;
5

```

```
6 /**
7 * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
8 */
9 @Entity
10 public class Address {
11
12     @Column(name="ADR_ID")
13     long id;
14
15     @Column(name="ADR_ACC_ID")
16     long accountId;
17
18     @Column(name="ADR_DESCRIPTION")
19     String description;
20
21     @Column(name="ADR_STREET")
22     String street;
23
24     @Column(name="ADR_CITY")
25     String city;
26
27     @Column(name="ADR_PROVINCE")
28     String province;
29
30     @Column(name="ADR_POSTAL_CODE")
31     String postalCode;
32
33
34     public long getId() {
35         return id;
36     }
37
38     public void setId(long id) {
39         this.id = id;
40     }
41
42     public String getDescription() {
43         return description;
44     }
45
46     public void setDescription(String description) {
47         this.description = description;
48     }
49 }
```

```
50  public String getCity() {
51      return city;
52  }
53
54  public void setCity(String city) {
55      this.city = city;
56  }
57
58  public long getAccountId() {
59      return accountId;
60  }
61
62  public void setAccountId(long accountId) {
63      this.accountId = accountId;
64  }
65
66  public String getPostalCode() {
67      return postalCode;
68  }
69
70  public void setPostalCode(String postalCode) {
71      this.postalCode = postalCode;
72  }
73
74  public String getProvince() {
75      return province;
76  }
77
78  public void setProvince(String province) {
79      this.province = province;
80  }
81
82  public String getStreet() {
83      return street;
84  }
85
86  public void setStreet(String street) {
87      this.street = street;
88  }
89
90  public String toString() {
91      return "Address[" +
92              "id=" + id +
93              ", accountId=" + accountId +
```

```
94     ", city='\" + city + '\" +  
95     ", street='\" + street + '\" +  
96     ", postalCode='\" + postalCode + '\" +  
97     ", province='\" + province + '\" +  
98     ", description='\" + description + '\" +  
99     '}';  
100    }  
101 }  
102
```

编写 Client，下面的代码示范了如何独立执行 JPA，产品开发中，可以使用 @PersistenceContext 直接注入 EntityManager，可以参考 JFox 提供了 perstore 模块。

```
1 package jfox.test.jpa;  
2  
3 import javax.persistence.EntityManager;  
4 import javax.persistence.Persistence;  
5 import javax.persistence.EntityManagerFactory;  
6  
7 /**  
8  * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>  
9 */  
10 public class AddressTestMain {  
11  
12     public static void main(String[] args) {  
13         // initialize EntityManager by persistence unit  
14         EntityManagerFactory emFactory =  
15             Persistence.createEntityManagerFactory("default");  
16         final EntityManager em = emFactory.createEntityManager();  
17  
18         // query  
19         Address address = (Address) em.createNativeQuery("select * from address where ADR_ID  
= $id", Address.class).setParameter("id", 1).getSingleResult();  
20         System.out.println("Address: " + address);  
21  
22         // close EntityManager, EntityManagerFactory  
23         em.close();  
24         emFactory.close();  
25     }  
26 }
```

同样加入 JFox 的 WEB-INF/lib/\*.jar 和 WEB-INF/classes 加入到执行时的 classpath 中，执行 AddressTestMain，成功则会打印出 Address 结果，应该类似如下：

Address: Address{id=1, accountId=1, city='Edmonton', street='434 Edward St.', postalCode='L5G

```
2P9', province='Alberta', description='Fake'}
```

注意：需要建立 ADDRESS 数据库表以及在 META-INF/persistence.xml 中正确的配置数据源才能正确执行成功，下面是创建数据库的 MySQL 脚本，以及 persistence.xml 中 persistence unit 配置。

```
1 CREATE TABLE ACCOUNT (
2     ACC_ID          INTEGER NOT NULL,
3     ACC_FIRST_NAME VARCHAR(32) NOT NULL,
4     ACC_LAST_NAME  VARCHAR(32) NOT NULL,
5     ACC_EMAIL       VARCHAR(32),
6     PRIMARY KEY (ACC_ID)
7 );
8
9 CREATE TABLE ADDRESS (
10    ADR_ID          INTEGER NOT NULL,
11    ADR_ACC_ID      INTEGER NOT NULL,
12    ADR_DESCRIPTION VARCHAR(32) NOT NULL,
13    ADR_STREET      VARCHAR(32) NOT NULL,
14    ADR_CITY        VARCHAR(32) NOT NULL,
15    ADR_PROVINCE   VARCHAR(32) NOT NULL,
16    ADR_POSTAL_CODE VARCHAR(32) NOT NULL,
17    PRIMARY KEY (ADR_ID)
18 );
19
20 INSERT INTO ACCOUNT VALUES(1,'Clinton', 'Begin', 'clinton.begin@ibatis.com');
21 INSERT INTO ACCOUNT VALUES(2,'Jim', 'Smith', 'jim.smith@somewhere.com');
22 INSERT INTO ACCOUNT VALUES(3,'Elizabeth', 'Jones', null);
23 INSERT INTO ACCOUNT VALUES(4,'Bob', 'Jackson', 'bob.jackson@somewhere.com');
24 INSERT INTO ACCOUNT VALUES(5,'Amanda', 'Goodman', null);
25
26 INSERT INTO ADDRESS VALUES(1, 1, 'Fake', '434 Edward St.', 'Edmonton', 'Alberta', 'L5G 2P9');
27 INSERT INTO ADDRESS VALUES(2, 2, 'Fake', '652 John St.', 'Vancouver', 'British Columbia', 'B42 4W2');
28 INSERT INTO ADDRESS VALUES(3, 3, 'Fake', '863 William St.', 'Regina', 'Saskatchewan', 'J9K 4L6');
29 INSERT INTO ADDRESS VALUES(4, 4, 'Fake', '237 Arthur St.', 'Winnipeg', 'Manitoba', 'P4G 2D3');
30 INSERT INTO ADDRESS VALUES(5, 5, 'Fake', '989 Memorial St.', 'Toronto', 'Ontario', 'Q5J 4F4');
31
```

persistence.xml 配置：

```
1 <persistence-unit name="DefaultHsqlDS">
2     <jta-data-source>java:/DefaultHsqlDS</jta-data-source>
3     <properties>
4         <property name="driver" value="com.mysql.jdbc.Driver"/>
5         <property name="url" value="jdbc:mysql://localhost:3306/test"/>
6         <property name="username" value="root"/>
```

```
7      <property name="password" value="root"/>
8  </properties>
9  </persistence-unit>
```

## 4. 3 第一个 Message Driven Bean

编写 MDB(Message Driven Bean)：

```
1 package jfox.test.ejb3.mdb;
2
3 import javax.ejb.ActivationConfigProperty;
4 import javax.ejb.MessageDriven;
5 import javax.jms.MessageListener;
6 import javax.jms.Message;
7
8 /**
9  * @author <a href="mailto:yang_y@sysnet.com.cn">Young Yang</a>
10 */
11 @MessageDriven(activationConfig =
12     {
13         @ActivationConfigProperty(propertyName="destinationType",
14             propertyValue="javax.jms.Queue"),
15         @ActivationConfigProperty(propertyName="destination", propertyValue="testQ")
16     })
17
18 public class QueueMDB implements MessageListener {
19
20     public void onMessage(Message recvMsg) {
21         System.out.println("Received message: " + recvMsg);
22     }
23 }
```

MDB 必须实现 javax. jms. MessageListener 接口，并且使用@MessageDriven 来描述，并申明两个 ActivationConfigProperty，一个为 destinationType 指定地址类型 (javax. jms. Queue 或者 javax. jms. Topic)，另一个为 destination，指定地址名称。

编写 Client:

```
1 package jfox.test.ejb3.mdb;
2
3 import java.util.Hashtable;
4 import javax.naming.Context;
5 import javax.naming.InitialContext;
6 import javax.jms.QueueConnectionFactory;
7 import javax.jms.QueueConnection;
```

```
8 import javax.jms.QueueSession;
9 import javax.jms.Session;
10 import javax.jms.Queue;
11 import javax.jms.QueueSender;
12
13 import org.jfox.framework.Framework;
14 import org.jfox.ejb3.naming.InitialContextFactoryImpl;
15 import org.jfox.ejb3.naming.url.javaURLContextFactory;
16
17 /**
18 * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
19 */
20 public class MDBClient {
21
22     public static void main(String[] args) throws Exception {
23         // start Framework
24         Framework framework = new Framework();
25         framework.start();
26
27         // initialize JNDI
28         Hashtable<String, String> prop = new Hashtable<String, String>();
29         prop.put(Context.INITIAL_CONTEXT_FACTORY,
InitialContextFactoryImpl.class.getName());
30         prop.put(Context.OBJECT_FACTORIES, InitialContextFactoryImpl.class.getName());
31         prop.put(Context.URL_PKG_PREFIXES,
javaURLContextFactory.class.getPackage().getName());
32         prop.put(Context.PROVIDER_URL, "java://localhost");
33         Context context = new InitialContext(prop);
34         QueueConnectionFactory connectionFactory =
(QueueConnectionFactory)context.lookup("defaultcf");
35         QueueConnection connection = connectionFactory.createQueueConnection();
36         QueueSession session = connection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
37         Queue queue = session.createQueue("testQ");
38         QueueSender sender = session.createSender(queue);
39         sender.send(session.createTextMessage("Hello, JMS! " +
System.currentTimeMillis()));
40         session.close();
41         connection.close();
42
43         // stop Framework
44         Thread.sleep(2000);
45         framework.stop();
46     }
```

47 }

48

同样加入 JFox 的 WEB-INF/lib/\*.jar 和 WEB-INF/classes 加入到执行时的 classpath 中，执行 MDBClient，成功则会打印出如下结果：

```
Received message: TextMessage {body=Hello, JMS! 1178634259625}
```

## 4.4 容器外运行与容器内运行

JFox 能完全支持容器外和容器内两种方式运行 EJB 和 JPA 组件，这得益于 EJB3 组件的轻量化设计以及 JFox 容器的轻量和可嵌入式设计。

### 4.4.1 容器外运行

要在容器外运行 EJB，首先需要将 WEB-INF/lib/\*jar 和 WEB-INF/classes 目录加到运行时的 classpath 中。注：以上的例子均为容器外方式运行。

- EJB

要运行 EJB，首先使用如下代码启动 Framework 即可：

```
Framework framework = new Framework();
framework.start();
```

Framework 启动时，会自动加载所有的 EJB 和 JPA Entity，启动之后，即可以通过 JNDI lookup 得到 EJB 并发起调用。

执行完毕之后，可以调用 Framework.stop()方法正常中止 JFox 进程。

- JPA

容器外运行 JPA，则可以通过如下方式创建 EntityManager：

```
EntityManagerFactory emFactory = Persistence.createEntityManagerFactory("default");
final EntityManager em = emFactory.createEntityManager();
```

创建 EntityManagerFactory 的时候，会自动加载 JPA 的 Entity。

然后便可以通过 EntityManager 发起各种持久化调用。

因为这种方式执行 EJB 和 JPA 更加轻量，所以建议在进行单元测试的时候，使用该方式，但是该方式无法提供远程访问能力。

### 4.4.2 容器内运行

要想容器运行，首先需要将 JFox 以 Webapp 方式部署到 Tomcat 容器中，并将你的应用打成 jar 包，可以发布至 WEB-INF/lib 下，JFox 提供了更好的以模块的形式来发布你的应用，关于模块的更多内容可以参考 5.4 JFox Module。

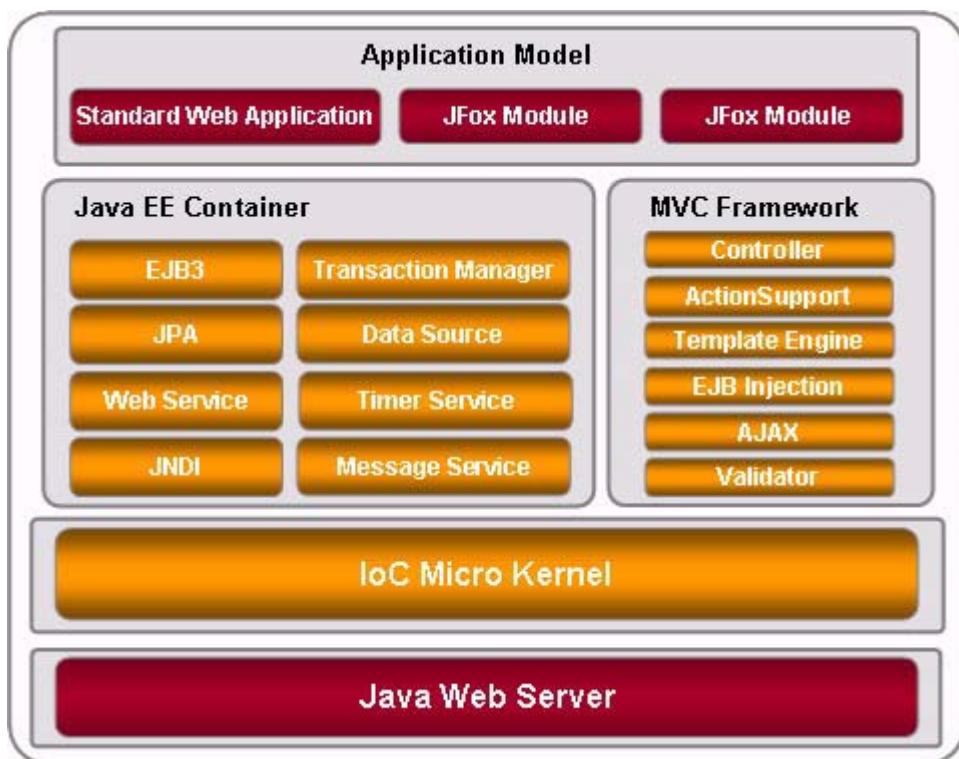
Tomcat 启动时，会自动启动 JFox 的 EJB 容器，并发布你的应用；可以通过查看日志或

者使用管理控制台来检查你的 EJB/JPA 组件是否发布成功。

需要注意的是，即时在该方式下，JFox 也不没有启动任何端口，所以也无法直接提供远程连接服务。从 JSP/Servlet 访问 EJB 时，这不会有问题是；而如果你需要远程访问 EJB，比如：使用独立发布的 Client Application，则需要将 EJB 发布成 Web Service，通过 Web Service 来远程访问 EJB。关于 Web Service 的更多内容，可以参考“JFox 体系结构”中的 Web Service 章节。

## 5 JFox 体系结构

JFox 应用服务器一共包含了 3 个部分，基于 IoC 的微内核、Java EE 容器和 MVC 框架，整体架构如下图所示：



### 5.1 JFox Java EE 容器

JFox 3 提供支持 EJB3 和 JPA 组件的 Java EE 容器，包括服务组件有：JNDI，EJB Container，JTA Transaction Manager，JMS Message Service，JPA Container，DataSource。

#### 5.1.1 JNDI

JNDI (Java Name and Directory Interface) 是 Java EE 应用服务器的基础服务，用来

定位应用服务器中的各种资源和组件。EJB 3 中，大多数时候，总能通过依赖注入来获得资源，JNDI 的使用变得很透明，但是有时候，也需要通过 JNDI 来获取。

要使用 JFox 的 JNDI，需要先设置好 jndi.properties 配置文件，或者初始化 JNDI 上下文时，即 new InitialContext(Properties prop) 的时候传入正确的 Properties。

你可以在 WEB-INF/classes 下找到 jndi.properties，它的配置如下：

```
1 java.naming.factory.initial=org.jfox.ejb3.naming.InitialContextFactoryImpl  
2 java.naming.factory.object=org.jfox.ejb3.naming.InitialContextFactoryImpl  
3 java.naming.provider.url=java://localhost
```

也可以使用 JFox 提供 JNDIContextHelper.getInitialContext() 来直接获得 InitialContext，JNDIContextHelper 全名为：org.jfox.ejb3.naming.JNDIContextHelper，在构造 InitialContext 时，已经传入了正确的参数。

部署的 EJB 的 JNDI 名称由两部分组成：

1. @Stateless/@Stateful 标注时指定的 name
2. 如果是@Remote，加上 “/remote”，如果是@Local，则加上 “/local”

在 EJB3 中，需要注意相对 EJB2 JNDI 的变化，EJB3 直接将 EJB 的存根绑定在了 JNDI 中，而不是 EJB2 的 EJBHome 的存根，这使得之前传统的 ServiceLocator 模式也可能需要改进，尤其是对于 Stateful Session Bean，不能再使用 ServiceLocator 中缓存 JNDI lookup 得到的存根，如果这样的话，所有的会话都将调用同一个 Stateful Session Bean 的实例。

另外，需要格外注意的是，JFox 的 JNDI 不具备远程访问能力，仅支持 Local 访问，也就是说，你不能在除 JFox 的 JVM 之外使用 JNDI，这样做，一方面是为了安全，另一方面大多数应用都在使用 Local 方式来访问 EJB，还有就是更推荐采用 Web Service 来实现远程访问。所以，如果你需要远程访问 EJB，那么更好的做法是将你的 EJB 发布成 Web Service，你的 Client 通过 Web Service 来实现远程访问，后面有例子会告诉你通过 Web Service 来访问你的 EJB 服务会是多么的简单。

## 5.1.2 EJB3 Container

JFox 的 EJB3 容器支持 Stateless、Stateful SessionBean 和 Message Driven Bean，为了保持容器的简单性，并不兼容 EJB2 容器，如果需要 EJB2 容器，请选择 JFox 2.x 的版本，但是更建议你升级你的应用至符合 EJB 3 规范。

### 5.1.2.1 依赖注入

支持依赖注入是 EJB3 规范的重要改进，JFox EJB3 容器使用 JDK 5 的 Annotation 来标注依赖，支持使用 @EJB, @Resource 等在 EJB 中注入依赖的资源。可以参考 JFox 测试用例 jfox.test.ejb3.injection 来了解依赖注入的使用，更多的关于依赖注入的知识请参考 EJB3 规范。

下面是一些使用依赖注入的示例代码：

1. 注入 EJB

```
1  @EJB(beanName = "injection.CalculatorBean")
2  private Calculator calculator;
```

2. 注入 SessionContext

```
1  @Resource
2  SessionContext sessionContext;
```

3. 注入 TimerService

```
1  @Resource
2  TimerService timerService;
```

4. 注入 EntityManager

```
1  @PersistenceContext(unitName = "JPetstoreMysqlDS")
2  EntityManager em;
```

## 5.1.2.2 Interceptors

JFox 3 支持 EJB 3 的 Interceptors，可以使用@AroundInvoke 来标注 EJB 方法为拦截方法，或者使用@Interceptors 指定外部的拦截类。可以参考 jfox 测试用例 jfox.test.ejb3.interceptor 来了解拦截器的使用。

下面是示例代码：

```
1 package jfox.test.ejb3.interceptor;
2
3 import javax.annotation.Resource;
4 import javax.annotation.PostConstruct;
5 import javax.annotation.PreDestroy;
6 import javax.ejb.Local;
7 import javax.ejb.Remote;
8 import javax.ejb.SessionContext;
9 import javax.ejb.Stateless;
10 import javax.interceptor.AroundInvoke;
11 import javax.interceptor.InvocationContext;
12 import javax.interceptor.Interceptors;
13
14 @Stateless(name = "interceptor.CalculatorBean")
15 @Remote
16 @Local
17 @Interceptors({OuterClassInterceptor.class})
```

```

18 public class CalculatorBean extends SuperCalculatorBean implements CalculatorRemote,
CalculatorLocal {
19
20     @Resource
21     SessionContext sessionContext;
22
23     public int add(int x, int y) {
24         return x + y;
25     }
26
27     @Interceptors({OuterMethodInterceptor.class})
28     public int subtract(int x, int y) {
29         return x - y;
30     }
31
32     @AroundInvoke
33     public Object aroundInvoke(InvocationContext invocationContext) throws Exception {
34         System.out.println("AroundInvoke: " + this.getClass().getName() + ".aroundInvoke,
method: " + invocationContext.getMethod().getName());
35         return invocationContext.proceed();
36     }
37
38 }
```

### 5.1.2.3 Timer Service

Timer Service 为 EJB 提供延时或定时执行某个方法的能力。EJB 3 中，可以通过@Resource 注入 Timer Service，并通过实现 TimedObject 接口来实现 Timer Service 执行的方法，或者使用@Timedout 来标注 Time Service 执行的方法。

下面的例子展示了如何使用 Timer Service，完整代码可以参考 JFox TimerService 测试用例，包名：jfox.test.ejb3.timer。

EJB 接口，定义用来提交定时任务的方法：

```

1 package jfox.test.ejb3.timer;
2
3 public interface ExampleTimer {
4     /**
5      * EJB 方法，用来提交定时任务
6      */
7     void scheduleTimer(long milliseconds);
8 }
```

EJB Bean 实现:

```
1 package jfox.test.ejb3.timer;
2
3 import java.util.Date;
4 import javax.ejb.Remote;
5 import javax.ejb.SessionContext;
6 import javax.ejb.Stateless;
7 import javax.ejb.Timeout;
8 import javax.ejb.Timer;
9 import javax.ejb.TimedObject;
10 import javax.ejb.TimerService;
11 import javax.annotation.Resource;
12
13 @Stateless(name = "timer.ExampleTimerBean")
14 @Remote
15 public class ExampleTimerBean implements ExampleTimer, TimedObject {
16
17     @Resource
18     private SessionContext ctx;
19
20     @Resource
21     TimerService timerService;
22
23     /**
24      * 使用两种方式分别提交一个定时任务
25      */
26     public void scheduleTimer(long milliseconds) {
27         ctx.getTimerService().createTimer(new Date(new Date().getTime() + milliseconds),
28                                         "Hello World");
29         timerService.createTimer(new Date(new Date().getTime() + milliseconds), "Hello
30                                         World2");
31     }
32
33     /**
34      * 使用 @Timeout 标注
35      */
36     @Timeout
37     public void timeoutHandler(Timer timer) {
38         System.out.println("-----");
39         System.out.println("* Received Timer event: " + timer.getInfo());
40         System.out.println("-----");
41         timer.cancel();
42     }
43 }
```

```

42  /**
43   * 实现 TimedObject 定义的方法
44   */
45  public void ejbTimeout(Timer timer) {
46      System.out.println("-----");
47      System.out.println("* Received interface Timer event : " + timer.getInfo());
48      System.out.println("-----");
49      timer.cancel();
50  }
51 }
52

```

更多关于 TimerService 的信息，可以参考 EJB3 规范或者访问下面的链接：  
[http://www.javabeat.net/javabeat/ejb3/articles/timer\\_services\\_api\\_in\\_ejb\\_3\\_0\\_1.php](http://www.javabeat.net/javabeat/ejb3/articles/timer_services_api_in_ejb_3_0_1.php)

### 5.1.2.4 在 EJB 中访问 HttpSession

EJB 方法中，有时也会用到 HttpSession，比如：检查当前登录用户。

JFox 扩展了 EJBContext，以便可以访问 HttpSession。在 org.jfoxejb3.ExtendEJBContext 中，可以看到该方法：

```

1  /**
2   * 获取 Http Session 中的数据，需要保证 Web Server 和 EJB Container 在同一虚拟机中
3   * @param attribute session key
4   */
5  public Object getSessionAttribute(String attribute) {
6      return EJBInvocation.current().getSessionAttribute(attribute);
7 }

```

在 EJB 中使用 @Resource 注入 EJBContext 之后，可以强制造型成 org.jfoxejb3.ExtendEJBContext，然后调用 getSessionAttribute 方法访问 HttpSession。

### 5.1.2.5 JFox 3 EJB 容器的局限性

了解 JFox 3 EJB 容器的局限性，有助于你更好的和正确使用 JFox。JFox 容器设计时，主要考虑普遍的 Web 容器和 EJB 容器署在同一个 JVM 中的情况，所以容器本身并没有提供远程访问服务。如果你有计划将 Web 和 EJB 部署分离部署在不同的 JVM 中，那么只能使用 Web Service 来调用 EJB 组件。

## 5. 1. 3 Security

EJB Security 为 EJB 提供基于角色的方法级安全，并可以和应用程序的安全系统集成，认证之后，认证信息可以在 Client 和 EJB 容器之间传播。

JFox 对 JAAS 进行了封装，提供了基于 JAAS 的登录服务，并实现了应用角色到 EJB 角色之间的映射，以及在 Client 和 EJB 容器之间传播 Seciruty Context。

为了使用 JFox EJB 安全，需要使用 JFox 提供的 LoginService 进行登录，并且需要配置好 roles.properties 配置文件以定义应用角色到 EJB 角色的映射，然后在 EJB 中使用 javax.annotation.security 提供的 Annotation 对 EJB 方法进行标注以进行安全控制。

- **EJB Security Annotation**

javax.annotation.security 提供了多种 Annotation 用来对 EJB 进行标注。下面的 EJB 展示了如何使用 EJB Security Annotation 来标注 EJB 的安全。

```
1 package jfox.test.ejb3.security;
2
3 import java.security.Principal;
4 import javax.annotation.security.DenyAll;
5 import javax.annotation.security.PermitAll;
6 import javax.annotation.security.RolesAllowed;
7 import javax.annotation.security.RunAs;
8 import javax.annotation.Resource;
9 import javax.ejb.Local;
10 import javax.ejb.Remote;
11 import javax.ejb.Stateless;
12 import javax.ejb.SessionContext;
13
14 @Stateless(name = "security.CalculatorBean")
15 @Remote
16 @Local
17 @RunAs("role")
18 public class CalculatorBean implements CalculatorRemote, CalculatorLocal {
19
20     @Resource
21     SessionContext ejbContext;
22
23     @PermitAll
24     public int add(int x, int y) {
25         return x + y;
26     }
27
28     @DenyAll
```

```

29     public int subtract(int x, int y) {
30         return x - y;
31     }
32
33     @RolesAllowed({"role1,role2"})
34     public int plus(int x, int y) {
35         return x*y;
36     }
37
38     public double devide(int x, int y) {
39         Principal principal = ejbContext.getCallerPrincipal();
40         System.out.println("devide caller principal: " + principal);
41         return x/y;
42     }
43
44 }
45

```

@RunAs 标注在 Class 上，指定使用特定的角色来执行该 Bean，一般来说，仅用于测试阶段；其它的 Annotation 标注在方法上，@PermitAll 表示所有角色可以访问，@DenyAll 表明所有角色都不能访问，@RoleAllowed 表示只有参数中的角色可以访问。注意这里说的角色均指 EJB 角色。

EJB Security Annotation 详细的内容可以参考 EJB 规范。

### ● Role 映射

为了建立 EJB 角色和应用角色之间的映射关系，需要编辑 roles.properties 文件。

roles.properties 文件格式如下：

```

1 #define application role link to ejb role
2 #application role = ejb role
3 account_manager=manager
4 technology_manager=manager

```

key 为应用角色(Application Role)，value 为 EJB 角色(EJB Role)，在 EJB Security 中，将以 EJB 角色来进行判断，如下面的代码，限定 plus 方法只能被 EJB 角色 role1 和 role2 访问：

```

1     @RolesAllowed({"role1,role2"})
2     public int plus(int x, int y) {
3         return x*y;
4     }

```

而对于没有配置对应 EJB 角色的应用角色，JFox 将直接使用应用角色作为 EJB 角色。

### 5.1.3.1 JAASLoginService

JFox 3 提供了 JAASLoginService 提供登录服务。

下面是 JAASLoginService 接口的定义：

```

1 package org.jfox.ejb3.security;
2
3 import javax.security.auth.callback.CallbackHandler;
4 import javax.servlet.http.HttpServletRequest;
5
6 import org.jfox.mvc.SessionContext;
7 import org.jfox.framework.component.Component;
8 import org.jfox.framework.annotation.Exported;
9
10 /**
11  * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
12 */
13 @Exported
14 public interface JAASLoginService extends Component {
15
16     static ThreadLocal<JAASLoginRequestCallback> loginRequestThreadLocal = new
ThreadLocal<JAASLoginRequestCallback>();
17     static ThreadLocal<JAASLoginResponseCallback> loginResponseThreadLocal = new
ThreadLocal<JAASLoginResponseCallback>();
18
19     /**
20      * 登录接口方法，如果在 Servlet 中直接调用，则可以使用该方法
21      *
22      * @param request http servlet request
23      * @param callbackHandler 进行登录的 callback，调用时由 client 指定
24      * @param params 登录时提供的参数，一般是用户名和密码
25      * @return 登录成功之后的对象，比如：Account 对象，取决于 callbackHandler 调用
JAASLoginResponseCallback.setCallbackObject
26      * @throws Exception 登录失败时抛出异常
27      */
28     public Object login(HttpServletRequest request, CallbackHandler callbackHandler,
String... params) throws Exception;
29
30     /**
31      * 登录接口方法，如果在 Action 中调用，则可以使用该方法，因为能够得到 SessionContext
32      *
33      * @param sessionContext session context
34      * @param callbackHandler 进行登录的 callback，调用时由 client 指定
35      * @param params 登录时提供的参数，一般是用户名和密码

```

```
36     * @return 登录成功之后的对象，比如：Account 对象，取决于 callbackHandler 调用
JAASLoginResponseCallback.setCallbackObject
37     * @throws Exception 登录失败时抛出异常
38     */
39     public Object login(SessionContext sessionContext, CallbackHandler callbackHandler,
String... params) throws Exception;
40 }
41
```

JAASLoginService 主要定义了两个方法，这两个方法之后第一参数不同，第一个参数用来得到提供 Session Context，JFox 使用 Session 传播 Security Context。

JAASLoginService 有一个实现类 JAASLoginServiceImpl，它一个 JFox 内核服务组件，为了使用 JAASLoginService，可以使用 JFox 微内核定义的 @Inject 注入，或者通过 ComponentContext.getComponentsByInterface 得到，关于 JFox 内核的内容，详细的描述见后面的“JFox 微内核”章节。

JFox Petstore 的 AccountAction 有以下代码，用来注入 LoginService：

```
1  @Inject
2  JAASLoginService loginService;
```

### 5.1.3.2 CallbackHandler

为了使用 JAAS 登录，需要提供 javax.security.auth.callback.CallbackHandler 回调对象，CallbackHandler 是 JAAS 定义的接口，在登录过程中由 JAAS 调用，用来实现用户认证的过程。接口定义如下：

```
1 package javax.security.auth.callback;
2 public interface CallbackHandler {
3     void handle(Callback[] callbacks)
4         throws java.io.IOException, UnsupportedCallbackException;
5 }
```

CallbackHandler.handle 方法的参数为 Callback[] 数组，由 JAAS 实现提供，JFox 提供的是一个有两个元素的数组，它们的类型分别是：

- org.jfox.ejb3.security.JAASLoginRequestCallback
- org.jfox.ejb3.security.JAASLoginResponseCallback

JAASLoginRequestCallback 携带有登录的参数数组，数组的大小这取决于你调用 JAASLoginService.login 时提供的可变参数 (String... params) 的大小，一般来说，0 号元素为 username，1 号元素为 password；

JAASLoginResponseCallback 用来携带登录完成之后需要返回的值，setCallbackObject 设置返回的对象，setPrincipalName 设置用户名，setRole 设置用户的角色，这些值将用于生成用户的 Security Context。

下面是一个 JFox Petstore 中 AccountAction 中实现的 handle 方法:

```

1  /**
2   * JAAS CallbackHandler method
3   */
4  public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
5      JAASLoginRequestCallback requestCallback =
(JAASLoginRequestCallback) callbacks[0];
6      JAASLoginResponseCallback responseCallback =
(JAASLoginResponseCallback) callbacks[1];
7
8      // first parameter is username
9      String username = requestCallback.getParams().get(0);
10     // second parameter is password
11     String password = requestCallback.getParams().get(1);
12
13     Account account = accountB0.getAccount(username, password);
14
15     // set callback object, will return by LoginService.login
16     responseCallback.setCallbackObject(account);
17     // set principal name
18     responseCallback.setPrincipalName(username);
19     // set role
20     responseCallback.setRole(username);
21 }
```

AccountAction. doPostSignOn 方法调用 JAASLoginService. login 登录，将 this 作为 CallbackHandler:

```
1 Account account = (Account)loginService.login(invocationContext.getSessionContext(), this,
invocation.getUsername(), invocation.getPassword())
```

让 Action 类实现 CallbackHandler 接口，可以方便的在 handle 方法中访问登录过程需要的资源。

### ● Security Context 的传播

用户登录成功之后，系统将为其生成 Security Context，在 Client 和 EJB 容器交互的过程，Security Context 需要在 EJB 的 Client 和 EJB 容器之间不断的进行传播，以便容器总是能够得到当前调用者的 Security 信息，以做出安全判断。

因为 JFox 3 将 EJB 容器嵌入在 Web 容器中设计，所以 Security Context 采用了 HttpSession 来进行传播，这是最方便的一种传播方式，用户登录时，LoginService 会将用户的 Security Context 存放在 Session 中，EJB 容器通过访问当前用户的 Session 就可获得其 Security 信息，在用户退出时，只需要销毁 HttpSession 即可以销毁用户的 Security

Context。这样的做法在 Web 容器和 EJB 容器运行在相同的 JVM 中时，能够非常好的工作，而如果 Web 容器和 EJB 容器运行在不同的 JVM 中，则需要其它的传播机制，其它的 Security Context 传播方式可能在后续版本中提供。

HttpSession 是在 JAASLoginService. login 的方法中，使用 HttpServletRequest 或者 SessionContext 提供的。

## 5. 1. 4 Web Service

EJB 3 让 Web Service 的使用变得非常简单。JFox 3 集成了 Xfire(<http://xfire.codehaus.org/>) 来支持 Web Service，要在 JFox 中使用 Web Service 非常简单。

要使用 Web Service，首先需要检查 web.xml 的配置，如下配置了用来接受 Web Service 请求的 Servlet：

```
1 <servlet>
2   <servlet-name>xfire_servlet</servlet-name>
3   <servlet-class>org.jfox.webservice.xfire.JFoxXFireServlet</servlet-class>
4   <load-on-startup>1</load-on-startup>
5 </servlet>
6 <servlet-mapping>
7   <servlet-name>xfire_servlet</servlet-name>
8   <url-pattern>/webservice/*</url-pattern>
9 </servlet-mapping>
```

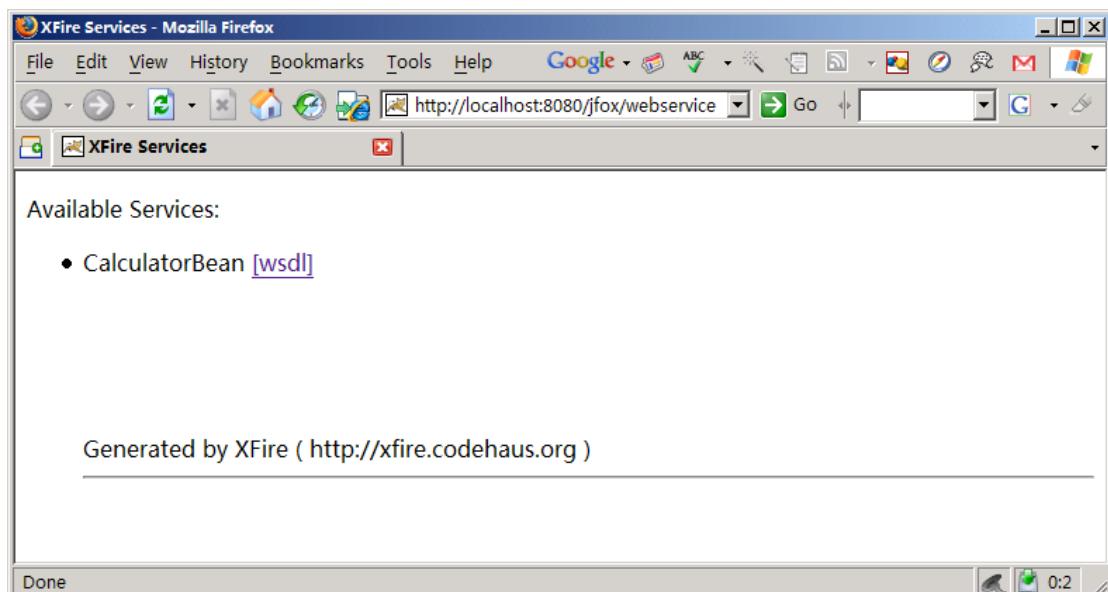
下面的例子来源于 JFox 测试用例，包名： jfox. test. ejb3. webservice。

给 EJB Bean 的加上@WebService 标注，以将 EJB 发布成 Web Service，endpointInterface 指明发布的接口名，不指定则为 EJB Bean 实现的所有接口，serviceName 指明 Web Service 的名称，默認為 Bean 的类名简称（即：CalculatorBean）。

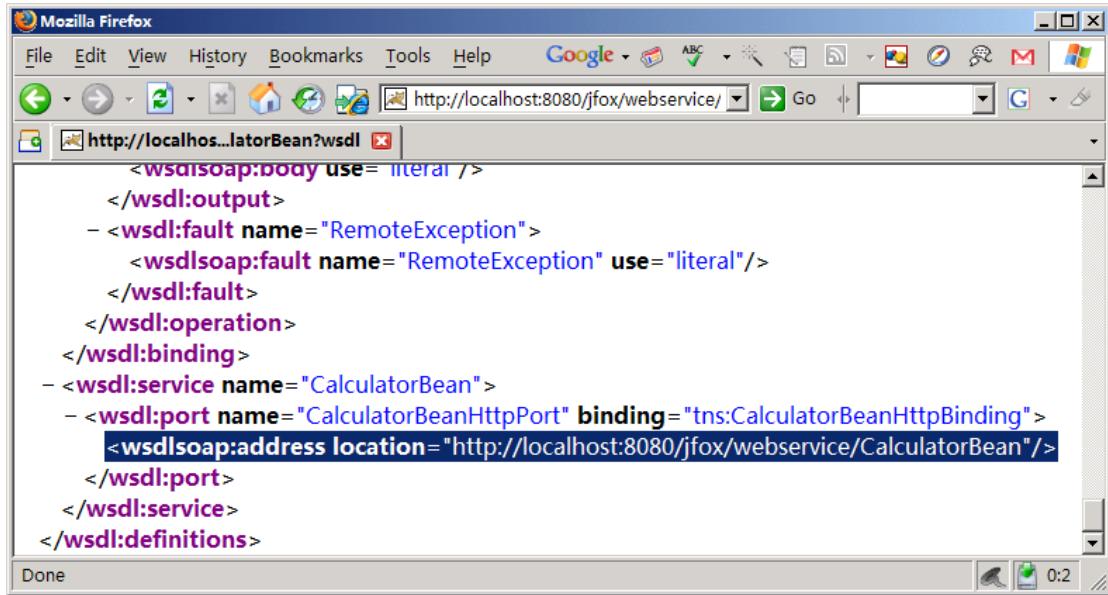
```
1 package jfox.test.ejb3.webservice;
2
3 import javax.ejb.Local;
4 import javax.ejb.Remote;
5 import javax.ejb.Stateless;
6 import javax.jws.WebService;
7
8 import jfox.test.ejb3.stateless.Calculator;
9
10 @Stateless(name = "webservice.CalculatorBean")
11 @Remote
12 @Local
13 @WebService(endpointInterface = "jfox.test.ejb3.webservice.Calculator", serviceName =
"CalculatorBean")
```

```
14 public class CalculatorBean implements Calculator {  
15  
16     public int add(int x, int y) {  
17         return x + y;  
18     }  
19  
20     public int subtract(int x, int y) {  
21         return x - y;  
22     }  
23  
24 }  
25
```

系统启动之后，访问 <http://localhost:8080/jfox/webservice> 即可以看到发布成功的 Web Service 列表，如下图：



点击 wsdl，就可以查看详细内容，其中有 soap 地址：



要调用 Web Service，可以使用 `WebServiceHelper.lookupWS`，第一个参数是你的 Web Service 发布的 endpoint URL，第二个参数为 Web Service 发布的接口，代码如下：

```
1 package jfox.test.ejb3.webservice;
2
3 import org.jfox.webservice.WebServiceHelper;
4
5 /**
6  * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
7 */
8 public class WebServiceClient {
9
10    public static void main(String[] args) throws Exception{
11
12        Calculator example =
13        WebServiceHelper.lookupWS("http://localhost:8080/jfox/webservice/CalculatorBean",
14        Calculator.class);
15
16        System.out.println("Web Service invoke Calculator.add(1,1): " + example.add(1, 1));
17        System.out.println("Web Service invoke Calculator.subtract(2,1): " +
example.subtract(2, 1));
18    }
19 }
```

输出如下：

```
Web Service invoke Calculator.add(1,1): 2
Web Service invoke Calculator.subtract(2,1): 1
```

WebServiceHelper 提供了基于 Java 语言的而且已经拥有 Web Service 接口的调用方法，如果需要使用其它的语言或者无法获得接口的情况下，则需要通过 WSDL 动态生成 Web

Service Client 需要的文件，然后按规范进行调用。

使用 JFox 提供的 Web Service 要注意一下几点：

1. EJB 规范要求只有 Stateless Bean 可以发布成 Web service；
2. JFox 暂时只支持使用 @WebService 指明接口来发布 Web Service，还不支持 @WebMethod 指定方法的形式发布 Web Service。

## 5. 1. 5 Transaction Manager

JFox 3 使用 JOTM (<http://jotm.objectweb.org>) 作为 Transaction Manager，为 EJB 和 JPA 提供事务处理服务。JOTM 这是一个久经考验的 JTA 的实现，JFox 集成了 JOTM 之后，又对 Transaction Manager 进行了大量的测试，来保证 Transaction 正确性以及高效性。JFox 支持所有 EJB 事务属性，EJB 3 的事务属性由 javax.ejb.TransactionAttributeType 定义，共包括：MANDATORY, NEVER, NOT\_SUPPORTED, REQUIRED, REQUIRES\_NEW, SUPPORTS。EJB 方法的默认的事务属性为：REQUIRED。

下面的代码展示了对 ejb 方法进行事务属性的声明：

```
1  @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
2  public void ejbMethod() {
3      System.out.println("ejbMethod");
4  }
5
```

虽然可以 EJB 支持 CMT (Container Managed Transaction) 和 BMT (Bean Managed Transaction)，但是建议你总是使用容器管理的事务。

你可以通过修改 global.properties 的 jta\_transaction\_timeout=60 来调整默认的 Transaction 超时时间。

关于 EJB3 的事务的更多内容请参考 EJB3 规范。

## 5. 1. 6 Message Service

JFox 实现了一个轻量的符合 JMS 规范的 MessageService。它非常轻量，仅用来实现 Message Driven Bean，所以它只实现了往本地分发消息的能力。

实现一个 Message Driven Bean 非常简单，下面的代码展示实现一个基于 Topic 的 Message Driven Bean。

```
1 import javax.ejb.MessageDriven;
2 import javax.ejb.ActivationConfigProperty;
3 import javax.jms.MessageListener;
4 import javax.jms.Message;
```

```
5
6 /**
7  * @author <a href="mailto:yang_y@sysnet.com.cn">Young Yang</a>
8 */
9 @MessageDriven(activationConfig =
10 {
11     @ActivationConfigProperty(propertyName="destinationType",
propertyValue="javax.jms.Topic"),
12     @ActivationConfigProperty(propertyName="destination", propertyValue="testT")
13 })
14 public class TopicMDB1 implements MessageListener {
15
16     public void onMessage(Message recvMsg) {
17         System.out.println("Received message: " + recvMsg);
18     }
19
20 }
```

注意：MDB 必须实现 javax.jms.MessageListener 接口，并且使用@MessageDriven 来描述，并申明两个 ActivationConfigProperty，一个为 destinationType 指定地址类型（javax.jms.Queue 或者 javax.jms.Topic），另一个为 destination，指定地址名称。

一般来说，应该使用一个 Stateless Session 来完成发送消息的任务，如果 Client 与应用服务器不在同一个 JVM 中，则可以将 Stateless Session Bean 发布成 Web Service，Client 通过 Web Service 来实现远程向 Message Service 发送消息。

下面的代码展示了通过 Stateless Session Bean 向 Message Service 发布消息：

```
1 package jfox.test.ejb3.mdb;
2
3 import javax.annotation.Resource;
4 import javax.ejb.EJBException;
5 import javax.ejb.Remote;
6 import javax.ejb.Stateless;
7 import javax.jms.Message;
8 import javax.jms.QueueConnection;
9 import javax.jms.QueueConnectionFactory;
10 import javax.jms.QueueSender;
11 import javax.jms.QueueSession;
12 import javax.jms.Session;
13 import javax.jms.TopicConnection;
14 import javax.jms.TopicConnectionFactory;
15 import javax.jms.TopicPublisher;
16 import javax.jms.TopicSession;
17
18 import org.jfox.ejb3.naming.JNDIContextHelper;
```

```
19
20 /**
21 * @author <a href="mailto:yang_y@sysnet.com.cn">Young Yang</a>
22 */
23 @Stateless
24 @Remote
25 public class MessageSenderBean implements MessageSender {
26
27     @Resource
28     QueueConnectionFactory queueConnectionFactory;
29
30     public void sendQueueMessage(Message message) {
31         try {
32             // use injected jms connection factory
33             QueueConnection qc = queueConnectionFactory.createQueueConnection();
34             QueueSession qs = qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
35             QueueSender sender = qs.createSender(qs.createQueue("testQ"));
36             sender.send(qs.createTextMessage("Hello, Queue MDB!"));
37         }
38         catch (Exception e) {
39             throw new EJBException(e);
40         }
41     }
42
43     public void sendTopicMessage(Message message) {
44         try {
45             // lookup jms connection factory by jndi
46             TopicConnectionFactory tcf =
47             (TopicConnectionFactory) JNDIContextHelper.lookup("defaultcf");
48             TopicConnection tc = tcf.createTopicConnection();
49             TopicSession ts = tc.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
50             TopicPublisher tp = ts.createPublisher(ts.createTopic("testT"));
51             tp.send(ts.createTextMessage("Hello, Topic MDB!"));
52         }
53         catch (Exception e) {
54             throw new EJBException(e);
55         }
56     }
57 }
```

关于 JMS 以及 Message Driven Bean 的更多内容可以参考相关规范。

## 5.1.7 JPA Container

JPA 是 Java EE 体系中另外一个重要的容器，在 EJB3 中，EntityBean 已经独立称 JPA 规范，JPA 参考了 Hibernate、JDO 等既有的 O/R 产品和规范，形成了统一的 Java 持久化 API。

JFox JPA 容器在兼容 JPA 接口的前提下，提供了一个可靠的、高效的、易于掌握且可控的持久化方案。

与纯 O/R Mapping 的框架不同，JFox JPA 容器采用了 SQLTemplate 设计思路来实现 JPA 的实现，SQLTemplate 最先由 iBatis(<http://ibatis.apache.org>) 提出，SQLTemplate 既大大简化了 JDBC 的开发，又很好地保留了 SQL 的特性，同时提供一定的 O/R 映射的能力。由于基于 SQLTemplate 更简单和容易掌握，且拥有更好的可控性，便于后期进行查错和调优，所以在一些大型的产品或者对数据库性能敏感的产品中，更受欢迎。

要掌握 JFox JPA，需要掌握 Named Query、字段映射、数据源、缓存等内容。

### 1.1.1.1 @NamedNativeQuery

JFox 3 使用 JPA 的 @NamedNativeQueries 和 @NamedNativeQuery 命名 SQLTemplate，SQLTemplate 采用的是 Velocity(<http://velocity.apache.org>) 模板表达式语言。@NamedNativeQuery 建议描述对 DAO Session Bean 上，但也不是必须的，也可以描述在 Entity 上。

下面的代码取自 JFox Petstore AccountDAOImpl，这是负责操作 Account Entity 的 DAO Session Bean。

```

1      @NamedNativeQuery(
2          name = AccountDAOImpl.GET_ACCOUNT_BY_USERNAME_AND_PASSWORD,
3          query = "select " +
4              "signon.username as userid," +
5              "account.email," +
6              "account.firstname," +
7              "account.lastname," +
8              "account.status," +
9              "account.addr1," +
10             "account.addr2," +
11             "account.city," +
12             "account.state," +
13             "account.zip," +
14             "account.country," +
15             "account.phone," +
16             "profile.langpref," +

```

```

17         "profile.favcategory," +
18         "profile.mylistopt," +
19         "profile.banneropt," +
20         "bannerdata.bannername" +
21         " from account, profile, signon, bannerdata" +
22         " where account.userid = $username" +
23         " and signon.password = $password" +
24         " and signon.username = account.userid" +
25         " and profile.userid = account.userid" +
26         " and profile.favcategory = bannerdata.favcategory",
27     resultClass = Account.class,
28     hints = {
29         @QueryHint(name = "cache.default.partition", value = "account")
30     }
31
32 )

```

上面的代码定义了根据 username 和 password 取得 Account 的 NamedNativeQuery, query 语句是使用 velocity 模板化的标准 SQL, 其中有两个变量 \$username 和 \$password, 它们会在执行查询时赋值, resultClass 说明返回的对象是 Account.class 类型, hints 在 JPA 中用来描述附加信息, 上面的 QueryHint 用于定义该 JFox JPA query 使用 cache, 下面会具体讲到。

执行上面定义的 NamedNativeQuery 看起来会像下面的代码:

```

1 public Account getAccount(String username, String password)
2     throws SQLException {
3     Query query = createNamedNativeQuery(GET_ACCOUNT_BY_USERNAME_AND_PASSWORD)
4             .setParameter("username", username)
5             .setParameter("password", password);
6     return (Account)query.getSingleResult();
7 }

```

AccountDAOImpl 继承了 org.jfox.entity.dao.DAOSupport, DAOSupport 的目的是为了让 DAO 的编写更简单, 调用 DAOSupport 提供的 createNamedNativeQuery 方法创建 Query, 然后使用 setParameter 方法设置 username 和 password 参数, 最后执行 getSingleResult 返回的即是需要的 Account 对象了, 非常的简单吧。

注意: JFox JPA 没有实现 EJBQL 解析, 所以不支持使用 EJBQL 的@NamedQuery, 当然用户也不需要再去学习令人困惑的 EJBQL 了。

### 5.1.7.1 @Entity

@Entity 用来描述 JPA 的实体对象, 一般来说 Entity 类是一个纯的 Java Bean 类, 只是在 Field 上增加了@Column 描述用来定义 O/R 映射, 注意, JFox JPA 容器只会分析 Field

上的@Column，并不会分析方法上的@Column。

以下是一个 Entity 的例子，JFox 的 Entity 基本上只需要使用@Entity、@Column 便可以完成所有的定义，这比掌握 JPA 定义的所有 Annotation 要简单很多。

```
1 package org.jfox.petstore.entity;
2
3 import java.io.Serializable;
4 import javax.persistence.Column;
5 import javax.persistence.Entity;
6
7 @Entity
8 public class Product implements Serializable {
9
10    @Column(name = "productid")
11    String productId;
12
13    @Column(name = "category")
14    String categoryId;
15
16    @Column(name = "name")
17    String name;
18
19    @Column(name = "descn")
20    String description;
21
22    public String getCategoryId() {
23        return categoryId;
24    }
25
26    public void setCategoryId(String categoryId) {
27        this.categoryId = categoryId;
28    }
29
30    public String getDescription() {
31        return description;
32    }
33
34    public void setDescription(String description) {
35        this.description = description;
36    }
37
38    public String getName() {
39        return name;
40    }
```

```

41
42     public void setName(String name) {
43         this.name = name;
44     }
45
46     public String getProductId() {
47         return productId;
48     }
49
50     public void setProductId(String productId) {
51         this.productId = productId;
52     }
53 }
```

一般来说，你总会为一个数据库表定义 Entity 对象，并用@Entity 描述它，但有时候，你可能并没有也不想定义一个 Entity，比如：一个多表查询的结果。如果你在使用 @NamedNativeQuery 时遇到这个情况，那么你可以指定 resultClass 为：org.jfox.entity.MappedEntity，或者不指定默认值即是；而如果是一个临时的查询，那么你可以指定 createNativeQuery 的第二个参数为为 org.jfox.entity.MappedEntity，看这个例子：

```

1     public MappedEntity getAccountMappedEntityById(long id) throws SQLException {
2         Query query = createNativeQuery("select * from ACCOUNT where ACC_ID=$id",
3                                         MappedEntity.class)
4             .setParameter("id", id);
5         return (MappedEntity)query.getSingleResult();
6     }
```

得到 MappedEntity 之后，你可以通过 MappedEntity.getColumnValue(String columnName) 方法来取得数据库字段的值了。实际上，MappedEntity 使用了一个 Map 并以数据库列名为 Key 对应的值为 Value 来保存查询的结果。

由于 JFox JPA 使用的 SQLTemplate 对 ID 没有特别的要求，所以@Id 没有意义，所以字段只需要用@Column 描述即可，你会惊喜的发现，JFox JPA 只需要你熟悉非常少数的几个 JPA Annotation 就可以完成所有的操作。

### 5.1.7.2 @Column and @MappingColumn

@Column 是 JPA 的标准 Annotation，用来描述类字段和数据库字段的映射，在 JFox 3 中，只需要指明 name 属性即可。JFox JPA 会@Column 所描述的字段的类型将取回的数据转换成正确的类型。

@MappingColumn 是 JFox 扩展的 Annotation，用来描述关系字段，需要设置 namedQuery

和 params 两个属性，在容器构造 Entity 对象时，会通过@MappingColumn 的 nameQuery 和 params 构造查询来获得所映射的对象，并为所描述的 Field 赋值。

@MappingColumn 的定义如下：

```

1 @Retention(RetentionPolicy.RUNTIME)
2 @Target({ElementType.FIELD, ElementType.METHOD})
3 public @interface MappingColumn {
4     /**
5      * 用来查询的 query
6      */
7     String namedQuery();
8
9     /**
10      * 传给 namedQuery 的参数，要引用改 Entity，使用$this，比如 $this.getId()
11      */
12     ParameterMap[] params() default {};
13 }
```

下面是一个使用@Column 和@MappingColumn 描述的 Entity：

```

1 @Entity
2 public class Account {
3
4     @Column(name="ACC_ID")
5     long id;
6
7     @Column(name="ACC_FIRST_NAME")
8     String firstName;
9
10    @Column(name="ACC_LAST_NAME")
11    String lastName;
12
13    @Column(name="ACC_EMAIL")
14    String mail;
15
16    @MappingColumn(namedQuery = AddressDAO.GET_ADDRESS_BY_ACCOUNT_ID,
17                  params = {@ParameterMap(name = "accountId", value="$this.getId()")})
18    Address address;
19
20 }
```

解释一下其中的@MappingColumn，namedQuery 指明了用来查询 Address 的 NamedNativeQuery 的名称，看一下 AddressDAO 中定义的名为 AddressDAO.GET\_ADDRESS\_BY\_ACCOUNT\_ID 的 NamedNativeQuery，它的定义如下：

```
1 @NamedNativeQuery(name = AddressDAO.GET_ADDRESS_BY_ACCOUNT_ID,
```

```
2         query = "select * from address where ADR_ACC_ID = $accountId",
3         resultClass = Address.class)
```

可以看出，这个 query 需要一个参数\$accountId，所以需要在@MappingColumn 的 params 用@ParameterMap 指定这个参数，参数名为 accountId，值为\$this.getId()，\$this 是 JFox JPA 的 SQLTemplate 中的特殊变量，用来指该 Entity，上例中即指 Account 对象。

@MappingColumn 的 params 属性是一个@ParameterMap 数组，如果 query 中需要多个参数，则需要指定多个@ParameterMap。

注意：@Column 的 name 属性实际上指的是 SQL 查询结果 ResultSet 中的列名，所以如果@Column 的 name 保持和数据库列名一致，那么在查询时就要慎用”as”；而所以如果数据库中表的列名和@Column 指定的名称和不匹配，则构建查询时可以通过”as”建立别名来和@Column 保持一致。

对于 Lob 大字段，在 JFox JPA 与一般的字段没有不同，不需要使用@Lob 来标注，看下面的代码。

```
1 package jfox.test.ejb3.lob;
2
3 import javax.persistence.Column;
4 import javax.persistence.Entity;
5
6 /**
7  * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
8 */
9 @Entity
10 public class Lobber {
11
12     @Column(name="id")
13     long id;
14
15     @Column(name="blobby")
16     byte[] blobby;
17
18     @Column(name="clobby")
19     String clobby;
20
21     public long getId() {
22         return id;
23     }
24
25     public void setId(long id) {
26         this.id = id;
27     }
28
29     public String getClobby() {
```

```
30     return clobby;
31 }
32
33 public void setClobby(String clobby) {
34     this.clobby = clobby;
35 }
36
37 public byte[] getBlobby() {
38     return blobby;
39 }
40
41 public void setBlobby(byte[] blobby) {
42     this.blobby = blobby;
43 }
44 }
```

### 5.1.7.3 DataSource 和 Cache

JFox JPA 使用 XAPool (<http://xapool.objectweb.org>) 提供的 XADatasource 来支持连接池以及和 Transaction Manager 协作以支持事务处理。

数据源的配置通过修改 persistence.xml 来完成, persistence.xml 的格式是由 JPA 的标准定义的, 它的配置已经在上面讲过。下面再展示了一下 persistence.xml 的配置。

```
1 <persistence>
2
3     <persistence-unit name="default">
4         <jta-data-source>java:/DefaultMySqlDS</jta-data-source>
5         <properties>
6             <property name="driver" value="com.mysql.jdbc.Driver"/>
7             <property name="url" value="jdbc:mysql://localhost:3306/test"/>
8             <property name="username" value="root"/>
9             <property name="password" value="root"/>
10            <property name="minSize" value="1"/> <!-- min pool size -->
11            <property name="maxSize" value="200"/> <!-- max pool size -->
12            <property name="lifeTime" value="1800000"/> <!-- 3h, connection max idle time,
in ms -->
13            <property name="sleepTime" value="600000"/> <!-- PoolKeeper sleep time, in ms
-->
14            <property name="deadLockRetryWait" value="2000"/> <!-- retry time if no free
connection, in ms -->
15            <property name="deadLockMaxWait" value="60000"/> <!-- max wait time if no free
connection, in ms -->
```

```

16      <property name="checkLevelObject" value="4"/> <!-- check connection closed -->
17      <property name="cache.algorithm" value="LRU"/> <!-- algorithm for "default"
cache category, LRU, LFU, FIFO-->
18      <property name="cache.ttl" value="600000"/> <!-- ttl for "default", in ms-->
19      <property name="cache.maxIdleTime" value="300000"/> <!-- maxidletime for
"default", in ms-->
20      <property name="cache.maxSize" value="1000"/> <!-- max size for "default" -->
21      <property name="cache.maxMemorySize" value="100000000"/> <!-- max memory size
for "default", in bytes-->
22    </properties>
23  </persistence-unit>
24
25 </persistence>

```

这里主要解释一下了 JFox JPA 的 Cache。JFox JPA 的 Cache 是基于 SQL Statement 来实现的，工作原理如下：执行一个 Statement 之后，比如 Select 语句，JFox 会将结果集缓存，如果下次进行同样的查询时，将直接从返回内存中缓存的结果集，判断是否是相同的查询的条件包括同样的 SQL 语句以及参数；而一旦对这些数据进行过写的操作，JFox 会将缓存的结果集销毁，下次查询时，将重新访问数据库，并重建缓存。这样的 Cache 的设计简单、高效，易于维护，并有效的避免了脏数据的产生。

JFox JPA Cache 有一个重要的概念，分区(Partition)。分区是为了细分 Cache 的区域，使得能够以更小的粒度重建 Cache，避免频繁的重建整个 Cache 而降低了 Cache 的工作效率。分区在 @NamedNativeQuery 中使用 @hints 来配置。

```

1  @NamedNativeQuery(
2      name = AccountDAOImpl.GET_USERNAME_LIST,
3      query = "select username as userid from signon",
4      resultClass = String.class,
5      hints = {
6          @QueryHint(name = "cache.partition", value = "account")
7      }
8
9  ),
10
11 @NamedNativeQuery(
12     name = AccountDAOImpl.UPDATE_ACCOUNT,
13     query = "update account set " +
14         "email = $account.getEmail(), " +
15         "firstname = $account.getFirstName(), " +
16         "lastname = $account.getLastName(), " +
17         "status = $account.getStatus(), " +
18         "addr1 = $account.getAddress1(), " +
19         "addr2 = $account.getAddress2(), "

```

```

20           "city = $account.getCity(), " +
21           "state = $account.getState(), " +
22           "zip = $account.getZip(), " +
23           "country = $account.getCountry(), " +
24           "phone = $account.getPhone() " +
25           "where " +
26           "userid = $account.getUsername(),
27 hints = {
28     @QueryHint(name = "cache.partition", value = "account")
29   }
30
31 ),

```

Cache 的销毁和重建是分区(partition)为单位的，比如上面的例子，当有用户修改了 Account 信息时，也就是 query `AccountDAOImpl.UPDATE_ACCOUNT` 被执行时，只会销毁 account 分区的 Cache，而有其它的 Cache(如 Product、Order)则不会收到影响。所以合理的使用分区，将能大大提高应用程序的性能。比如：把很少被修改的数据，放在一个分区中；而把因为会被经常修改而需要重载的数据分别放在多个较小粒度的分区当中就是一个不错的做法。

在 JFox 管理控制台中可以查看 Cache 的使用情况，以及手动销毁 Cache。

#### 5.1.7.4 ID 生成器

一些 O/R Mapping 工具都提供了多种 ID 生成方式，比如：建立 ID 生成表或使用 Sequence 等。JFox JPA 提供了完全基于时间的 ID 生成器 `org.jfox.entity.dao.PKGenerator`，根据时间生成一个 19 位的 long 型数字，如：2006080816404856650。

如果要使用 JFox JPA 提供的 ID 生成器，请将数据库表的 ID 字段设为相应的类型，如：NUMBER(19) 或 NUMERIC(19)，而对应的 Java 类型为 long。

DAOSupport 的 `nextPK()` 方法使用 PKGenerator 生成 PK。

```

1 /**
2  * PK 只保证唯一，不包含任何业务意义，比如：对于 ID 连续性的要求
3 */
4 public long nextPK() {
5     return PKGenerator.getInstance().nextPK();
6 }

```

注意：由于 ID 生成器依赖于系统时间，所以在系统上线时调整好系统时间，如果今后把系统时间改为已过去的时间，则可能造成 ID 重复的危险；另外，由于 JFox JPA 采用的是基于 SQLTemplate 的实现，所以并不需要对 ID 字段做特殊处理，对于 ID 字段并不需要用@Id 来标注。

### 5.1.7.5 DAO (Data Access Object)

EJB3 中，DAO 显得尤为重要，因为需要通过 DAO 注入 JPA Persistence Context，以让 JPA 容器和 EJB 容器协作，推荐将 DAO 实现为 Stateless Session Bean，使用 @PersistenceContext 注入 EntityManager 以获得操作 Entity 的能力。

JFox 已经封装好了 DAOSupport 超类，提供了 newEntityObject、createNativeQuery 和生成 ID 的方法，还定义了 getEntityManager 抽象方法，由子类提供 EntityManager 实例。

下面是 DAOSupport 的代码：

```

1 package org.jfox.entity.dao;
2
3 import java.util.HashMap;
4 import java.util.Map;
5 import javax.persistence.EntityManager;
6
7 import org.jfox.entity.EntityFactory;
8 import org.jfox.entity.MappedEntity;
9 import org.jfox.entity.QueryExt;
10
11 /**
12 * DAOSupport, 封装了 DAO 的基本操作
13 * 需要由子类提供 EntityManager, 在 getEntityManager 方法中返回
14 *
15 * DAO 应该都从 DAOSupport 继承，并且实现为 Stateless Local SessionBean
16 *
17 * DAO 可以脱离容器运行，此时不能使用@PersistenceContext 注入，
18 * 而只能通过 javax.persistence.Persistence 的静态方法来构造
19 *
20 * @author <a href="mailto:jfox.young@gmail.com">Yang Yong</a>
21 */
22 public abstract class DAOSupport implements DataAccessObject {
23
24
25     /**
26      * 返回 EntityManager, 由子类使用 @PersistenceContext 注入
27      */
28     protected abstract EntityManager getEntityManager();
29
30     /**
31      * 根据 Entity Class 生成 Entity 对象
32      *
33      * @param entityClass entity class
34      */

```

```
35     public static <T> T newEntityObject(Class<T> entityClass) {
36         return newEntityObject(entityClass, new HashMap<String, Object>());
37     }
38
39     /**
40      * 使用数据 Map, 生成 Entity 对象, Map 的 key 和 Entity 的 Column 对应
41      *
42      * @param entityClass entity class
43      * @param dataMap data dataMap
44      */
45     public static <T> T newEntityObject(final Class<T> entityClass, final Map<String, Object>
dataMap) {
46         return EntityFactory.newEntityObject(entityClass, dataMap);
47     }
48
49     /**
50      * 给定 query name 创建 NamedNativeQuery
51      * @param queryName query name
52      */
53     public final QueryExt createNamedNativeQuery(String queryName) {
54         return (QueryExt) getEntityManager().createNamedQuery(queryName);
55     }
56
57     /**
58      * 给定 sql 构造 Query, Result 类型为 MappedEntity
59      * @param sql native sql template
60      */
61     public final QueryExt createNativeQuery(String sql) {
62         return createNativeQuery(sql, MappedEntity.class);
63     }
64
65     /**
66      * 给定 sql 构造 Query, Result 类型为 resultClass
67      * @param sql native sql template
68      * @param resultClass 返回的结果对象类型
69      */
70     public final QueryExt createNativeQuery(String sql, Class<?> resultClass) {
71         return (QueryExt) getEntityManager().createNativeQuery(sql, resultClass);
72     }
73
74     /**
75      * 生成 19 的 PK, 比如: 2006080816404856650
76      * PK 只保证唯一, 不包含任何业务意义, 比如: 对于 ID 连续性的要求
77      */
```

```
78     public long nextPK() {
79         return PKGenerator.getInstance().nextPK();
80     }
81
82 }
83
```

下面的代码取自 JFox Petstore 提供的 CategoryDAOImpl。

```
1 package org.jfox.petstore.dao;
2
3 import java.sql.SQLException;
4 import java.util.List;
5 import javax.ejb.Local;
6 import javax.ejb.Stateless;
7 import javax.persistence.EntityManager;
8 import javax.persistence.NamedNativeQueries;
9 import javax.persistence.NamedNativeQuery;
10 import javax.persistence.PersistenceContext;
11
12 import org.jfox.entity.dao.DAOSupport;
13 import org.jfox.petstore.entity.Category;
14
15 /**
16  * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
17  */
18 @NamedNativeQueries(
19     {
20         @NamedNativeQuery(
21             name = CategoryDAOImpl.GET_CATEGORY,
22             query = "select catid, name, descn from category where catid = $id",
23             resultClass = Category.class
24         ),
25         @NamedNativeQuery(
26             name = CategoryDAOImpl.GET_CATEGORY_LIST,
27             query = "select catid, name, descn from category",
28             resultClass = Category.class
29         )
30     }
31 )
32 @Stateless
33 @Local
34 public class CategoryDAOImpl extends DAOSupport implements CategoryDAO{
35
36     public static final String GET_CATEGORY = "getCategory";
```

```
37     public static final String GET_CATEGORY_LIST = "getCategoryList";
38
39     @PersistenceContext(unitName = "JPetstoreMysqlDS")
40     EntityManager em;
41
42     protected EntityManager getEntityManager() {
43         return em;
44     }
45
46     public Category getCategory(String categoryId) throws SQLException {
47         return
(Category)createNamedNativeQuery(GET_CATEGORY).setParameter("id", categoryId).getSingleResult();
48     }
49
50     public List<Category> getCategoryList() throws SQLException {
51         return (List<Category>)createNamedNativeQuery(GET_CATEGORY_LIST).getResultList();
52     }
53
54 }
55
```

CategoryDAOImpl 继承了 DAOSupport，并实现 CategoryDAO，标注成 @Stateless 和 @Local，并使用@NamedNativeQuery 定义了 query。

其它的 EJB 使用 CategoryDAO 时，可以通过@EJB 注入：

```
1 import java.util.List;
2 import java.util.Collections;
3 import javax.ejb.EJB;
4 import javax.ejb.Stateless;
5 import javax.ejb.Local;
6
7 import org.jfox.petstore.dao.CategoryDAO;
8 import org.jfox.petstore.entity.Category;
9
10 /**
11  * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
12 */
13 @Stateless
14 @Local
15 public class CategoryB0Impl implements CategoryB0{
16
17     @EJB
18     CategoryDAO categoryDAO;
19
```

```

20     public Category getCategory(String categoryId) {
21         try {
22             return categoryDAO.getCategory(categoryId);
23         }
24         catch(Exception e) {
25             e.printStackTrace();
26             return null;
27         }
28     }
29
30     public List<Category> getCategoryList() {
31         try {
32             return categoryDAO.getCategoryList();
33         }
34         catch(Exception e) {
35             e.printStackTrace();
36             return Collections.emptyList();
37         }
38     }
39 }
```

### 5.1.7.6 如何分页

分页是查询中的普遍问题，JPA 的 Query 接口中已经提供了两个方法以方便在查询的时候实现分页，它们分别是：

```
Query.setFirstResult(int startPosition)
```

```
Query.setMaxResults(int maxResult)
```

setFirstResult 用来设置取值的起始位置，从 0 开始计数；setMaxResults 用来设置取多少行记录。

利用这两个方法，便可以轻松进行分页了，下面的例子展示了分页的用法：

```

1 List<Account> accounts = em.createNativeQuery("select * from account", Account.class)
2             .setFirstResult(2)
3             .setMaxResults(2)
4             .getResultList();
5
```

需要注意的是：JFox 实现分页时，采用的并不是物理分页，而是使用 JDBC ResultSet 的游标定位来实现的，一般情况下可以满足对性能的要求，但是如果你期望获得最大的性能，则建议根据不同的数据库构造 SQL 语句来实现。

### 5.1.7.7 如何支持多数据库

应用 JFox JPA，可以在应用中方便的支持多数据库。由于 JFox JPA 采用的是 SQLTemplate 方式，所以需要一定的编码，但好处就是，可以充分应用不同数据库类型的特性，让程序在不同类型的数据库上都能获得最佳的性能。

为了支持多种数据库，需要为不同的数据库定义 NamedNativeQuery，因为这些 NamedNativeQuery 使用了相同的名称，那么为了区分它们，则必须再 hints 中用 “jdbc.compatible” 指定其使用的数据库类型，并用逗号分隔。如下表：

```

1 @NamedNativeQuery(
2     name = AccountDAOImpl.GET_USERNAME_LIST,
3     query = "select username as userid from signon",
4     resultClass = String.class,
5     hints = {
6         @QueryHint(name = "cache.partition", value = "account"),
7         @QueryHint(name = "jdbc.compatible", value =
"mysql,sqlserver,oracle,db2")
8     }
9
10 )

```

那么，上例中，如果需要支持新的数据库（newDB），且 sql 语句不同，则需要增加类似如下的 NamedNativeQuery：

```

1 @NamedNativeQuery(
2     name = AccountDAOImpl.GET_USERNAME_LIST,
3     query = "查询 username as userid from signon",
4     resultClass = String.class,
5     hints = {
6         @QueryHint(name = "cache.partition", value = "account"),
7         @QueryHint(name = "jdbc.compatible", value = "newDB")
8     }
9
10 )

```

JPA 容器会根据当前连接的数据库类型选择正确的 NamedNativeQuery，没有配置 “jdbc.compatible” QueryHint 的 NamedNativeQuery 则被视为通用的，即满足所有数据库类型。如果同时存在通用 NamedNativeQuery 和满足数据库类型的特定 NamedNativeQuery，则会优先采用特定 NamedNativeQuery。

### 5.1.7.8 了解 JFox JPA 局限

JFox JPA 采用的是基于 SQLTemplate 的实现，SQLTempalte 被证明是一种简单直接的持

久化方式，它能满足企业应用对 O/R 的需求，并提供了更好的可控性，方便应对复杂的数据  
库问题以及后期的查错和调优，而 JFox JPA 在保持 SQLTemplate 特性的前提下，提供了与  
JPA 接口的兼容，同时，由于 SQLTemplate 的特性，有些 JPA 内容则没有必要或者不需要去  
实现。

JFox JPA 没有实现的主要有：

1. Id
2. persist, merge, remove
3. EJBQL
4. EntityListener 和 callback

## 5.2 JFox 微内核

JFox 微内核是 JFox 应用服务器的基础结构，提供了 IoC 注入、Interceptor、面向组  
件编程以及模块化能力，JFox 采用的是自主研发的 IoC 微内核，存在于包  
org.jfox.framework 下，它和 JFox 应用服务器一起，经过了反复的重构和改进。

了解 JFox 内核，将能够更好的理解 JFox 应用服务器的工作机制，以及基于内核编写新  
的服务扩展 JFox 应用服务器的功能。

要了解 JFox 基于 IoC 的微内核，需要了解几个重要的概念：Framework、Module 、  
Component。

- Framework 用来负责启动以及停止内核，同时它也管理整个微内核的资源，以及负  
责出发和传递事件。
- Module 是模块，一个模块用来完成一个领域的问题，模块拥有封闭性，通过发布  
接口和其它模块进行交互，模块有自己个独立的 ClassLoader。Framework 会根据  
web.xml 中的 MODULES\_DIR 配置加载多个模块，Framework 将本身也加载为一个模  
块，模块名为\_\_SYSTEM\_MODULE\_\_。在后面的 JFox Module 一章中，你将看到看到  
可以将一个应用模块做一个 Module 发布到 JFox 应用服务器中。
- Component 即为一个组件，用来实现一个特定的功能，组件被实现为 Java 类，它  
们都从 org.jfox.framework.component.Component 继承而来，组件总是被加载在  
一个模块中。

### 5.2.1 Component

org.jfox.framework.component.Component 只是一个申明类，并没有定义任何方法，  
但是所有的组件都应该实现该接口，JFox 内核会根据该接口判断组件的有效性。

有四个接口直接继承了 Component 接口，用来定义不同类型的组件，他们分别是：

1. org.jfox.framework.component.SingletonComponent
2. org.jfox.framework.component.ActiveComponent
3. org.jfox.framework.component.InterceptableComponent
4. org.jfox.framework.component.RunnableComponent

- **SingletonComponent**

SingletonComponent 通过接口来强制了 Component 的构造行为，SingletonComponent 表明该组件总是单例的，而忽略@Service(singleton=false) 描述。

如果一个 Component 没有实现 SingletonComponent 接口，也可以通过@Service(singleton=true) 来指明该组件是单例的，而当父类想强制其子类均应该为单例时，则在父类中实现 SingletonComponent 接口是更好的方式。

- **ActiveComponent**

ActiveComponent 表明该组件是活跃的，意味着该组件在由 IoC 容器加载之后，会立即进行实例化，这对于一些需要立即提供服务的组件是有用的，比如 JFox 的 EJB 容器就是实现了该接口，以便加载即服务。

和 SingletonComponent 一样，也可以用@Service(active=true) 来描述一个组件为 Active，而如果父类想强制其子类均应为 Active 时，则在父类中实现 ActiveComponent 接口更好。

- **InterceptableComponent**

InterceptableComponent 定义了两个新的方法，以在执行 Component 方法前后进行进行拦截，这两个新的方法是：

```

1 package org.jfox.framework.component;
2
3 import java.lang.reflect.Method;
4
5 /**
6  * 可进行方法拦截的组件。
7  * 实现该接口的组件，在执行器方法的时候，可以在方法执行前后，进行额外的操作，比如：进行日志记录
8  * 注意：该功能通过 Java 动态代理技术实现
9  *
10 * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
11 */
12 public interface InterceptableComponent extends Component {
13     /**
14      * do something before invoke a method defined in component interface
15      * example: want to log, or repack the params
16      *
17      * @param method method to invoke
18      * @param params parameter array
19      * @return true/false, if false, will not invoke the method
20      */
21     boolean preInvoke(Method method, Object[] params);
22
23     /**
24      * do something after invoke a method example: want to repack the result
25      *
26      * @param method method invoked
27      * @param params parameter array

```

```

28     * @param result result return by the method
29     * @param exception if throws exception
30     * @return the last object return, may not be the result parameter
31     */
32     Object postInvoke(Method method, Object[] params, Object result, Throwable exception);
33
34 }
35

```

如果需要在执行方法前后进行拦截工作，比如进行日志，那么可以实现 InterceptableComponent 接口。

preInvoke 如果返回 false，将会拒绝执行 Component 的方法，并抛出异常。

- **RunnableComponent**

RunnableComponent 继承了 java.lang.Runnable 接口，如果你需要编写一个提供服务的线程组件，比如 Socket Server，那么可以直接实现 RunnableComponent 接口。

## 5.2.2 ComponentContext

ComponentContext 用来描述组件上下文环境，通过 ComponentContext 可以得到 Component 相关信息(比如：Component Id)，也可以用来和内核进行交互(比如：根据 id 和接口来查找其它的组件，用来发出 ComponentEvent)等。

如果你想访问 ComponentContext，通过实现 ComponentInitialization 接口，并在 postConstruct 方法中可以传入，ComponentInitialization 在后面会讲到。

## 5.2.3 Annotation

与一般 IoC 容器采用 xml 配置不同，JFox IoC 容器全部采用 JDK5 的 Annotation 来标注组件的属性以及依赖关系，一共有 4 个 Annotation：@Service，@Exported，@Inject，@Constant。

- **@Service**

@Service 用来描述一个 Component 以注册到 IoC 容器中，组件实现完之后，必须使用 @Service 标注才能被内核加载。IoC 容器启动时，会搜索 classpath 中所有描述了@Service 的类，将合法的 Component 按要求注册到 IoC 容器中。

Component 在注册到容器之后，均会被分配一个 Component Id，Component Id 可以由 @Service 的 id 属性指定，如未指定，则为 Component 实现类的类名的简称 (Class.getSimpleName())，Component Id 用来在 IoC 容器中精确查找一个组件。

@Service 的定义如下：

```

1 package org.jfox.framework.annotation;
2
3 import java.lang.annotation.Target;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;

```

```
6 import java.lang.annotation.ElementType;
7
8 /**
9  * 描述在 Component 上以便将该 Component 发布至 IoC 容器中
10 *
11 * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
12 */
13 @Retention(RetentionPolicy.RUNTIME)
14 @Target({ElementType.TYPE})
15 public @interface Service {
16     /**
17      * 该组件实现的接口，组件只能通过指定的接口来访问，
18      * 如果没有指定接口呢，则为所有接口
19      */
20     Class[] interfaces() default {};
21
22     /**
23      * 部署的 id
24      * Default 为 Component 实现类的类名的简称
25      */
26     String id() default "";
27
28     /**
29      * 是否在 getInstance 时，才实例化
30      * 否则，在部署之后，立即进行实例化
31      */
32     boolean active() default false;
33
34     /**
35      * 是否单例，单例表示该组件只能部署一次
36      */
37     boolean singleton() default true;
38
39     /**
40      * 优先级，在一个模块中组件，优先级值小的会被加载
41      */
42     int priority() default 50;
43
44     /**
45      * 描述信息
46      */
47     String description() default "";
48
49 }
```

```
50 }
51
```

@Service 的 active 和 singleton 属性比 ActiveComponent、SingletonComponent 接口的优先级低，如果组件实现了以上接口，对应的 active 和 singleton 将不起作用。

- **@Inject**

@Inject 用来标注组件的 Field，以使得该组件在实例化的时候，由 IoC 容器进行依赖注入。

```
1 package org.jfox.framework.annotation;
2
3 import java.lang.annotation.Target;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.ElementType;
7
8 /**
9  * 用来标注一个组件的 Field，以使得该组件在实例化的时候，由 IoC 容器进行依赖注入
10 *
11 * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
12 */
13 @Retention(RetentionPolicy.RUNTIME)
14 @Target({ElementType.FIELD})
15 public @interface Inject {
16
17     /**
18      * 注入的组件的类型
19      * 如果未指定，则由 IoC 容器根据 Filed 的类型判断
20     */
21     Class type() default FieldType.class;
22
23     /**
24      * 通过 component id 进行注入
25     */
26     String id() default "";
27
28     public static class FieldType {}
29 }
30
```

可以通过 type 和 id 进行注入，type 会通过给定的接口注入满足条件的组件，如果有多个组件满足要求，将会抛出异常；id 则提供根据 Component Id 进行精确注入。如果均没有指定，将根据所描述的 Field 的 Type 进行注入。

- **@Constant**

@Constant 用来为 Field 注入常量，注入的常量值用 @Constant 的 value 属性指定，value 的值实际为一个 Velocity 表达式，会使用 Velocity 模板引擎并使用在 global.properties 中定义的全局占位符进行解析，这使得可以动态注入常量，弥补了 Annotation 相对 XML 配置在编译时确定的弱点，增加了程序的动态配置能力。

@Constant 的定义如下：

```

1 package org.jfox.framework.annotation;
2
3 import java.lang.annotation.Target;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.ElementType;
7
8 /**
9  * 注入一个常量，可以使用 placeholder，placeholder 由 global.properties 导入
10 *
11 * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
12 */
13 @Retention(RetentionPolicy.RUNTIME)
14 @Target({ElementType.FIELD})
15 public @interface Constant {
16
17     /**
18      * 对于值注射时，值的对象类型，默认为 Field 本身的类型
19      */
20     Class type() default FieldType.class;
21
22     /**
23      * 注射一个固定的值，可以带有占位符(如: ${database.driver})，会在注入之前使用 Velocity
24      * 解析
25      */
26     String value() default "";
27
28     /**
29      * 由解析器负责根据 Injection 描述的 Field/Constructor 来判断真实的 type
30      */
31
32 }
33

```

在 global.properties 中定义全局占位符，目前 global.properties 的内容如下：

```

1 #---- JFOX3 Default Gloable PlaceHolder definition, don't delete -----
2
3 # jta transaction timetou, in second, used in SimpleEJB3Container

```

```
4 jta_transaction_timeout=60
5
6
7 #---- user placeholder definition here -----
```

在 SimpleEJB3Container 中使用了定义 jta\_transaction\_timeout 常量：

```
1 // default Transaction timeout
2 @Constant(type = Integer.class, value = "$jta_transaction_timeout")
3 private int transactionTimeout = 60; // default transaction timeout 60 seconds
4
```

那么 SimpleEJB3Container 实例化时，会将 \$jta\_transaction\_timeout 替换成 global.properties 中配置的值，然后进行注入。

更多关于全局占位符的内容在下面还将讨论。

### ● @Exported

由于 JFox IoC 容器设计支持模块化，不同的模块使用不同 ClassLoader 来加载，所以模块之间是隔离的，如果要使一个模块中的接口能够被其它的模块引用，则需要使用 @Exported 来标注，@Exported 标注的接口能在不同模块的类加载器之间共享。

@Export 的定义如下：

```
1 package org.jfox.framework.annotation;
2
3 import java.lang.annotation.Target;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.ElementType;
7
8 /**
9 * Exported 用来标注一个 Component 接口,
10 * 标注了的 Component 接口将能够被所有的模块发现,
11 * 所以如果一个 Component 要跨模块提供服务, 则需要将服务接口使用 @Exported 描述
12 *
13 * 描述了 Exported 的接口在 Module reload 的时候, 也不会重新加载
14 * Exported 描述在 Class 上, 没有意义
15 *
16 * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
17 */
18 @Retention(RetentionPolicy.RUNTIME)
19 @Target({ElementType.TYPE})
20 public @interface Exported {
21
22 }
23
```

比如：JAASLoginService 组件使用@Exported 进行了标注，这样，其它的模块才能引用它，以便使用它提供的 JAAS 登录服务。

```

1 @Exported
2 public interface JAASLoginService extends Component

```

注意：@Export 标注的接口，不会在模块重启的时候重新加载，所以如果@Exported 标注的接口发生的变化，则必须重启 JFox 应用服务器，或者重启 JFox 部署的 Web Application。

## 5. 2. 4 事件和监听器

JFox 内核设计了各种 Event，用来通知 Component 容器内发生的事件，有 3 种事件类型：

- org.jfox.framework.event.ComponentEvent
- org.jfox.framework.event.ModuleEvent
- org.jfox.framework.event.FrameworkEvent

分别表示 Component、Module、Framework 发出的事件。

组件可以通过实现对应的 3 个 Event Listener：

- org.jfox.framework.event.ComponentListener
  - org.jfox.framework.event.ModuleListener
  - org.jfox.framework.event.FrameworkListener
- 来监听对应事件。

JFox 的 EJB Container 即是通过监听 ModuleEvent 来实现 load/unload EJB 的，代码片段如下：

```

1 /**
2  * 监听 Module 事件，根据 Module 的 load/unload 事件来加载其中的 EJB
3  *
4  * @param moduleEvent module event
5  */
6 public void moduleChanged(ModuleEvent moduleEvent) {
7     Module module = moduleEvent.getModule();
8     if (moduleEvent instanceof ModuleLoadingEvent) {
9         EJBBucket[] buckets = loadEJB(module);
10        for (EJBBucket bucket : buckets) {
11            bucketMap.put(bucket.getEJBNName(), bucket);
12        }
13    } else if (moduleEvent instanceof ModuleUnloadedEvent) {
14        unloadEJB(module);
15    }
16 }

```

## 5.2.5 生命周期回调方法

JFox IoC 容器还提供了关于 Component 生命周期的回调方法，它们分别定义在两个接口中：

- org.jfox.framework.component.ComponentInitialization
- org.jfox.framework.component.ComponentUnregistration

ComponentInitialization 定义了组件在实例化之后以及依赖注入完成之后的回调方法：

```

1 package org.jfox.framework.component;
2
3 /**
4 * ComponentInitialization 提供了两个回调方法,
5 * 使得在组件构造之后, 以及依赖注入完成之后进行额外的操作。
6 *
7 * instantiated 方法是唯一可以传入 ComponentContext 的回调方法,
8 * 如果 Component 需要使用 ComponentContex, 就必须实现该接口
9 *
10 * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
11 */
12 public interface ComponentInitialization {
13
14     /**
15      * Component 实例化之后的回调方法
16      * 可以做依赖注入开始前的初始化操作
17      * @param componentContext Component context
18      */
19     public void postConstruct(ComponentContext componentContext);
20
21     /**
22      * Component 依赖注入完成之后的回调方法
23      * 可以做通过该方法进行注入之后额外的检查工作, 以及做组件初始化操作
24      */
25     public void postInject();
26
27 }
28

```

通过实现 ComponentInitialization 接口，还能获得一个好处，就是通过 postConstruct 获得 ComponentContext，如果你需要使用 ComponentContext，那么实现 ComponentInitialization 是唯一的途径。

ComponentUnregistration 定义了在组件被注销之前和注销之后的回调方法。

```

1 package org.jfox.framework.component;
2
3 /**
4 * 组件被注销之前和注销之后的回调方法。
5 *
6 * 组件注册的时候，没有回调方法，因为此时组件还没有实例化。
7 *
8 * @See ComponentInitialization
9 *
10 * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
11 */
12 public interface ComponentUnregistration {
13
14     /**
15      * 组件被注销之前的回调方法
16      *
17      * @param context component context
18      * @return true if can be unregister, false can not be unregister
19      */
20     boolean preUnregister(ComponentContext context);
21
22     /**
23      * 组件被注销之后的回调方法，进行注销之后的资源清除操作。
24      */
25     void postUnregister();
26 }
27

```

如果在注销时，需要做额外的动作，比如：关闭文件或者网络连接，那么有必要实现 ComponentUnregistration 接口。

## 5.2.6 全局占位符

@Constant 一节讲到了使用常量注入时，可以使用全局占位符，全局占位符提供了根据外部配置注入常量的能力，从而增加了使用 Annotation 注入的灵活性。全局占位符在 global.properties 中进行配置，位于 WEB-INF/classes 目录下，使用 java property 格式进行配置。使用 @Constant 对 Field 进行常量注入时，系统会调用 Velocity (<http://velocity.apache.org>) 引擎使用 global.properties 定义的只对 @Contant 的 value 进行解析，将解析之后的值注入给 Field.

如在 gobal.properties 有如下定义：

APPLICATION\_NAME = JFOX Example Application

而在 ExampleComponent 有以下 Field:

```
@Constant(value=" $APPLICATION_NAME 3.0" )  
private String appName;
```

那么 ExampleComponent 由 IoC 容器实例化之后，其 appName 的值会设置成：  
JFOX Example Application 3.0。

## 5.3 JFox MVC 框架

JFox 提供 MVC 框架，它直接建立在 JFox 微内核之上，所以获得了 IoC 以及模块化能力，不仅让你可以在 Action 类中方便的注入 EJB 以及其它组件，而且完全可以满足你按模块设计、开发、部署和升级的需要。

有了 MVC，JFox 应用服务器为企业应用开发提供了 IoC + MVC + O/R Mapping + EJB3 的整套开发平台，以满足企业开发对快速化、模块化的需要。

### 5.3.1 web.xml

首先需要熟悉一下 JFox MVC 的 web.xml 配置文件，这个配置文件是定义在整个 JFox 应用中，与 JFox MVC 相关的有以下几段。

- **context-param**

下面的 XML 片段描述了一些与 JFox MVC 相关的参数：

```
1 <context-param>  
2   <!--encoding used by request & velocity -->  
3   <param-name>DEFAULT_ENCODING</param-name>  
4   <param-value>UTF-8</param-value>  
5 </context-param>  
6  
7 <context-param>  
8   <!-- modules deployment dir -->  
9   <param-name>MODULES_DIR</param-name>  
10  <param-value>WEB-INF/MODULES</param-value>  
11 </context-param>  
12  
13 <context-param>  
14   <!-- module's view template dir -->  
15   <param-name>VIEW_DIR</param-name>  
16   <param-value>views</param-value>  
17 </context-param>  
18  
19 <context-param>  
20   <!-- action url suffix -->  
21   <param-name>ACTION_SUFFIX</param-name>
```

```

22      <param-value>.do</param-value>
23  </context-param>
24

```

以上参数的作用如下：

Param-name	Param-value	说明
DEFAULT-ENCODING	UTF-8	定义默认编码为 UTF-8，主要作用于解析 Form 数据解码，以及 velocity/freemarker 加载模板文件时采用的编码
MODULES_DIR	WEB-INF/MODULES	应用模块所在的目录，默认为 WEB-INF/MODULES 目录，应用服务器启动时，会加载该目录下所有应用模块
VIEW_DIR	views	View 模板文件所处的路径，JFox MVC 使用模板引擎(velocity/freemarker)来输出 HTML，VIEW_DIR 配置的是模块中模板文件相对模块根目录所在的目录名，默认为 views。 这里放置的文件除了模板文件以外，image, jsp 等 web 可以访问的一切资源也都应放在该目录下(注：JFox MVC 仅支持将 jsp 作为模板使用)
ACTION_SUFFIX	.do	访问 ACTION 时使用的后缀名，默认为 .do

### ● Controller Servlet

Controller Servlet 用来为 JFox MVC 解析 Http 请求，根据规则将请求分发给 Action 执行，或者分发给模板引擎输出页面，起到控制器的作用。

下面的 XML 片段配置了 JFox MVC 的 Controller Servlet：

```

1  <servlet>
2      <servlet-name>controller_servlet</servlet-name>
3      <servlet-class>org.jfox.mvc.servlet.ControllerServlet</servlet-class>
4      <init-param>
5          <param-name>MAX_UPLOAD_FILE_SIZE</param-name>
6          <param-value>5000000</param-value>
7          <!-- 5Mbytes -->
8      </init-param>
9      <load-on-startup>1</load-on-startup>
10     </servlet>
11     <servlet-mapping>
12         <servlet-name>controller_servlet</servlet-name>
13         <url-pattern>/modules/*</url-pattern>
14     </servlet-mapping>

```

主要完成以下几项定义：

1. Controller Servlet 的类名为 org.jfox.mvc.servlet.ControllerServlet
2. MAX\_UPLOAD\_FILE\_SIZE 定义了上传文件的最大尺寸
3. Controller Servlet 响应所有 /modules/\* 开头的 URL，它会根据 URL 而把请求分发到正确的模块中，如：  
 http://localhost:8080/jfox/modules/petstore/index.jsp，会调用 petstore 模块中 views 目录下的 index.jsp，而 <http://www.jfox.org.cn/modules/manager/console.sysinfo.do> 则将调用 manager 模块下的 console Action 组件的 sysinfo 的 Action 方法。

关于 Action 组件以及 Action 方法将在下面讲到。

### ● Template Servlet

Template Servlet 负责使用 Velocity/Freemarker 模板引擎，根据传递的数据对配置的模板文件进行渲染，输出最终的 HTML 页面。

下面的 XML 片段配置了 JFox MVC 的 Template Servlet：

```

1 <servlet>
2   <servlet-name>template_servlet</servlet-name>
3   <servlet-class>org.jfox.mvc.servlet.TemplateServlet</servlet-class>
4   <init-param>
5     <description>velocity render</description>
6     <param-name>org.jfox.mvc.velocity.VelocityRender</param-name>
7     <param-value>*.vm, *.tmpl, *.vhtm, *.vhtml</param-value>
8   </init-param>
9   <init-param>
10    <description>freemarker render</description>
11    <param-name>org.jfox.mvc.freemarker.FreemarkerRender</param-name>
12    <param-value>*.ftl, *.fhtm, *.fhtml</param-value>
13  </init-param>
14  <load-on-startup>1</load-on-startup>
15 </servlet>
16 <servlet-mapping>
17   <servlet-name>template_servlet</servlet-name>
18   <url-pattern>*.vm</url-pattern>
19 </servlet-mapping>
20 <servlet-mapping>
21   <servlet-name>template_servlet</servlet-name>
22   <url-pattern>*.tmpl</url-pattern>
23 </servlet-mapping>
24 <servlet-mapping>
25   <servlet-name>template_servlet</servlet-name>
26   <url-pattern>*.vhtm</url-pattern>
```

```

27 </servlet-mapping>
28 <servlet-mapping>
29   <servlet-name>template_servlet</servlet-name>
30   <url-pattern>*.vhtml</url-pattern>
31 </servlet-mapping>
32 <servlet-mapping>
33   <servlet-name>template_servlet</servlet-name>
34   <url-pattern>*.ftl</url-pattern>
35 </servlet-mapping>
36 <servlet-mapping>
37   <servlet-name>template_servlet</servlet-name>
38   <url-pattern>*.fhtm</url-pattern>
39 </servlet-mapping>
40 <servlet-mapping>
41   <servlet-name>template_servlet</servlet-name>
42   <url-pattern>*.fhtml</url-pattern>
43 </servlet-mapping>
44

```

配置的内容主要有以下几点：

1. Template Servlet 类名为 org.jfox.mvc.servlet.TemplateServlet
2. 使用了两个 init-param 分别定义 velocity render 和 freemarker render，并为它们配置了的后缀名
3. servlet-mapping 将所有可能的模板后缀名都交给 Template Servlet 来处理，Template Servlet 接收到模板请求之后，会再次根据后缀配置选择交给 velocity 还是 freemarker 来处理

### 5.3.2 Action

JFox MVC Action 用来处理 Http 请求，和一般 MVC 框架的 Action 类不同，JFox MVC Action 可以拥有多个 Action 方法，Action 方法使用 @ActionMethod 描述。

JFox MVC 提供了 org.jfox.mvc.ActionSupport 来支持 Action 的编写，所以所有的 Action 类也必须从这里继承，ActionSupport 实现了 org.jfox.framework.component.Component，所以是 JFox 内核组件，也就具备 IoC 注入功能。

- Action 方法和 @ActionMethod

@ActionMethod 用来描述在 Action 类的一个方法上，Action 方法可以被 URL 直接访问，在一个 Action 类中可以描述多个 Action 方法，Action 方法必须具备如下条件：

1. 只有一个参数 InvocationContext
2. 返回 void
3. 抛出 Exception

有了@ActionMethod，JFox MVC 不再需要编写和维护类似 structs.xml 的配置文件了，从而简化了 Action 的配置。

下面是@ActionMethod 的定义：

```
1 /*
2  * JFox - The most lightweight Java EE Application Server!
3  * more details please visit http://www.huihoo.org/jfox or http://www.jfox.org.cn.
4  *
5  * JFox is licenced and re-distributable under GNU LGPL.
6  */
7 package org.jfox.mvc.annotation;
8
9 import java.lang.annotation.ElementType;
10 import java.lang.annotation.Retention;
11 import java.lang.annotation.RetentionPolicy;
12 import java.lang.annotation.Target;
13
14 import org.jfox.mvc.Invocation;
15
16 /**
17  * 表示一个 Action 方法
18  * 用该 Annotation 描述的方法，需要满足以下条件
19  * 1. 只有一个参数 Invotation
20  * 2. 返回 Void 类型
21  * 3. 抛出 Exception 异常
22  *
23  * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
24  */
25 @Retention(RetentionPolicy.RUNTIME)
26 @Target({ElementType.METHOD})
27 public @interface ActionMethod {
28
29     public static enum ForwardMethod {
30         FORWARD, REDIRECT
31     }
32
33     public static enum HttpMethod {
34         GET, POST, ALL
35     }
36
37     /**
38      * ActionMethod name, url 访问时使用该名称
39     */
40     String name();
```

```

41
42     /**
43      * 调用成功时, 跳转的页面
44      */
45     String successView();
46
47     /**
48      * 发生错误时, 跳转的页面, 如果没有定义, 将直接在浏览器中显示异常信息
49      */
50     String errorView() default "";
51
52     /**
53      * 跳转的方式, 默认为 forward
54      */
55     ForwardMethod forwardMethod() default ForwardMethod.FORWARD;
56
57     /**
58      * 接受的 Http 调用类型
59      */
60     HttpMethod httpMethod() default HttpMethod.ALL;
61
62     /**
63      * 用来组装 HttpRequest 参数的类, 为一个标准的 Java Bean, file name 与 form input name
64      * 对应。
65      *
66      * 可以在 field 上加上 validate annotation 来进行数据校验。
67      *
68      * 支持 file upload, 文件上传的 Field 类型必须为 FileUploaded
69      */
70     Class<? extends Invocation> invocationClass() default Invocation.class;
71 }
72

```

从@ActionMethod 定义可以看出, 在使用时, name、successView 必须要赋值。

两个名字相同而 httpMethod 不同的@ActionMethod 被视为两个不同的 Action 方法。

下面展示了 Petstore 模块的 AccountAction 中, 使用@ActionMethod 描述 doGetSignOn 方法, 指定了 name、successView 和 httpMethod:

```

1  @ActionMethod(name="signon", successView = "signon.vhtml", httpMethod =
ActionMethod.HttpMethod.GET)
2  public void doGetSignon(InvocationContext invocationContext) throws Exception {
3      // don't need do anything, just forward to successView
4  }

```

下面是 Petstore 模块的 AccountAction 中，使用 @ActionMethod 描述 doPostSignOn 方法，指定了更多的属性：

```

1  @ActionMethod(name="signon", successView = "index.vhtml", errorView = "signon.vhtml",
invocationClass = SignonInvocation.class, httpMethod = ActionMethod.HttpMethod.POST)
2  public void doPostSignon(InvocationContext invocationContext) throws Exception {
3      SignonInvocation invocation =
(SignonInvocation) invocationContext.getInvocation();
4
5      Account account =
(Account) loginService.login(invocationContext.getSessionContext(), this,
invocation.getUsername(), invocation.getPassword());
6      if (account == null) {
7          String msg = "Invalid username or password. Signon failed";
8          PageContext pageContext = invocationContext.getPageContext();
9          pageContext.setAttribute("errorMessage", msg);
10         throw new Exception(msg);
11     }
12     else {
13         SessionContext sessionContext = invocationContext.getSessionContext();
14         sessionContext.setAttribute(ACCOUNT_SESSION_KEY, account);
15     }
16 }
```

@ActionMethod 中的 successView/errorView 均使用相对于模块 views 目录的路径，successView/errorView 也可以指向另外一个 ActionMethod，只需要指明目标 Action 方法的相对 url 即可；在没有指定 errorView，当出现异常时，异常将会被抛出；

@ActionMethod 中的 TargetMethod 用来指定页面的跳转方式是采用 Forward 还是 Redirect，默认使用 Forward，而有些情况，比如：为了避免出现 F5 refresh 重复提交问题，则需使用 Redirect 进行跳转。注意 Redirect 情况下，只能使用 Session 才能进行数据传递。

@ActionMethod 可以通过 invocationClass 指定用来封装请求参数的 Invocation 类，不指定则会使用匿名 Invocation 类，Invocation 将在后面讲到。

下面的例子展示了 successView 指向另一个 Action 方法，并使用 REDIRECT 方式跳转，并指定 invocationClass：

```

1  @ActionMethod(successView = "console.jpa.do", targetMethod =
ActionMethod.TargetMethod.REDIRECT, invocationClass = TestConnectionInvocation.class)
2  public void doGetClearCacheConfig(InvocationContext invocationContext) throws
Exception {
3      TestConnectionInvocation invocation =
(TestConnectionInvocation) invocationContext.getInvocation();
```

```
4     String unitName = invocation.getUnitName();
5
6 EntityManagerFactoryBuilderImpl.getEntityManagerFactoryByName(unitName).clearCache();
7 }
```

### ● Action类

JFox 的 MVC Action 类因为可以申明多个 Action 方法，所以 Action 类可以设计为完成一个相关领域对象所有操作。

以 org.jfox.manager.demo.CartsAction 为例，它有多个 Action 方法，用来完成与 Carts 相关的所有动作。下面展示了 CartsAction 的代码：

```
1 package org.jfox.manager.demo;
2
3 import java.util.ArrayList;
4
5 import javax.ejb.EJB;
6
7 import org.jfox.framework.annotation.Service;
8 import org.jfox.mvc.ActionSupport;
9 import org.jfox.mvc.Invocation;
10 import org.jfox.mvc.InvocationContext;
11 import org.jfox.mvc.SessionContext;
12 import org.jfox.mvc.PageContext;
13 import org.jfox.mvc.annotation.ActionMethod;
14
15 /**
16  * carts actions
17 *
18  * @author <a href="mailto:jfox.young@gmail.com">Young Yang</a>
19 */
20 @Service(id="carts")
21 public class CartsAction extends ActionSupport {
22
23     @EJB
24     ICarts carts;
25
26     @ActionMethod(name="view", successView = "demo/carts.fhtml", httpMethod =
27     RequestMethod.HttpMethod.GET)
28     public void doGetView(InvocationContext invocationContext) throws Exception{
29         // do nothing, just forward to successView
30     }
31 }
```

```
31     @ActionMethod(name="submit", successView = "demo/carts.fhtml", invocationClass =
32         CartInvocation.class, httpMethod = ActionMethod.HttpMethod.POST)
33     public void doPostSubmit(InvocationContext invocationContext) throws Exception {
34         CartInvocation invocation = (CartInvocation)invocationContext.getInvocation();
35         SessionContext sessionContext = invocationContext.getSessionContext();
36         ArrayList<String> carts = (ArrayList<String>)sessionContext.getAttribute("carts");
37         if(carts == null) {
38             carts = new ArrayList<String>();
39             sessionContext.setAttribute("carts", carts);
40         }
41         if(invocation.getSubmit().equals("add")) {
42             carts.add(invocation.getItem());
43         }
44         else if(invocation.getSubmit().equals("remove")){
45             carts.remove(invocation.getItem());
46         }
47         PageContext pageContext = invocationContext.getPageContext();
48         pageContext.setAttribute("carts", carts);
49     }
50
51     public static class CartInvocation extends Invocation {
52         private String item;
53         private String submit;
54
55         public String getItem() {
56             return item;
57         }
58
59         public void setItem(String item) {
60             this.item = item;
61         }
62
63         public String getSubmit() {
64             return submit;
65         }
66
67         public void setSubmit(String submit) {
68             this.submit = submit;
69         }
70     }
71
72 }
73 }
```

一个完整的 Action 类应该包含这几部分：

1. @Service 描述，并指明 id(不指定则为 Action 类名的简称)，用来将 Action 部署到 IoC 内核，并且使用 URL 访问时，也会用到 id；
2. 一个或者多个@ActionMethod 描述的方法，并指明相关属性，如：successView, invocationClass 等；
3. 被定义成 public static 的 Invocation 内部类，以及其 Field 上相关的 Validation 描述。关于 Invocation 的更多内容可以参考下面专门关于 Invocation 的章节。

上例中，访问 do getView 的 URL 应该像下面这样：

<http://localhost:8080/jfox/modules/manager/carts.view.do>

URL 中 cart.view.do 解释如下：

carts 是 Action 组件@Service 描述的 id, view 是 do getView 方法名除去 doGet 之后的剩余部分，均使用小写名，.do 是 web.xml 中配置的 Action 访问的后缀名。

### ● 回调方法

在 ActionSupport 中定义了回调方法，以便更好的控制 Action 方法的执行。

以下是 ActionSupport 定义的 3 个回调方法：

```

1  /**
2   * do something before invoke action method
3   *
4   * @param invocationContext invocation context
5   */
6  protected void preAction(InvocationContext invocationContext) {
7
8  }
9
10 /**
11  * do something after invocation action method
12  *
13  * @param invocationContext invocation context
14  */
15 protected void postAction(InvocationContext invocationContext) {
16
17 }
18
19 /**
20  * 在 execute 过程中出现异常时，将回调该方法。
21  * 该方法可以做一些补偿性工作，比如：在 buildInvocation 过程中出现了异常，
22  * 可以通过实现该方法恢复数据，以便能够在 errorView 中显示用户数据
23  *
24  * @param invocationContext invocationContext
25 */

```

```

26     protected void doActionFailed(InvocationContext invocationContext) {
27
28 }
```

preAction 在正式执行 Action 方法之前被调用，而 postAction 在 Action 方法执行之后被调用。

doActionFailed 回调方法在 Action 方法执行失败时被调用，可以在 doActionField 中进行一些数据补偿工作，比如：当用户登录失败时，在跳转到失败页面之前，需要做一些数据恢复的操作，以便跳转到失败页面时能保留已经输入的信息。

在 Petstore 的 AccountAction 中，使用了 doActionField，代码如下：

```

1  /**
2   * 在 doPostEdit 发生异常时，通过该方法设置 PageContext account,
3   * 以便跳转到 errorView 时，可以预设数据
4   *
5   * @param invocationContext invocationContext
6   */
7  protected void doActionFailed(InvocationContext invocationContext) {
8      if (invocationContext.getActionMethod().getName().equals("doPostEdit")) {
9          SessionContext sessionContext = invocationContext.getSessionContext();
10         Account account = (Account) sessionContext.getAttribute(ACCOUNT_SESSION_KEY);
11
12         PageContext pageContext = invocationContext.getPageContext();
13         pageContext.setAttribute("account", account);
14         pageContext.setAttribute("languages", languages);
15
16         List<Category> categories = categoryB0.getCategoryList();
17         pageContext.setAttribute("categories", categories);
18     }
19     else if(invocationContext.getActionMethod().getName().equals("doPostCreate")) {
20         NewAccountInvocation invocation =
(NewAccountInvocation) invocationContext.getInvocation();
21         Account newAccount = EntityFactory.newEntityObject(Account.class);
22         newAccount.setUsername(invocation.getUsername());
23         newAccount.setStatus("OK");
24         newAccount.setPassword(invocation.getPassword());
25         newAccount.setAddress1(invocation.getAddress1());
26         newAccount.setAddress2(invocation.getAddress2());
27         newAccount.setBannerOption(invocation.getBannerOption());
28         newAccount.setCity(invocation.getCity());
29         newAccount.setCountry(invocation.getCountry());
30         newAccount.setEmail(invocation.getEmail());
31         newAccount.setFavouriteCategoryId(invocation.getFavouriteCategoryId());
32         newAccount.setFirstName(invocation.getFirstName());
```

```

33     newAccount.setLanguagePreference(invocation.getLanguagePreference());
34     newAccount.setLastName(invocation.getLastName());
35     newAccount.setListOption(invocation.getListOption());
36     newAccount.setPassword(invocation.getPassword());
37     newAccount.setPhone(invocation.getPhone());
38     newAccount.setState(invocation.getState());
39     newAccount.setZip(invocation.getZip());
40     PageContext pageContext = invocationContext.getPageContext();
41     pageContext.setAttribute("account", newAccount);
42     try {
43         doGetNewAccount(invocationContext);
44     }
45     catch(Exception e) {
46         logger.error("doActionFailed error.", e);
47     }
48 }
49 }
50

```

### 5.3.3 Invocation

在@ActionMethod 描述中，有一个重要的属性 invocationClass，Invocation 是用来实现 Form mapping 的 Java Bean 类，JFox MVC 会自动将 HTML Form 数据映射成 Invocation 对象，Invocation 一般直接在 Action 类中使用 public static class 定义，Invocation 类也可以写成外部类，之所以推荐写成内部类，主要是为了让 Action 类看起来更完整。,

需要注意的是，为了实现自动映射，Invocation 的 Field 名称必须于 Form 的 input field 的名称保持一致！

下面是 AccountAction 定义的用于登录的 SignonInvocation。

```

1  public static class SignonInvocation extends Invocation {
2      @StringValidation(minLength = 4, nullable = false)
3      private String username;
4
5      @StringValidation(minLength = 4, nullable = false)
6      private String password;
7
8      public String getPassword() {
9          return password;
10     }
11
12     public void setPassword(String password) {
13         this.password = password;

```

```

14     }
15
16     public String getUsername() {
17         return username;
18     }
19
20     public void setUsername(String username) {
21         this.username = username;
22     }
23 }
```

可以看出，Invocation 类需要从 org.jfox.mvc.Invocation 继承，同时在 Field 上可以使用 Annotation 进行数据校验。

如果一个 Action 方法没有指定 invocationClass，则会自动构造匿名的 org.jfox.mvc.Invocation 的实例，此时可以使用 Invocation 类中定义的 getAttribute 方法根据 Form 中的 input name 获得用户输入的值。

#### ● Validator

Invocation 的 Filed 上，可以使用各种 Validation Annotation 来对用户的输入数据进行校验。Validation Annotation 的定义在 org.jfox.mvc.validate 包下，目前已经提供以下几种 Validation：

1. @StringValidation
2. @IntegerValidation
3. @LongValidation
4. @FloatValidation
5. @DobuleValidation
6. @EmailValidation

用户还可以根据需要扩展其它的 Annotation。

Invocation 超类中还定义了 validateAll 方法，在 Invocation 的属性都设置完毕之后，进行综合验证，下面的例子展示了注册用户时，使用 validateAll 验证两次输入的密码是否一致的情况。

```

1     public void validateAll() throws ValidateException {
2         //验证密码是否一致
3         if (!getPassword().equals(getRepeatpassword())) {
4             throw new ValidateException("password twice input are different.",
5 "password", getPassword());
6         }
7     }
```

注意： JFox MVC 的 Validator 只进行服务器端验证，如果需要在浏览器端进行验证，则仍然需要手工编写 javascript 代码。

Validator 包括两部分：

1. Validation Annotation, 用来描述在 Invocation Class 的 Field 上;
2. 用来实现验证过程的 Validator 类;

### ■ Validation Annotation

我们以 String Validate 为例, 看一下 StringValidation 的定下:

```

1 package org.jfox.mvc.validate;
2
3 import java.lang.annotation.Retention;
4 import java.lang.annotation.RetentionPolicy;
5 import java.lang.annotation.Target;
6 import java.lang.annotation.ElementType;
7
8 /**
9  * 字符串验证 Annotation
10 * @author <a href="mailto:jfox.young@gmail.com">Yang Yong</a>
11 */
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target(ElementType.FIELD)
14 public @interface StringValidation {
15
16     /**
17      * default String validator class
18      */
19     Class<? extends Validator> validatorClass() default StringValidator.class;
20
21     /**
22      * 最小长度
23      */
24     int minLength() default Integer.MIN_VALUE;
25
26     /**
27      * 最大长度
28      */
29     int maxLength() default Integer.MAX_VALUE;
30
31     /**
32      * 是否可以为空
33      */
34     boolean nullable() default false;
35 }
36

```

Validation Annotation 都需要包含一个 validatorClass 属性, 指定用来执行校验的 Validator 类, 这个方法是 JFox MVC 对于 Validation Annotation 约定的一个方法, 会在运行时通过反射该方法得到 Validator 类, 以完成数据的校验, 所以在自定义的 Validation

Annotation 中，也必须包含该属性。

在@StringValidation 中，validatorClass 的定义如下：

```
class<? extends Validator> validatorClass() default StringValidator.class;
```

那么对于@StringValidation，默认会使用 StringValidator 类进行校验。如果默认的 Validator 类不能满足要求，可以在使用@StringValidation 的时候，指定其它的 Validator 类，如下面的例子，指定了 demo.MyStringValidator 来进行校验。

```
1 @StringValidation(minLength = 4, nullable = false, validatorClass =
"demo.MyStringValidator")
2 private String password;
3
```

### ■ Validator

Validator 用来完成校验的过程，Validator Class 都必须实现 org.jfox.mvc.validate.Validator 接口。

Validator 接口定义如下：

```
1 package org.jfox.mvc.validate;
2
3 import java.lang.annotation.Annotation;
4
5 /**
6 * Validator interface
7 *
8 * @author <a href="mailto:jfox.young@gmail.com">Yang Yong</a>
9 */
10 public interface Validator<T> {
11
12     /**
13      * verify the input according validation
14      * return Object constructed by input if success
15      * @param inputValue input string
16      * @param validation validation annotation
17      * @throws ValidateException if validate failed
18     */
19     public T validate(String inputValue, Annotation validation) throws ValidateException;
20 }
21
```

Validator 中只有一个方法 validate，第一个参数为用户输入的值，第二个参数为 Annotation，通过 Annotation，可以获得校验参数，如果验证失败，应该抛出 ValidateException 异常；如果验证通过，返回正确类型的值，如：IntegerValidator 返回 Integer，DateValidator 返回 Date。

看一下 IntegerValidator 的例子：

```

1 package org.jfox.mvc.validate;
2
3 import java.lang.annotation.Annotation;
4
5 /**
6  * validate integer input
7 *
8  * @author <a href="mailto:jfox.young@gmail.com">Yang Yong</a>
9 */
10 public class IntegerValidator implements Validator<Integer> {
11
12     public Integer validate(String inputValue, Annotation validation) throws
ValidateException {
13         IntegerValidation integerValidation = (IntegerValidation)validation;
14
15         // 整型数据
16         int minValue = integerValidation.minValue();
17         int maxValue = integerValidation.maxValue();
18         try {
19             int intValue = Integer.parseInt(inputValue);
20             if (intValue < minValue || intValue > maxValue) {
21                 throw new ValidateException("The input value " + inputValue + " must between
" + minValue + ", " + maxValue, inputValue);
22             }
23             return intValue;
24         }
25         catch (NumberFormatException e) {
26             throw new ValidateException("Illegal Integer format for input: " + inputValue,
inputValue);
27         }
28     }
29
30 }
```

### ■ 获得 Validate 异常信息

如果用户的输入不正确的时候，Validator.validate() 方法会抛出 ValidateException，你可以获得 exception 的信息可以在页面上输出，获得异常信息的方式如下：

```
$ J_VALIDATE_EXCEPTIONS.get("username").getMessage()
```

\$ J\_VALIDATE\_EXCEPTIONS 是 JFox MVC 中定义的固化变量，实际的类型是一个只读的 Map，通过 get 方法传入 Form 的 input name 即可得到该 input 的 ValidateException 异常，使用 Exception.getMessage() 即可得到异常信息，所以在 validate 方法中抛出有意义的异

常信息是很有必要的。

JFox Petstore 中，用户登录的输入 formHTML 代码如下：

```
1 <form action="account.signon.do" method="POST">
2
3     <table align="center" border="0">
4         <tr>
5             <td colspan="2" align="center">
6                 Please enter your username and password.
7                 <center><b><font color="RED">$!errorMessage </font></b></center>
8             </td>
9         </tr>
10        <tr>
11            <td>Username:</td>
12            <td><input type="text" name="username" value="j2ee"/>
13                <font color="RED">
$!J_VALIDATE_EXCEPTIONS.get("username").getMessage()</font>
14            </td>
15        </tr>
16        <tr>
17            <td>Password:</td>
18            <td><input type="password" name="password" value="j2ee"/>
19                <font
color="RED">$!J_VALIDATE_EXCEPTIONS.get("password").getMessage()</font>
20            </td>
21        </tr>
22        <tr>
23            <td> </td>
24            <td><input type="image" border="0" src="images/button_submit.gif"
name="update"/></td>
25        </tr>
26    </table>
27
28 </form>
```

如果用户名或者密码输入格式有错，便会在页面上用红字打印出来，如下图：

Username:	<input type="text" value="j2ee"/>	<b>input length must between [4,2147483647] !</b>
Password:	<input type="password" value="****"/>	
<b>Submit</b>		

### 5.3.4 InvocationContext

InvocationContext 是 Action 方法需要的唯一的参数，它封装了 Action 调用的上下文资源。通过 InvocationContext，可以得到 PageContext，SessionContext，Invocation，Action Method 以及 ServletConfig 等。

在一个典型的 Action 方法中，对 InvocationContext 会有以下的操作步骤：

1. 通过 InvocationContext.getInvocation() 获得 Invocation 对象，并造型成正确的类型，以获得 Http 请求中的各种参数；
2. 如果需要使用 Session，可以调用 InvocationContext.getSessionContext() 获得 Session 上下文；
3. 最后需要通过 InvocationContext.getPageContext() 获得页面上下文对象，并设置各种属性，以交给模板引擎来渲染页面。

下面的代码来自 Petstore 的 CartsAction：

```

1  @ActionMethod(successView = "Cart.vhtml", invocationClass = CartInvocation.class)
2  public void doGetAddItem(InvocationContext invocationContext) throws Exception {
3      CartInvocation invocation = (CartInvocation) invocationContext.getInvocation();
4
5      SessionContext sessionContext = invocationContext.getSessionContext();
6      Cart cart = (Cart) sessionContext.getAttribute(CART_SESSION_KEY);
7      if (cart == null) {
8          cart = new Cart();
9          sessionContext.setAttribute(CART_SESSION_KEY, cart);
10     }
11     if (!cart.containsItemId(invocation.getWorkingItemId())) {
12         Item item = itemBO.getItem(invocation.getWorkingItemId());
13         cart.addItem(item, itemBO.isItemInStock(invocation.getWorkingItemId()));
14     }
15     else {
16         cart.incrementQuantityByItemId(invocation.getWorkingItemId());
17     }
18
19     PageContext pageContext = invocationContext.getPageContext();
20     pageContext.setAttribute("cart", cart);
21 }
```

#### ● PageContext

PageContext 用来设置页面上下文，其中的数据将被模板引擎(Velocity, Freemarker)用来渲染模板，以正确输出 HTML 页面。

在 Action 方法中，执行完毕之后，一般都需要通过 PageContext.setAttribute() 来设置相关的属性值，以便模板引擎用来渲染页面。如下面打开新建用户页面的例子：

```

1  @ActionMethod(successView = "NewAccountForm.vhtml")
2  public void doGetNewAccount(InvocationContext invocationContext) throws Exception {
```

```

3      // do nothing
4      PageContext pageContext = invocationContext.getPageContext();
5      pageContext.setAttribute("languages", languages);
6
7      List<Category> categories = categoryBO.getCategoryList();
8      pageContext.setAttribute("categories", categories);
9  }

```

MVC 框架已经在 PageContext 中设置了一些通用的数据属性，其中包括一些环境信息和校验异常、业务异常等，这些变量均为大写，且以 J\_ 开头，以避免和应用设置的属性重名而覆盖了应用属性，因为通用属性在。

下面是通用属性列表：

属性名	解释	
J_VALIDATE_EXCEPTION	校验异常，这是一个 Map，Key 为 From input field name，value 为 ValidateException	
J_EXCEPTION	业务异常，即 Action 执行过程中抛出的异常	
J_SESSION_CONTEXT	执行 SessionContext，获得 Session 中存储的数据	
J_PAGE_CONTEXT	PageContext 本身	
J_REQUEST	HttpServletRequest 对象	
J_WEBAPP_CONTEXT_PATH		
J_SERVLET_PATH		

另外，使用 PageContext.setTargetView 可以手动设置 Action 的跳转页面，用于程序控制的页面跳转。

### ● SessionContext

SessionContext 是 JFox MVC 用来存在 Session 数据的上下文对象，它和 InvocationContext 关联在一起，随时可以通过 InvocationContext.getSessionContext() 获得。

SessionContext 使用了 Map<String, Serializable> 来存储 Session 数据，所以存储到 Session 中的数据都应该是可序列化的：

```

1  /**
2   * 使用 Map 存储 Session 数据
3   */
4  private Map<String, Serializable> sessionMap = new HashMap<String, Serializable>();
5

```

SessionContext 提供了以下方法用来操作 Session 数据：

```

1  public void setAttribute(String key, Serializable value) {
2      sessionMap.put(key, value);

```

```

3      }
4
5      public Serializable getAttribute(String key) {
6          return sessionMap.get(key);
7      }
8
9      public boolean containsAttribute(String key) {
10         return sessionMap.containsKey(key);
11     }
12
13     public Serializable removeAttribute(String key) {
14         return sessionMap.remove(key);
15     }
16
17     public String[] getAttributeNames() {
18         return sessionMap.keySet().toArray(new String[sessionMap.size()]);
19     }
20
21     /**
22      * 销毁 SessionContext
23      */
24     public void destroy() {
25         sessionMap.clear();
26         SessionContext disassociateThreadSessionContext();
27         request.getSession().removeAttribute(SESSION_KEY);
28     }
29

```

因为 SessionContext 在初始化的时候，已经和当前的线程进行了关联，所以你还可以通过静态方法 SessionContext.currentThreadSessionContext() 获得与当前线程关联的 SessionContext。

```

1      /**
2      * 得到与当前线程绑定的 SessionContext
3      * @return return null if current thread not associate session context
4      */
5      public static SessionContext currentThreadSessionContext() {
6          SessionContext sessionContext = threadSession.get();
7          if(sessionContext == null) {
8              threadSession.remove();
9          }
10         return sessionContext;
11     }
12

```

## 5. 3. 5 依赖注入

在 Action 中可以，直接使用 @EJB 来注入 EJB，从而简化了 Action 类对 EJB 的访问。

在上面的例子中，在字段定义部分，可以看到如下的代码：

```
23  @EJB
24  ICarts carts;
```

上面说到，Action 是一个 JFox 内核标准，所以自然能够使用@Inject @Constant 来进行注入；但同时，Action 中也可以使用@EJB 来注入 EJB，@Resource 从 JNDI 中注入资源，和在 EJB 中使用完全一样，这大大简化了 Action 对 EJB 容器资源的访问。

可以说，JFox MVC is EJB3 ready！

还是来看 Petstore 中 AccountAction 的例子：

```
1 @Service(id = "account")
2 public class AccountAction extends ActionSupport implements CallbackHandler {
3     @Inject
4     JAASLoginService loginService;
5
6     @EJB
7     AccountBO accountBO;
8
9     @EJB
10    CategoryBO categoryBO;
11
12    ...
13}
```

## 5. 3. 6 文件上传

使用 JFox MVC 实现文件上传是非常容易的，只需要在 Invocation Class 中定义一个类型为 org.jfox.mvc.FileUploaded 字段即可，通过 FileUploaded.getFileName() 可以得到文件名，通过 FileUploaded.getContent() 可以得到文件内容，返回值的类型为 byte 数组。

```
1  public static class UploadInvocation extends Invocation {
2
3      private FileUploaded uploadFile;
4
5      public FileUploaded getUploadFile() {
6          return uploadFile;
7      }
8
9      public void setUploadFile(FileUploaded uploadFile) {
10         this.uploadFile = uploadFile;
11     }
12 }
```

```
11      }
12  }
```

上传文件的大小限制在 web.xml 中配置。

### 5.3.7 使用 JAAS 登录

JFox Java EE 容器中已经讲述了 EJB Security，再来看一下 JFox MVC 中如果使用 JAAS 进行登录。

为了使用 JAAS 登录，你需要对你的 Action 做如下动作：

1. 使用@.Inject 注入 JAASLoginService
2. 让 Action 实现 javax.security.auth.callback.CallbackHandler 接口
3. 在 handle 方法中进行登录，并设置 setCallbackObject, setPrincipalName, setRole
4. 使用 LoginService 的 login 方法登录

以下的代码摘自 Petstore 模块的 org.jfox.petstore.action.AccountAction。

1. 实现 JAAS CallbackHandler 接口，并注入登录服务 JAASLoginService

```
1 @Service(id = "account")
2 public class AccountAction extends ActionSupport implements CallbackHandler {
3
4     @Inject
5     JAASLoginService loginService;
6
```

2. 在 handle 实现登录过程，Action 方法中调用 JAASLoginService.login 方法

```
1     @ActionMethod(successView = "index.xhtml", errorView = "signon.xhtml", invocationClass
= SignonInvocation.class)
2     public void doPostSignon(InvocationContext invocationContext) throws Exception {
3         SignonInvocation invocation =
(SignonInvocation)invocationContext.getInvocation();
4
5         Account account =
(Account)loginService.login(invocationContext.getSessionContext(), this,
invocation.getUsername(), invocation.getPassword());
6         if (account == null) {
7             String msg = "Invalid username or password. Signon failed";
8             PageContext pageContext = invocationContext.getPageContext();
9             pageContext.setAttribute("errorMessage", msg);
10            throw new Exception(msg);
11        }
12    else {
```

```

13         SessionContext sessionContext = invocationContext.getSessionContext();
14         sessionContext.setAttribute(ACCOUNT_SESSION_KEY, account);
15     }
16 }
17
18 //JAAS CallbackHandler method
19 public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
20     JAASLoginRequestCallback requestCallback =
(JAASLoginRequestCallback) callbacks[0];
21     JAASLoginResponseCallback responseCallback =
(JAASLoginResponseCallback) callbacks[1];
22
23     // first parameter is username
24     String username = requestCallback.getParams().get(0);
25     // second parameter is password
26     String password = requestCallback.getParams().get(1);
27
28     Account account = accountB0.getAccount(username, password);
29
30     // set callback object, will return by LoginService.login
31     responseCallback.setCallbackObject(account);
32     // set principal name
33     responseCallback.setPrincipalName(username);
34     // set role
35     responseCallback.setRole(username);
36 }
```

注意：handle 方法中，需要设置 JAASLoginResponseCallback 的各项属性。

3. AccountAction.doPostSignIn 方法调用 JAASLoginService.login 登录，将 this 作为 CallbackHandler：

```
1 Account account = (Account) loginService.login(invocationContext.getSessionContext(), this,
invocation.getUsername(), invocation.getPassword())
```

### 5.3.8 防止重复提交

JFox MVC 内置了防止重复提交的控制。为了使用该特性，只需要在提交的 Form 中，加入如下一行即可：

```
<input type="hidden" name="request_token" value="${_REQUEST_TOKEN}">
```

JFox 会在生成页面时，生成 REQUEST TOKEN，在 Form 提交时，对 TOKEN 进行判断，发现重复提交，则会抛出异常。

### 5. 3. 9 访问 Action

最后，再总结一下如何通过 URL 访问你的 Action 以及 view。

比如 manager 模块，访问静态文件的 url 为：

http://localhost/jfox/modules/manager/index.html，或者

http://localhost/jfox/modules/manager/sub/index.html；

访问 Action 的 url 为： /jfox3/modules/manager/example.view.do

example 是 Action 部署的 id, view 是 Action 中的 Action 方法，如果是 Get 访问则是 doGetView 方法，如果是 Post 访问，则是 doPostView。

### 5. 4 JFox Module

现在来讲解 JFox 模块的概念，之所以把模块单独作为一个章节来将，是因为模块既与 JFox 内核相关，也与 MVC 框架相关。

模块的特性表现在首先模块具有一定的自包含性，能够比较独立的开发、运行和维护，同时模块又可以与其它模块组合并交互以实现复杂的功能，并且可以不断的增加模块来完善整个系统，所以模块化在当今的系统设计中越来越被重视。但同时，由于模块的自包含性和可交互性这一对矛盾，使得设计时，首先需要将一个系统按照业务需求以及依赖关系合理的划分成多个模块，比如：模块之间如果有循环依赖的话，就需要考虑一下模块的划分是否合理了；其次还需要有一个平台来支持。JFox 提供了良好的模块化支持能力。

JFox 的模块分为两种类型，一种只提供服务，没有界面；而另一种则拥有界面。没有界面的模块只需要 JFox 内核就可以支持，而有界面的模块则需要 JFox MVC 框架支持。

如： JFox 应用服务器的内置组件都处于默认的名为 \_\_SYSTEM\_MODULE\_\_ 的模块中， JFox 控制台 (manager) 以及 Petstore 则被实现成具有界面的模块，部署在 WEB-INF/MODULES 下。

如果你正在考虑基于 JFox 实现一个 B/S 结构的简单应用，你也许考虑不需要模块化支持，那么直接使用 JFox 应用服务器所在的 Web Application，并且可以使用你熟悉的 MVC 框架比如： Structs, Webwork, Tapestry 等；而如果你正在开发的是一个较大的应用，并且已经有比较清晰的业务模块的划分，而且期望将来能够按模块进行开发、部署、维护，那么 JFox 提供的模块化功能将刚好能够满足你的要求，你可以将你的应用按业务模块分解成 JFox 的一个一个的模块，并按模块组织你的开发团队，如此将能够获得模块化带来的诸多好处。

实际上，即使你的应用不需要模块，我们仍建议你将其设计成只有一个模块的应用，以 Module 的形式来部署。

## 5. 4. 1 模块结构

模块主要有这几部分组成：配置文件，Java 类和 jar，views（包括模板和其它 Web 资源），以及其它目录和文件。模块一般以独立目录的形式部署到 JFox 应用服务器中，默认的发布目录为：WEB-INF/MODULES。

对于模块的结构，是有一定要求的。

目录名	作用	备注
conf	模块配置文件目录	该目录会加入到模块的 ClassLoader，以便可以用 ClassLoader.getResource 获得文件
conf/module.xml	模块配置文件（见安装模块的配置一节）	如果不存在，则 JFox 会使用默认配置进行部署
src	模块的源码路径	运行时不需要
views	存放 template 以及其它可以直接 Http 访问的资源	可以存在子目录，如果在 view 文件里使用相对路径，均指相对与该路径，而不是相对于文件所在的路径
lib WEB-INF/lib	该模块需要的 jar 文件	每个模块的 jar 由不同的类加载器加载，是完全隔离的，即使有相同的也不会发生冲突；支持多层目录
classes WEB-INF/classes	模块源码的编译输出路径	也可打成 jar 包之后放在 lib 目录下

每个模块都拥有完善的项目结构，所以开发时，可以作为独立项目来开发，而模块之间依赖关系，也上升为项目之间的依赖关系；这将可以避免多个模块作为同一个项目开发时，因为模块之间的依赖关系得不到控制，导致最后模块界限无法划分的情况。

## 5. 4. 2 模块类路径

每个模块都由独立的类加载器加载，类加载器的加载路径为：conf, classes, WEB-INF/classes, lib, WEB-INF/lib。对于 lib 和 WEB-INF/lib 目录，类加载器会加载其子目录下的 jar 文件，所以，在开发和部署的时候，可以方便的将不同的 jar 文件放在不同的二级目录中。

## 5. 4. 3 模块的配置

可以使用 module.xml 来为模块进行配置，主要包括模块的名称、描述、加载优先级以及所引用的模块列表，也可以省略 module.xml 文件，那么加载使用默认模块的配置。

如 manager 模块的 module.xml 配置如下：

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <module>
3   <name>Manager</name>
4   <description>JFox manager module</description>

```

```
5 <priority>1</priority>
6 <!--
7 <ref-modules>Module_1,Module_2,Module_3,Module_4</ref-modules>
8 -->
9 </module>
```

在 WEB-INF/MODULES 下的模块，会按照 priority 值从小到大的顺序进行加载。

<ref-modules>以逗号为分隔列出该模块所有依赖的模块，该模块里的组件只能依赖这里列出的模块中的@Exported 的接口。

## 5. 4. 4 模块的部署

模块开发完成之后，需要部署，JFox 追求开发结构和部署结构一致的理念，所以开发完成之后，只需要将模块目录 copy 至默认的 WEB-INF/MODULES 下，即可在 JFox 应用服务器启动时被加载。

如果你不希望将源码一起部署，则尽管删除就是。你也可以使用 ant 等工作将模块打包成 zip 文件，并拷贝到 WEB-INF/MODULES 下，JFox 启动时会将其解压并加载。

部署之后，即可以通过浏览器访问模块中的 Action 或者静态文件了，详见 JFox MVC 框架中的访问 Action 章节；或者编写 Web Service 客户端访问模块中发布的 Web Service。

# 6 管理控制台

JFox 已经内置了 manager，用于提供管理控制台，你可以通过它查看服务器的各项参数。

管理控制台共提供了 System Information、JNDI View、EJB Container、JPA Container、Modules 五个方面的功能。

## 6. 1 System Information

查看各种系统信息，包括 JFox 版本、Web 容器版本、JVM 信息等。

The screenshot shows the JFox Management Console interface in Mozilla Firefox. The title bar reads "JFox Management Console - Mozilla Firefox". The menu bar includes File, Edit, View, History, Bookmarks, Tools, and Help. The toolbar has standard icons for back, forward, search, and refresh. The address bar shows "http://localhost:8080/jfox3/modules/manager/con". The main content area features a logo of a pig's head and the text "The JFox Project J2EE Application Server Implementation Project". Below this is a yellow "Menu" bar with tabs: System Information (selected), JNDI View, EJB Container, JPA Container, and Modules. The main content area is titled "System Information" and contains a table titled "Information" with columns: JFox Version, Web Container, JVM Version, JVM Vendor, OS Name, OS Version, OS Architecture, and Memory(Max/Used). The table data is as follows:

JFox Version	Web Container	JVM Version	JVM Vendor	OS Name	OS Version	OS Architecture	Memory(Max/Used)
3.0	Apache Tomcat/5.5.20	1.5.0_11	Sun Microsystems Inc.	Windows 2003	5.2	x86	63(M) /13(M)

At the bottom of the content area, it says "Copyright © JFox, 2002-2007". The status bar at the bottom right shows "Done" and the system tray.

## 6. 2 JNDI View

查看 JNDI 信息，这里列出了包括 Transaction Manager、DataSource、JMS Connection Factory 和 EJB 所有的注册到 JNDI 中的对象。

The screenshot shows the JFox Management Console interface in Mozilla Firefox. The title bar reads "JFox Management Console - Mozilla Firefox". The menu bar includes File, Edit, View, History, Bookmarks, Tools, and Help. The toolbar has standard icons for back, forward, search, and refresh. The address bar shows "http://localhost:8080/jfox/modules/manager/console.jndi.do". The main content area features a logo of a pig's head and the text "The JFox Project J2EE Application Server Implementation Project". Below this is a yellow "Menu" bar with tabs: System Information, JNDI View (selected), EJB Container, JPA Container, and Modules. The main content area is titled "JNDI Bindings Manager" and contains a table titled "JNDI Bindings (43)" with columns: No., Naming, and Object. The table data is as follows:

No.	Naming	Object
1	AccountBOImpl/local	\$ejb_proxy_stub[ejbid=AccountBOImpl,interface=[org.jfox.petstore.bo.AccountBO]]
2	AccountBOImpl/remote	\$ejb_proxy_stub[ejbid=AccountBOImpl,interface=[jfox.test.ejbcomponent.bo.AccountBO]]
3	AccountDAOImpl/local	\$ejb_proxy_stub[ejbid=AccountDAOImpl,interface=[org.jfox.petstore.dao.AccountDAO, org.jfox.entity.dao.DataAccessObject]]
4	AccountDAOImpl/remote	\$ejb_proxy_stub[ejbid=AccountDAOImpl,interface=[jfox.test.ejbcomponent.dao.AccountDAO, org.jfox.entity.dao.DataAccessObject]]
5	AddressDAO/remote	\$ejb_proxy_stub[ejbid=AddressDAO,interface=[org.jfox.entity.dao.DataAccessObject]]
6	CartsImpl/remote	\$ejb_proxy_stub[ejbid=CartsImpl,interface=[org.jfox.manager.demo.ICarts]]
7	CategoryBOImpl/local	\$ejb_proxy_stub[ejbid=CategoryBOImpl,interface=[org.jfox.petstore.bo.CategoryBO]]
8	CategoryDAOImpl/local	\$ejb_proxy_stub[ejbid=CategoryDAOImpl,interface=[org.jfox.petstore.dao.CategoryDAO, org.jfox.entity.dao.DataAccessObject]]

At the bottom of the content area, it says "Copyright © 2007 JFox™ www.jfox.org.cn". The status bar at the bottom right shows "Done" and the system tray.

## 6. 3 EJB Container

这里列出了所有的部署的 EJB 的详细信息，包括 EJB 名称、类型、接口类、实现类，所在模块等。

The screenshot shows the JFox Management Console interface in Mozilla Firefox. The title bar reads "JFox Management Console - Mozilla Firefox". The address bar shows the URL "http://localhost:8080/jfox/modules/manager/console.container.do". The main content area displays the "EJB Container Manager" section. It includes a "Transaction Manager" configuration table with a row for "Transaction Timeout (seconds)" set to 60. Below this is a table titled "EJBs (29)" listing three EJBs:

No.	EJB name Mapped Names	Bean class Interfaces	EJB type WebService	Module
1	CartsImpl CartsImpl/remote	org.jfox.manager.demo.CartsImpl org.jfox.manager.demo.JCarts	Stateful false	Manager
2	AccountBOImpl AccountBOImpl/local	org.jfox.petstore.bo.AccountBOImpl org.jfox.petstore.bo.AccountBO	Stateful false	Petstore
3	AccountDAOImpl AccountDAOImpl/local	org.jfox.petstore.dao.AccountDAOImpl org.jfox.petstore.dao.AccountDAO org.jfox.entity.dao.DataAccessObject	Stateful false	Petstore

## 6. 4 JPA Container

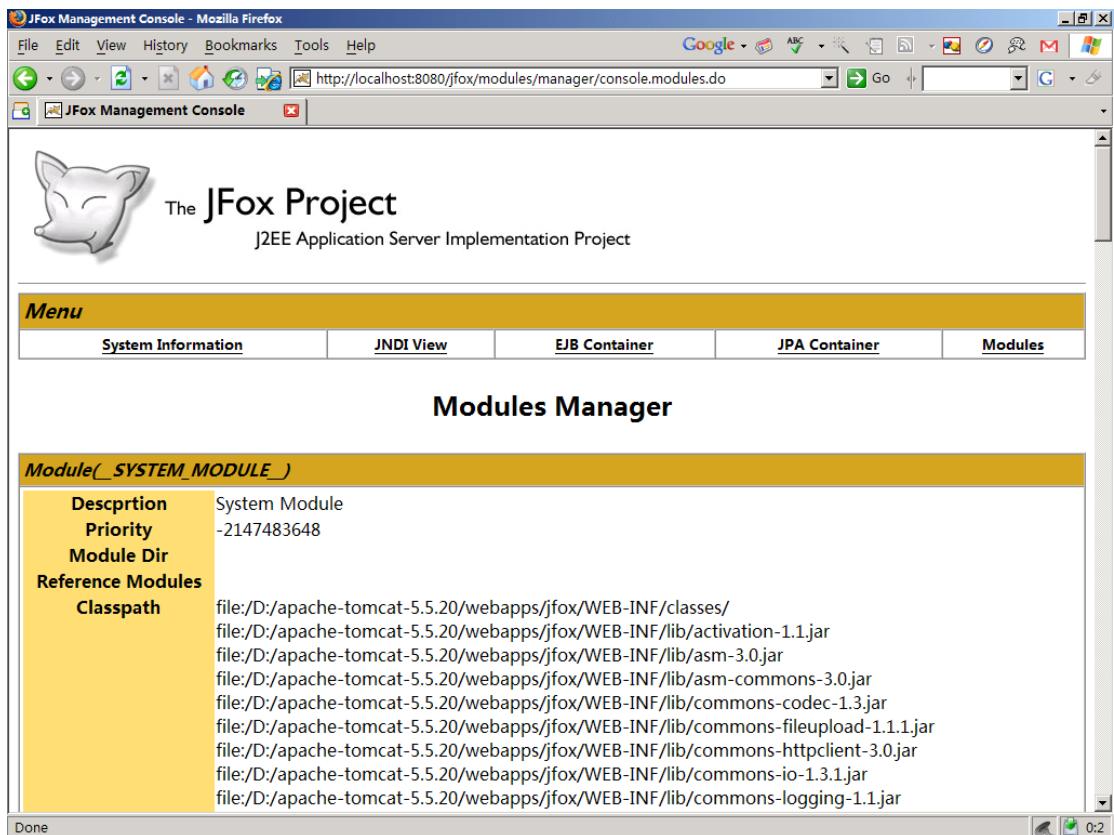
这里列出与 JPA 相关的 Persistence Units、Caches、Named Queries，并且可以测试数据源是否能够连接，以及手动销毁一个 Persistence Unit 的 Cache。

The screenshot shows the JFox Management Console interface in Mozilla Firefox. The title bar reads "JFox Management Console - Mozilla Firefox". The address bar shows the URL "http://localhost:8080/jfox/modules/manager/console.jpa.do". The main content area displays the "JPA Container Manager" page. At the top left is a cartoon fox logo with the text "The JFox Project" and "J2EE Application Server Implementation Project". Below this is a menu bar with tabs: "System Information", "JNDI View", "EJB Container", "JPA Container", and "Modules". The "JPA Container" tab is selected. The main content area is titled "Persistence Units (3)". A table lists three persistence units:

Unit name	JTA Data Source	Driver URL Username	Connection Pool	Cache Config	Command
JPetstoreMysqlDS	java:/JPetstoreMysqlDS	com.mysql.jdbc.Driver jdbc:mysql://localhost:3306/jpetstore root	<b>minSize: 1</b> <b>mxnSize: 200</b> <b>lifeTime: 1800000 ms</b> <b>sleepTime: 60000 ms</b> <b>deadLockRetryWait: 2000 ms</b> <b>deadLockMaxWait: 60000 ms</b> <b>checkLevelObject: 4</b>	<b>Name: JPetstoreMysqlDS</b> <b>Algorithm: LRU</b> <b>Max Size: 1000</b> <b>Max Memory Size: 100000000 bytes</b> <b>TTL: 600000 ms</b> <b>Max Idle Time: 300000 ms</b>	<a href="#">Test Connection</a> <a href="#">Clear Cache</a>
			<b>minSize: 2</b> <b>mxnSize: 50</b>		

## 6.5 Modules

这里可以查看已成功部署至 JFox 内核的模块的详细信息，包括 Classpath 以及模块中已经部署的组件。



## 6.6 查看 Web Service

如果要查看已经发布的 Web Service，可以访问

<http://localhost:8080/jfox/webservice>

## 7 安装 JFox Petstore 模块

JPetstore 是一个在线宠物商店，一般用来演示 J2EE Web 应用。JFox Petstore 基于 iBatis JPetstore 改造，针对 JFox 的 EJB3、JPA 以及 MVC 进行了重写，该模块对于如何基于 JFox 来开发应用具有一定的指导意义。

接下来介绍如何正确的安装和配置 JFox Petstore。

### 7.1 环境准备

首先请确认已经正确的安装了 JFox 3 和 MySQL(建议 5.0 以上版本)，如何安装 JFox 3 请参考本文 JFox 安装与配置一章，MySQL 的安装请参考相关文档；然后从 JFox 项目网站 (<http://code.google.com/p/jfox>) 下载 Petstore 模块(文件名 petstore.zip)，并拷贝到 JFox 模块发布目录(默认为 WEB-INF/MODULES)。注：也可以解压后，以目录的形式拷贝到发布目录下(默认为 WEB-INF/MODULES)，目录名为 petstore。

## 7.2 配置 Petstore 数据源

打开 JFox JPA 配置文件，WEB-INF/classes/META-INF/persistence.xml，添加名称为的JPetstoreMysqlDS 的 persistence-unit，类似如下：

```

1  <persistence-unit name="JPetstoreMysqlDS">
2      <jta-data-source>java:/JPetstoreMysqlDS</jta-data-source>
3      <properties>
4          <property name="driver" value="com.mysql.jdbc.Driver"/>
5          <property name="url" value="jdbc:mysql://localhost:3306/jpetstore"/>
6          <property name="username" value="root"/>
7          <property name="password" value="root"/>
8          <property name="minSize" value="1"/> <!-- min pool size -->
9          <property name="maxSize" value="200"/> <!-- max pool size -->
10         <property name="lifeTime" value="1800000"/> <!-- 3h, connection max idle time,
in ms -->
11         <property name="sleepTime" value="600000"/> <!-- PoolKeeper sleep time, in ms
-->
12         <property name="deadLockRetryWait" value="2000"/> <!-- retry time if no free
connection, in ms -->
13         <property name="deadLockMaxWait" value="60000"/> <!-- max wait time if no free
connection, in ms -->
14         <property name="checkLevelObject" value="4"/> <!-- check connection closed -->
15         <property name="cache.algorithm" value="LRU"/> <!-- algorithm for "default"
cache category, LRU, LFU, FIFO-->
16         <property name="cache.ttl" value="600000"/> <!-- ttl for "default", in ms-->
17         <property name="cache.maxIdleTime" value="300000"/> <!-- maxidletime for
"default", in ms-->
18         <property name="cache.maxSize" value="1000"/> <!-- max size for "default" -->
19         <property name="cache.maxMemorySize" value="100000000"/> <!-- max memory size
for "default", in bytes-->
20     </properties>
21 </persistence-unit>
```

应该保证用于创建数据库连接的 driver、url、username、password 是正确的。

## 7.3 导入初始化数据

注意：JFox Petstore 默认使用的数据库名为 jpetstore，请确认当前 mysql 中没有该库，因为导入数据的时候会删除以存在的 jpetstore 库而造成你的数据丢失。

用 于 初 始 化 MySQL 的 脚 本 位 于 Petstore 模 块 下 的 /db/mysql/jfox-petstore-backup.sql 中，使用你熟悉的工具或者命令行将其导入，建议你使用 SQLyog (<http://www.webyog.com/>)，一个不错的开源 MySQL GUI Manager Tool。

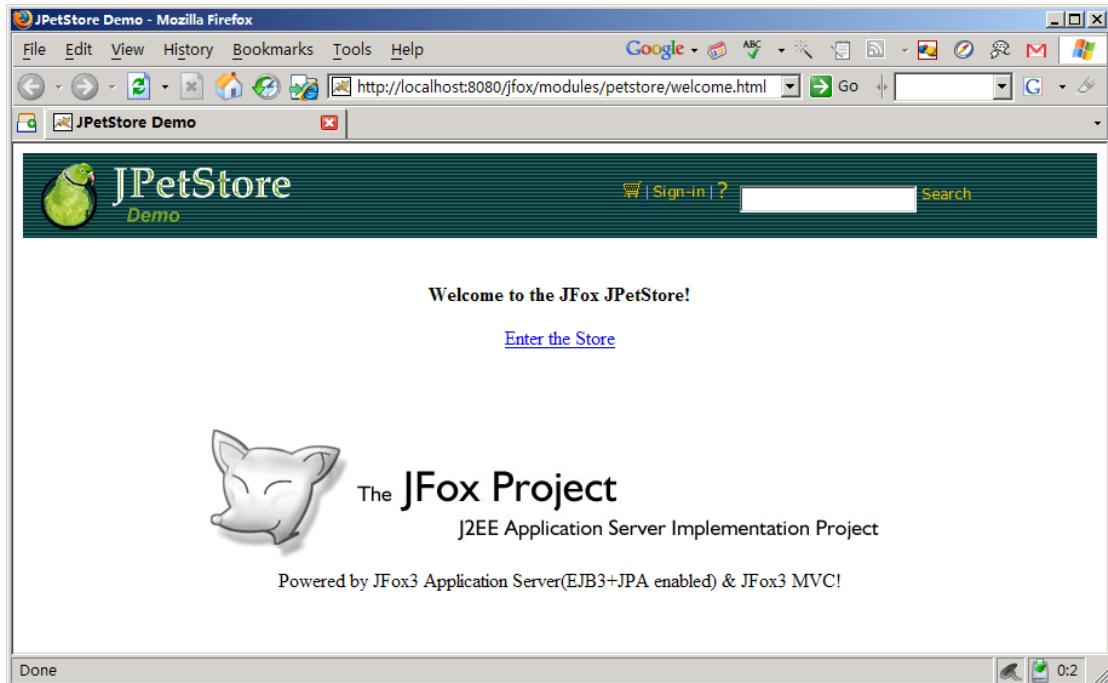
## 7.4 访问 Petstore

接下来，启动 Tomcat，正常的话，可以看到类似如下的日志输出：

```
17:29:08,532      INFO      [Framework]      Module:      Petstore      loaded,      from  
D:\apache-tomcat-5.5.20\webapps\jfox3\WEB-INF\MODULES\petstore
```

然后，打开浏览器，访问

<http://localhost:8080/jfox3/modules/petstore/welcome.html>，  
即可以看到 Petstore 的欢迎页面了，恭喜！



< 全文完 >

2007-10-08