

# JdonFramework 使用开发指南

板桥里人(banq J 道 <http://www.jdon.com>)

English:<http://www.jdon.org>

6.5 版本： 2011 年 10 月

JdonFramework 下载地址: <http://www.jdon.com/jdonframework/download.html>

技术支持论坛: <http://www.jdon.com/jivejdon/61>



## Jdon 框架安装说明

在 Jdon 框架源码包中的 dist 目录下，有下列几个包，版本不同可能不相同，以实际 dist 目录下包为主：

jdonFramework.jar	Jdon 框架核心包	必须
aopalliance.jar	AOPAlliance 包	必须
jdom.jar	读取 XML 的 JDOM 包	必须
picocontainer-1.2.jar	Picocontainer 包	必须
commons-pool-1.2.jar cglib.jar disruptor-2.0.2.jar ehcache-1.6.2.jar scannotation-1.0.2.jar	辅助包	必须

由于 JF 持续更新中，上述包不断变化，以实际为主。

6.5 版本以后，为突出 Jdon 框架 DDD 领域驱动和 DCI 框架特点，将和具体技术如 Struts 或 JDBC 模板或 Hibernate 等功能打包在源码 JdonAccessory 目录中：

如果项目中涉及到表现层和持久层开发，都需要 JdonAccessory 目录中的 jar 包。

## 在 Tomcat 中安装

Jdon 框架在 Tomcat 下安装主要问题是 log4j 问题，下面是安装步骤：

注意：可以将所有包放在项目的 WEB-INF/lib 下，下面方法是一次性安装方式：

编辑项目目录 WEB-INF/classes/log4j.properties：

```
log4j.rootLogger=DEBUG, R, A1

# Configuration for standard output ("catalina.out").
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout

# Print the date in ISO 8601 format
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c - %m%n

# Configuration for a rolling log file ("tomcat.log").
log4j.appender.R=org.apache.log4j.DailyRollingFileAppender
log4j.appender.R.DatePattern='yyyy-MM-dd

# Edit the next line to point to your logs directory.
# The last part of the name is the log file name.
```

```
log4j.appender.R.File=${catalina.home}/logs/jdon.log
log4j.appender.R.layout=org.apache.log4j.PatternLayout

# Print the date in ISO 8601 format
log4j.appender.R.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %5p [%F:%L]
%c{8}%.%M() - %m%n
```

主要第一行:

```
log4j.rootLogger=DEBUG, R, A1
```

表示当前日志输出有两个, 一个 R, 代表文件;另外一个 A1, 代表屏幕。

```
log4j.appender.R.File=${catalina.home}/logs/jdon.log
```

定义了 R 的文件所在位置, 在你的 Tomcat 的 logs 目录下的 jdon.log 中。

配置 Jdon 框架运行过程输出, 在 log4j.properties 中下面一行:

```
log4j.logger.com.jdon=DEBUG
```

该配置将会显示 Jdon 框架的主要运行信息, 如果你要关闭, 只要更改如下:

```
log4j.logger.com.jdon=ERROR
```

重新启动 Tomcat, 这时可以从 tomcat.log 看到输出记录。

## 在 JBoss 中安装

安装步骤:

1. 确保已经安装 J2SE 1.4 以上版本, 然后设置操作系统的环境变量 JAVA\_HOME=你的 J2SE 目录
  2. 下载 JBoss 3.X/JBoss 4.x
  3. 安装 Jdon 框架驱动包: 将 Jdon 框架源码包中的 dist 目录下除 log4j.jar 和 src 或 classes 目录下 log4j.properties 以外的包拷贝到 jboss/server/default/lib 目录下。
  4. 安装 struts 驱动包, 下载 struts 1.2, 将 jar 包拷贝到 jboss/server/default/lib。
- 或者使用 Jdon 框架例程 samples 中 SimpleJdonFrameworkTest 项目的 lib 目录。将该目录下 jar 包拷贝到 jboss/server/default/lib

对于具体 J2EE 应用系统, 需要配置 Jboss 的数据库连接池 JNDI:

- 1.配置 JBoss 的数据库连接:

将数据库驱动包如 MYSQL 的 mysql-connector-java-3.0.14-production-bin.jar 或 Oracle 的 class12.jar 拷贝到 jboss/server/default/lib。

2. 选择 JBoss 的数据库 JNDI 定义文件:

在 jboss 的 docs 目录下寻找你的数据库的配置文件, 如果是 MySQL, 则是 mysql-ds.xml; 如果是 Oracle; 则是 oracle-ds.xml。下面以 MySQL 为例子。

将 mysql-ds.xml 拷贝到 jboss/server/default/deploy 目录下。

3. 修改配置数据库定义文件:

打开 jboss/server/default/deploy/mysql-ds.xml, 如下:

```
<datasources>
<local-tx-datasource>
  <jndi-name>DefaultDS</jndi-name> <!--JNDI 名称 应用程序中使用 java:/DefaultDS 调用 -->
  <connection-url>jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=UTF-8
```

```

</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>root</user-name><!-- MySQL 数据库访问用户和密码，缺省是 root -->
    <password></password>
</local-tx-datasource>
</datasources>

```

#### 4. 启动 JBoss

打开 jboss/server/default/log/server.log 如果没有错误，一切 OK，一般可能是数据库连接错误，检查 mysql-ds.xml 配置，查取相关资料，看懂每行意义。

至此，可以将基于 Jdon 框架开发的 J2EE 应用程序部署到 JBoss 中。

一般是将 \*.ear 或 \*.war 拷贝到 jboss/server/default/deploy 目录下即可。

日志输出：

当运行具体应用系统时，打开 jboss/server/default/log/server.log 时，会看到很多 Jdon 框架本身的输出，如果你觉得非常混乱，可以关闭这些输出，在 jboss/server/default/conf/log4j.xml 中加入如下配置行：

```

<category name="com.jdon.aop">
    <priority value="ERROR"/>
</category>
<category name="com.jdon.container">
    <priority value="ERROR"/>
</category>
<category name="com.jdon.model">
    <priority value="ERROR"/>
</category>
<category name="com.jdon.controller">
    <priority value="ERROR"/>
</category>
<category name="com.jdon.security">
    <priority value="ERROR"/>
</category>
<category name="com.jdon.bussinessproxy">
    <priority value="ERROR"/>
</category>
<category name="com.jdon.strutsutil">
    <priority value="ERROR"/>
</category>

```

## 在 Weblogic 等服务器中安装

只要将 Jdon 框架包和 struts 1.2 包安装到服务器的库目录下即可，或者配置在系统的 classpath 中即可。如果你的服务器没有 log4j 包，那么还需要 log4j.jar，并将 log4j.properties 放置在系统 classpath 中。

1. 将 struts 和 JdonFramework 所有驱动包拷贝到 Weblogic 的 common/lib 目录下。

2. 在 weblogic 的启动文件中加入如下命令：将 jar 包加入系统的 classpath。

```
set CLASSPATH=%CLASSPATH%;%WL_HOME%\common\lib\log4j.jar;
%WL_HOME%\common\lib\mysql-connector-java-3.0.14-production-bin.jar;
set CLASSPATH=%CLASSPATH%;%WL_HOME%\common\lib\jdonFramework.jar;
%WL_HOME%\common\lib\jdom.jar;%WL_HOME%\common\lib\commons-pool-1.2.jar;
%WL_HOME%\common\lib\aopalliance.jar;%WL_HOME%\common\lib\picocontainer-1.1.jar
set CLASSPATH=%CLASSPATH%;%WL_HOME%\common\lib\struts.jar;
%WL_HOME%\common\lib\jakarta-oro.jar;%WL_HOME%\common\lib\commons-validator.jar;
%WL_HOME%\common\lib\antlr.jar;%WL_HOME%\common\lib\commons-beanutils.jar;
%WL_HOME%\common\lib\commons-collections.jar;%WL_HOME%\common\lib\commons-digester.jar
```

r

3. 在具体 Web 项目打包部署时，需要将 log4j.properties 加到 WEB-INF/classes 目录下，更改 log4j.properties 中配置，使之日志输出到你指定一个文件中，注意：当时部署 log4j 日志不会激活 log4j，必须重新启动 Weblogic 即可（项目必须在 Weblogic 中）。

## 如何判断安装启动成功

当启动 Jdon 框架任何一个应用程序，后台日志出现：

```
<===== Jdon Framework started successfully! =====>
```

表示你的系统配置和启动 Jdon 框架成功。

Jdon 框架自从 6.2 版本以后，通过其 log.xml 配置关闭上述显示，如果部署时没有出现任何 error 错误认为是正常启动。log.xml 内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <log>
    <level>6</level><!-- set 6 will get high performance set 1 to debug -->
    <log4j>true</log4j>
  </log>
</config>
```

# 快速入门 ABC

## 一步到位

如果使用 Jdon 框架 6.0 以上，可以直接使用 Jdon 框架的 Annotation，写好代码即可部署运行。

注意：本案例不是 DDD 领域事件或 DCI 架构：

举例如下：

在 HelloServiceImpl 加入注解：

```
@Service("helloService")
public class HelloServiceImpl implements HelloService {
    private UserRepository userRepository;

    public HelloServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public String hello(String name) {
        User user = userRepository.findUser(name);
        System.out.print("call ok");
        return "Hello, " + user.getName();
    }
}
```

HelloServiceImpl 中并没有对接口 UserRepository 实例化，只需在接口 UserRepository 的子类中加入注解 @Component，如下：

```
@Component
public class UserRepositoryInMEM implements UserRepository {

}
```

如果在 Servlet 或 JSP 等客户端调用上述代码，如下：

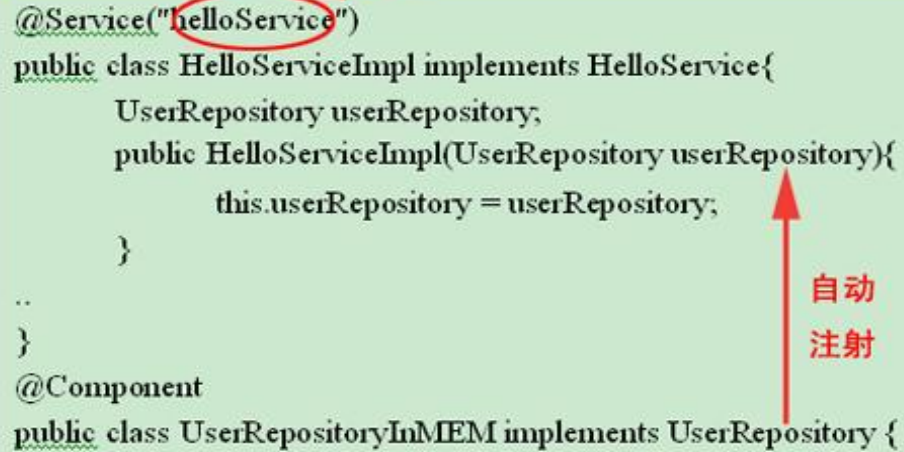
```
HelloService helloService = (HelloService) WebAppUtil.getService("helloService", request);
String result = helloService.hello(myname);
输出：Hello XXXX
```

原理图：

客户端调用代码:

```
HelloService helloService = (HelloService) WebAppUtil.getService("helloService");  
String result = helloService.hello(myname);
```

```
@Service("helloService")  
public class HelloServiceImpl implements HelloService{  
    UserRepository userRepository;  
    public HelloServiceImpl(UserRepository userRepository){  
        this.userRepository = userRepository;  
    }  
    ..  
}  
@Component  
public class UserRepositoryInMEM implements UserRepository {
```



客户端代码中也没有将接口 UserRepository 子类 UserRepositoryInMEM 实例化创建后，赋给 HelloServiceImpl，这一切都是由 Jdon 框架悄悄在背后实现了。

具体全部代码见 JdonFramework 源码包目录下 examples/testWeb 项目。

更详细的 Annotation 注解使用见 [第 6 版更新说明](#)。

关于 AOP 拦截 Annotation 使用见 [6.2 版本更新说明](#)

## 启动 Jdon 框架

如果我们没有使用 XML 配置，全部使用 Annotation，就无需启动 Jdon 框架，直接将应用部署到 Web 容器就可以运行。

第一：直接启动：

如果你全部使用 annotation，没有 jdonframework.xml，直接部署就启动，无需特别配置。

如果你需要在应用停止部署时做一些内存数据保存的工作，那么可以实现 com.jdon.container.pico.Startable 接口，在其 start 和 stop 方法中进行初始化或关闭的工作，然后在 web.xml 中配置：

```
<listener>  
    <listener-class>com.jdon.container.startup.ServletContainerListener</listener-class>  
</listener>
```

第二：jdonframework.xml 的 web.xml 配置方式：

如果你不使用 Struts，但是有 jdonframework.xml，可以通过 web.xml 下列配置来启动 Jdon 框架。

```

<context-param>
    <param-name> modelMapping-config </param-name>
    <param-value> jdonframework.xml </param-value>
</context-param>
.....
<listener>
    <listener-class>com.jdon.container.startup.ServletContainerListener</listener-class>
</listener>

```

第三：结合 struts 配置方式(需要 struts 基础知识):

在 struts-config.xml (或其它 struts 模块配置文件如 struts-config-admin.xml 等等) 中配置 Plugin 实现子类:

```

<plug-in className="com.jdon.strutsutil.InitPlugIn">
    <set-property property="modelMapping-config" value="jdonframework.xml" />
</plug-in>

```

第四：无 Web 容器但有 jdonframework.xml 普通 Application 方式:

```

ContainerSetupScript css = new ContainerSetupScript();

Application da = new Application();

css.prepare("com.jdon.jdonframework.xml", da);

AppUtil appUtil = new AppUtil("com.jdon.jdonframework.xml");

```

启动后如果出现下面错误:

```

ERROR com.jdon.bussinessproxy.target.POJOObjectFactory - [JdonFramework]create
error:org.picocontainer.defaults.UnsatisfiableDependenciesException:
com.jdon.sample.test.domain.simplecase.repository.RepositoryImp      has      unsatisfiable
dependencies:.....

```

说明 RepositoryImp 的构造函数中需要的一个类没有用 @Component 标注。



## JdonFramework 架构

JdonFramework 是一个 DDD 领域驱动设计框架，Domain Model + In-memory + Events，常驻内存 In-memory 的领域模型 Domain Model 通过领域事件 Domain Events 驱动技术实现各种功能，正如基因 DNA 是生命各种活动功能的核心一样，实现了以领域模型而不是数据表为核心的新的模型驱动开发架构 MDD。

### 没有领域设计的坏设计

以订单为例子，如果不采取 DDD 设计，而是通常朴素的数据表库设计，将订单设计为订单数据表，那么带来的问题是：

将实体的职责分离到不同限定场景，比如订单中有 OrderItemId, OrderId, ProductId 和 Qty，这是合乎逻辑的最初订单，后来有 MinDeliveryQty 和 PendingQty 字段，是和订单交货有关，这其实是两个概念，订单和订单的交货，但是我们把这些字段都混合在一个类中了。

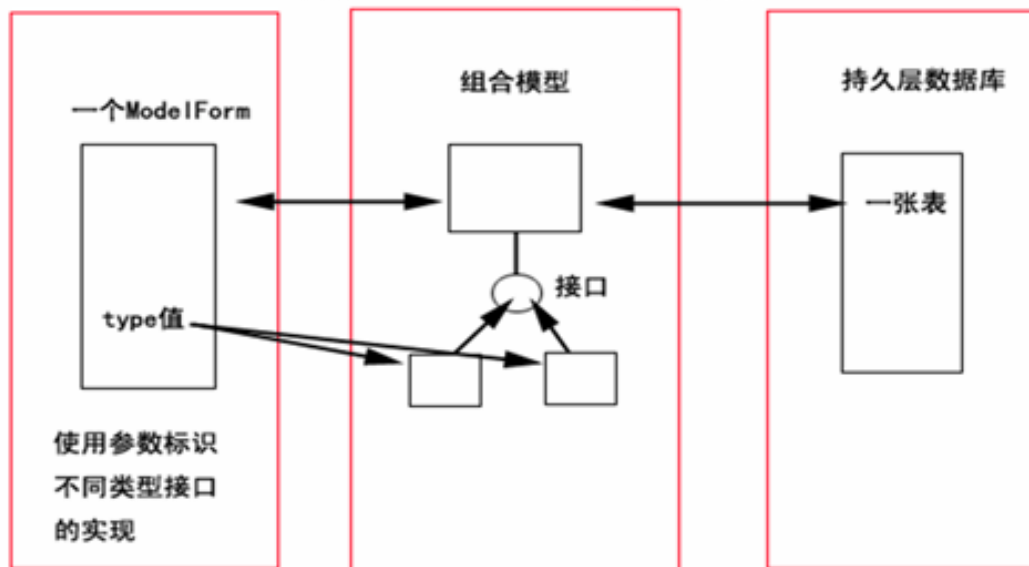
混淆在一起的问题就是将来难以应付变化，因为实体的职责是各自变化的。

领域不是把实体看成铁板一块，一开始就把它分解到各种场景。下订单和订单交货交付是两个场景，它们应该有彼此独立的接口，由实体来实现。这就能够让实体和很多场景打交道，而彼此不影响，这就是组合模型 composite model 的一个关键优点。

在数据库中它们是一个，也就是说，从 ER 模型上看，它们是一个整体，但是从 domain model 领域模型角度看，它们是分离的。

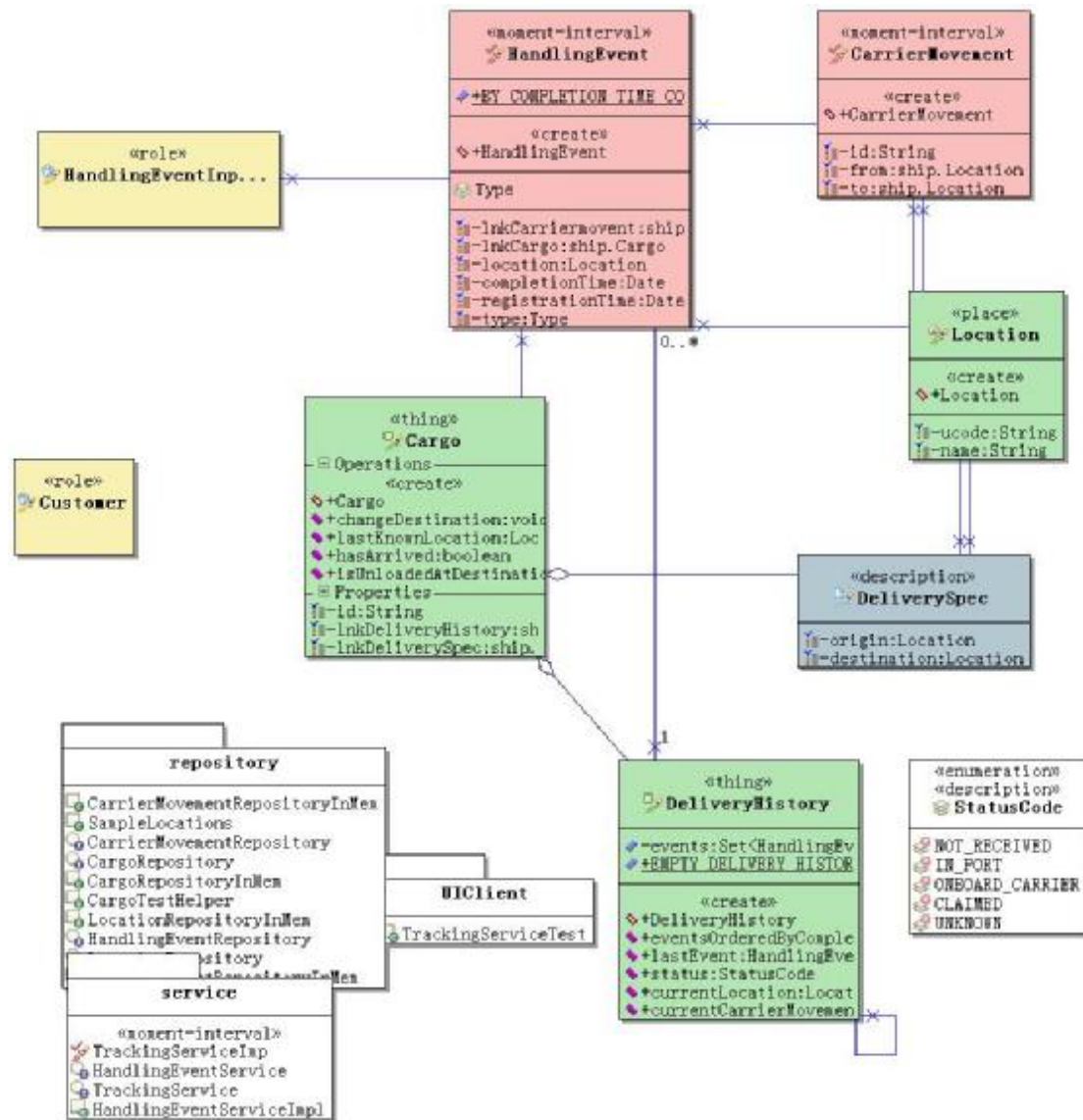
下图是基于 Jdon 框架开发的 JiveJdon 中领域模型，中间的领域模型可能是继承等关系组成的对象群，或者成为聚合群 Aggregation，但是对应的数据表可能是一张表，由此可见，领域模型能够更准确反映业务需求。

案例见JiveJdon中的Subscription实现



## 领域驱动设计 DDD

当我们接到一个新项目时，使用 UML 工具，通过面向对象 DDD 的分析设计方法将其变成领域模型图，如下：



这是一个典型的 DDD 建模图,这个模型图可以直接和 Java 代码对应,比如其中 Cargo 模型的代码如下,两者是完全一一对应,可以使用 together 等建模工具直接转换, Jdon 框架的 @Model 就是针对 Cargo 这样模型, 将其运行在 Java 平台中,:

```

package ship;
@Model
public class Cargo {
    private String id;

    private ship.DeliveryHistory lnkDeliveryHistory;
    private ship.DeliverySpec lnkDeliverySpec;

    public Cargo(String trackingId, DeliverySpec deliverySpec) {
        this.id = trackingId;
        this.lnkDeliverySpec = deliverySpec;
    }

    public void changeDestination(final Location newDestination) {

```

```

        lnkDeliverySpec.setDestination(newDestination);
    }

    //跟踪货物位置
    public Location lastKnownLocation() {
        final HandlingEvent lastEvent = this.getLnkDeliveryHistory().lastEvent();
        if (lastEvent != null) {
            return lastEvent.getLocation();
        } else {
            return null;
        }
    }

    //当货物在运输抵达目的地时
    public boolean hasArrived() {
        return lnkDeliverySpec.getDestination().equals(lastKnownLocation());
    }

    //跟踪顾客货物的关键装卸事件
    public boolean isUnloadedAtDestination() {
        for (HandlingEvent event : this.getLnkDeliveryHistory().eventsOrderedByCompletionTime())
        {
            if (HandlingEvent.Type.UNLOAD.equals(event.getType())
                && hasArrived()) {
                return true;
            }
        }
        return false;
    }

    .....
}

```

当领域模型 Cargo 出来以后，下一步就是使用 Jdon 框架来将其运行起来，因为 Jdon 框架分为领域模型和组件技术等两个部分，Cargo 无疑属于 @Model 模型架构，我们只要给模型加上 @Model，就能让 Cargo 的对象生活在内存缓存中。

```

@Model
public class Cargo {
}

```

## 面向 DDD 的事件驱动架构

为了更好地突出应用需求为主，Jdon 框架采取面向 DDD 的主要设计思想，主要特点是常驻内存 in-memory 的领域模型向技术架构发送事件消息，驱动技术架构为之服务，如同人体的 DNS 是人体各种活动的主要控制者和最高司令，领域模型是一个软件系统的最高司令。

JF 有五种模型组件，如下：

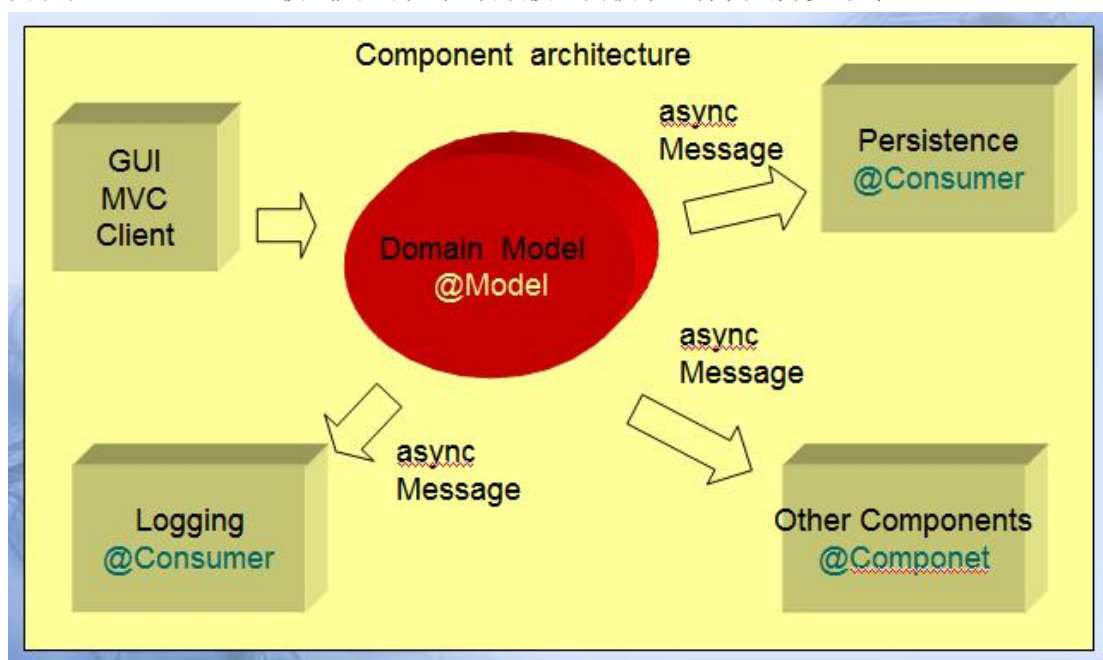
- 一. 实体聚合根对象 元注释 `@Model`;
  - 二. 服务 Service 元注释 `@Service`;
  - 三. 普通类组件构件 `@Component`;
  - 四. 生产者 Producer-消费者模型 `@send @Consumer`;
  - 五. 拦截器 `@Interceptor`, 首先需要导入点 `@Introduce`;
- 所有都在 `com.jdon.annotation.*`包中。

这些模型组件其实有划分为两大类：业务和技术：

常驻内存 `@Model` 领域模型，包括实体模型 值对象和领域服务，与技术架构无关。  
相当于鱼；生存空间是缓冲器中

`@Component` 技术组件架构，用以支撑领域模型在计算机系统运行的支撑环境，相当于鱼生活的水。空间在 Context container,例如 `ServletContext` 中。

两者以 Domain Events 模式交互方式：领域模型向技术组件发出异步命令。



## 实体模型

根据 DDD 方法，需求模型分为实体模型 值对象和领域服务三种，实际需求经常被划分为不同的对象群，如 Cargo 对象群就是 Cargo 为根对象，后面聚合了一批与其联系非常紧密的子对象如值对象等，例如轿车为根对象，而马达则是轿车这个对象群中的一个实体子对象。

在 Jdon 框架中，实体根模型通常以 `@Model` 标识，并且要求有一个唯一的标识主键，你可以看成和数据表的主键类似，它是用来标识这个实体模型为唯一的标志，也可以使用 Java 对象的 `HashCode` 来标识。

Jdon 框架是实体模型的主键标识有两个用处：

首先是用来缓存实体模型，目前缓冲器是使用 `EHcache`，可以无缝整合分布式云缓存

Terracotta 来进行伸缩性设计。

只要标识 @Model 的实体，在表现层 Jsp 页面再次使用时，将自动直接从缓存中获得，因为在中间业务层和表现层之间 Jdon 框架存在一个缓存拦截器 CacheInterceptor，见框架的 aspect.xml 中配置。

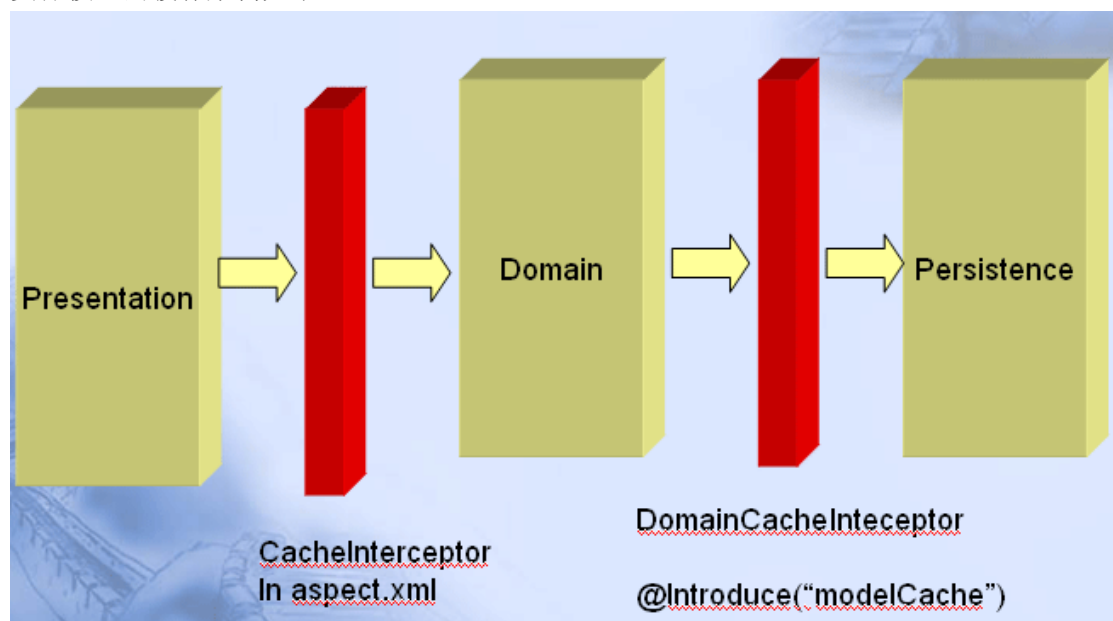
为了能够在业务层能够使用缓存中的模型，需要在业务层后面的持久层或 Repository 中手工加入缓存的 Annotation 标签，如下：

```
@Component("mymrepository")
@Introduce("modelCache")
public class RepositoryImp implements MyModel {

    @Around
    public MyModel getModel(Long key) {
        MyModel mym = new MyModel();
        mym.setId(key);
        return mym;
    }
}
```

Auto cache MyModel

实体模型的缓存架构如下：



注意：这里可能是 Jdon 框架 6.2 的一个创新或重点，在 JF6.2 之前实际上没有对模型进行突出的支持，就象画个圈，圈子外面基本都就绪，圈子里面留白，这个圈子就是 Domain Model，原来因为考虑持久层 Hibernate 等 ORM 巨大影响力，就连 Spring 也是将 Model 委托给 Hibernate 处理，自己不做模型层，但是当 NoSQL 运动蓬勃发展，DDD 深入理解，6.2 则找到一个方式可以介入模型层，同时又不影响任一持久层框架的接入。

Jdon 框架 6.2 通过在持久层的获得模型对象方法上加注释的办法，既将模型引入了内存缓存，又实现了向这个模型注射必要的领域事件 Domain Events。

## 事件 Event Sourcing

JF 的最大特点是领域模型驱动技术架构；

如果说普通编程缺省是顺序运行，那么事件模型缺省是并行运行，两者有基本思路的不同。

Event Sourcing 适合将复杂业务领域和复杂技术架构实现分离的不二之选。实现业务和技术的松耦合，业务逻辑能够与技术架构解耦，将”做什么”和”怎么做”分离

事件模型也是一种 EDA 架构 Event-driven Architecture，可以实现异步懒惰加载 Asynchronous Lazy-load 类似函数式语言的懒功能，只有使用时才真正执行。

具有良好的可伸缩性和可扩展性，通过与 JMS 等消息系统结合，可以在多核或多个服务器之间弹性扩展。

事件模型也是适合 DDD 落地的最佳解决方案之一。领域模型是富充血模型，类似人类的 DNA，是各种重要事件导向的开关。用户触发事件，事件直接激活领域模型的方法函数，再通过异步事件驱动技术活动，如存储数据库或校验信用卡有效性等。

2009 年 Jdon 框架 6.2 就推出了 Domain Model + In-memory + Events., 2001 年 Martin fowler 在其文章 [LMAX 架构](#) 推荐 In-memory + Event Sourcing 架构。以下是该文的精选摘要，从一个方面详细说明了事件模型的必要性：

内存中的领域模型处理业务逻辑，产生输出事件，整个操作都是在内存中，没有数据库或其他持久存储。将所有数据驻留在内存中有两个重要好处：首先是快，没有 IO，也没有事务，其次是简化编程，没有对象/关系数据库的映射，所有代码都是使用 Java 对象模型。

使用基于内存的模型有一个重要问题：万一崩溃怎么办？电源掉电也是可能发生的，“事件”(Event Sourcing)概念是问题解决的核心，业务逻辑处理器的状态是由输入事件驱动的，只要这些输入事件被持久化保存起来，你就总是能够在崩溃情况下，根据事件重演重新获得当前状态。

事件方式是有价值的因为它允许处理器可以完全在内存中运行，但它有另一种用于诊断相当大的优势：如果出现一些意想不到的行为，事件副本们能够让他们在开发环境重放生产环境的事件，这就容易使他们能够研究和发现出在生产环境到底发生了什么事。

这种诊断能力延伸到业务诊断。有一些企业的任务，如在风险管理，需要大量的计算，但是不处理订单。一个例子是根据其目前的交易头寸的风险状况排名前 20 位客户名单，他们就可以切分到复制好的领域模型中进行计算，而不是在生产环境中正在运行的领域模型，不同性质的领域模型保存在不同机器的内存中，彼此不影响。

LMAX 团队同时还推出了开源并发框架 [Disruptor](#)，他们通过测试发现通常的并发模型如 Actor 模型是有瓶颈的，所以他们采取 Disruptor 这样无锁框架，采取了配合多核 CPU 高速缓冲策略，而其他策略包括 JDK 一些带锁都是有性能陷阱的：[JVM 伪共享](#)。

JF 的领域事件是基于号称最快的并发框架 [Disruptor](#) 开发的，因此 JF 的事件是并行并发模型，不同于普通编程是同一个线程内的顺序执行模型。

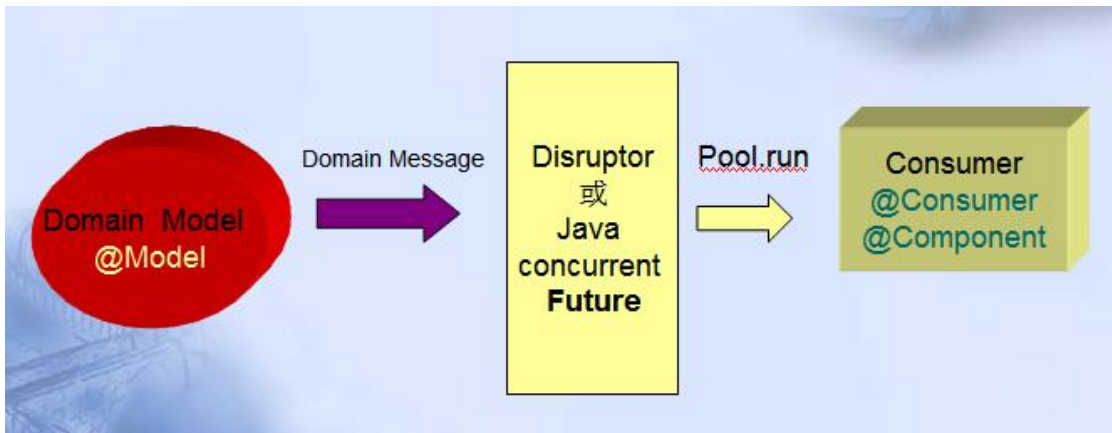
JF 的领域事件是一种异步模式 + 生产者-消费者模式。主题 topic 和 Queue 队列两种。领域模型是生产者；消费者有两种：

.@Consumer; 可以实现 1:N 多个，内部机制使用号称最快的并发框架 Disruptor 实现。适合轻量；小任务；原子性；无状态。

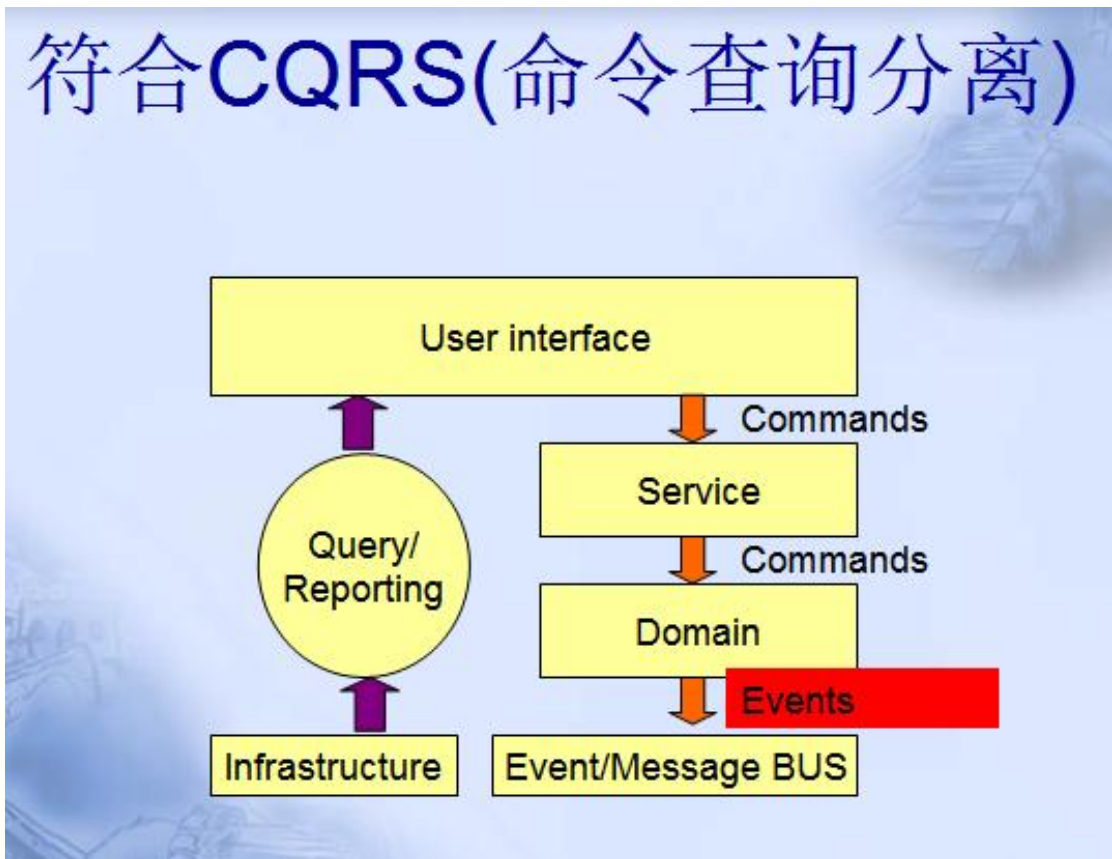
.@Componet; 直接使用普通组件类作为消费者，使用 jdk future 机制，只能 1:1，适合大而繁重的任务，有状态，单例。



Domain Events 实现机制如下:



JF 的事件模型还是一种 CQRS 架构, 可以实现模型的修改和查询分离, 也就是读写分离的架构:



## 无堵塞的并发编程

顺序编程和并发编程是两种完全不同的编程思路, 堵塞 Block 是顺序编程的家常便饭, 常常隐含在顺序过程式编程中难以发现, 最后, 成为杀死系统的罪魁祸首; 但是在并发编程中, 堵塞却成为一个目标非常暴露的敌人, 堵塞成为并发不可调和绝对一号公敌。

因为无堵塞所以快, 这已经成为并发的一个基本特征。



过去我们都习惯了在一个线程下的顺序编程，比如，我们写一个 Jsp(PHP 或 ASP)实际都是在一个线程

下运行，以 google 的 adsense.Jsp 为例子：

```
<%  
//1.获得当前时间  
long googleDt = System.currentTimeMillis();  
//2.创建一个字符串变量  
StringBuilder googleAdUrlStr = new StringBuilder(PAGEAD);  
//3.将当前时间加入到字符串中  
googleAdUrlStr.append("&dt=").append(googleDt);  
//4.以字符串形成一个 URL  
URL googleAdUrl = new URL(googleAdUrlStr.toString());  
%>
```

以上 JSP 中 4 步语句实际是在靠一个线程依次顺序执行的，如果这四步中有一步执行得比较慢，也就是我们所称的是堵塞，那么无疑会影响四步的最后执行时间，这就象乌龟和兔子过独木桥，整体效能将被跑得慢的乌龟降低了。

过去由于是一个 CPU 处理指令，使得顺序编程成为一种被迫的自然方式，以至于我们已经习惯了顺序运行的思维；但是如今是双核或多核时代，我们为什么不能让两个 CPU 或多个 CPU 同时做事呢？

如果两个 CPU 同时运行上面代码会有什么结果？首先，我们要考虑两个 CPU 是否能够同时运行这段逻辑呢？

考虑到第三步是依赖于前面两步，而第二步是不依赖第一步的，因此，第一步和第二步可以交给两个 CPU 同时去执行，然后在第三步这里堵塞等待，将前面两步运行的结果在此组装。

很显然，这四步中由于第三步的堵塞等待，使得两个 CPU 并行计算汇聚到这一步又变成了瓶颈，从而并不能充分完全发挥两个 CPU 并行计算的性能。

我们把这段 JSP 的第三步代码堵塞等待看成是因为业务功能必须要求的顺序编程，无论前面你们如何分开跑得快，到这里就必须合拢一个个过独木桥了。

但是，在实际技术架构中，我们经常也会因为非业务原因设置人为设置各种堵塞等待，这样的堵塞就成为并行的敌人了，比如

我们经常有(特别是 Socket 读取)

```
While(true){  
    .....  
}
```

这样的死循环，无疑这种无限循环是一种堵塞，非常耗费 CPU，它无疑成为并行的敌人。

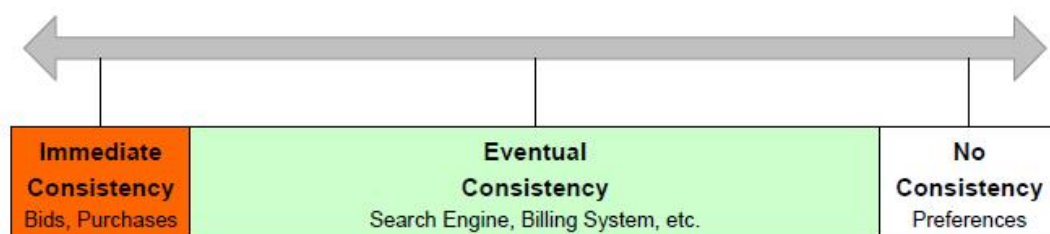
比如 JDK 中 java.concurrent. BlockingQueue LinkedBlockingQueue,也是一种堵塞式的所谓并行包，这些并行功能必须有堵塞存在的前提下才能完成并行运行，很显然是一种伪并行。

由于各种技术上的堵塞存在，包括多线程中锁机制，也是一种堵塞，因为锁机制在某个时刻只允许一个线程进行修改操作，所以，并发框架 Disruptor 可以自豪地说：无锁，所以很快。

现在非常流行事件编程模型，如 Event Sourcing 或 Domain Events 或 Actor 等等，事件编程实际是一种无堵塞的并行编程，因为事件这个词语本身有业务模型上概念，也有技术平台上的一个规范，谈到事件，领域专家明白如同电话铃事件发生就要接，程序员也能明白只要有事件，CPU 就要立即处理(特别是紧急事件)，而且事件发生在业务上可能是随机的，因此事件之间难以形成互相依赖，这就不会强迫技术上发生前面 Jsp 页面的第三步堵塞等待。

因此，在事件模型下，缺省编程思维习惯是并发并行的，如果说过去我们缺省的是进行一个线程内的顺序编程，那么现在我们是多线程无锁无堵塞的并发编程，这种习惯的改变本身也是一种思维方式的改变。

在缺省大多数是并发编程的情况下，我们能将业务上需要的顺序执行作为一种特例认真小心对待，不要让其象癌细胞一样扩散。我们发现这种业务上的顺序通常表现为一种高一致性追求，更严格的一种事务性，每一步依赖上一步，下一步失败，必须将上一步回滚，这种方式是多核 CPU 克星，也是分布式并行计算的死穴。值得庆幸的是这种高一致性的顺序编程在大部分系统中只占据很小一部分，下图是电子商务 EBay 将他们的高一致性局限在小部分示意图：



由此可见，过去我们实现的顺序编程，实际上是我们把一种很小众的编程方式进行大规模推广，甚至作为缺省的编程模式，结果导致 CPU 闲置，吞吐量上不去同时，CPU 负载也上不去，CPU 出工不出力，如同过去计划经济时代的人员生产效率。

据统计，在一个堵塞或有锁的系统中，等待线程将会闲置，这会消耗很多系统资源。消耗资源的公式如下：

闲置的线程数  $= (\text{arrival\_rate} * \text{processing\_time})$

如果 arrival\_rate(访问量达)很高，闲置线程数将会很大。堵塞系统是在一个没有效率的方式下运行，无法通过提高 CPU 负载获得高吞吐量。

避免锁或同步锁有多种方式，volatile 是一种方式，主要的不变性设计，比如设计成 DDD 的值对象那种，要变就整个更换，就不存在对值对象内部共享操作；还有克隆原型，比如模型对象克隆自己分别传给并发的多个事件处理；用 Visibility 使资料对所有线程可见等等；

最彻底的方式就是使用无锁 Queue 队列的事件或消息模型，Disruptor 采取的是一个类似左轮手枪圆环 RingBuffer 设计，见 <http://www.jdon.com/jivejdon/thread/42466>，这样既能在两个线程之间共享状态，又实现了无锁，比较巧妙，业界引起震动。

当然 Scala 的那种 Share nothing 的 Actor 模型也是一种无锁并发模型，不过奇怪的是，发明 Disruptor 的 LMAX 团队首先使用过 Actor 模型，但是并发测试发现还是有性能瓶颈的，所以，他们才搞了一个 Disruptor。

领域模型发出事件，事件由 Disruptor 的 RingBuffer 传递给另外一个线程(事件消费者)；实现两个线程之间数据传递和共享。

当事件消费者线程处理完毕，将结果放入另外一个 RingBuffer，供原始线程(事件生

生产者)读取或堵塞读取，是否堵塞取决于是否需要顺序了。

非堵塞并发使用模式如下：

发出调用以后不必立即使用结果，直至以后需要时才使用它。发出调用后不要堵塞在原地等待处理结果，而是去做其他事情，直至你需要使用这个处理结果时才去获取结果。

被调用者将会返回一个“future”句柄，调用者能够获得这个句柄后做其他事情，其他事情处理完毕再从这个句柄中获得被调用者的处理结果。

JF 6.5 版本为推广适合多核 CPU 的无堵塞并发编程范式进行了探索，使用了 Domain Events 和 DCI 等不同抽象层次对并发编程进行了封装，从而降低开发者使用并发编程的难度。

## DCI 架构

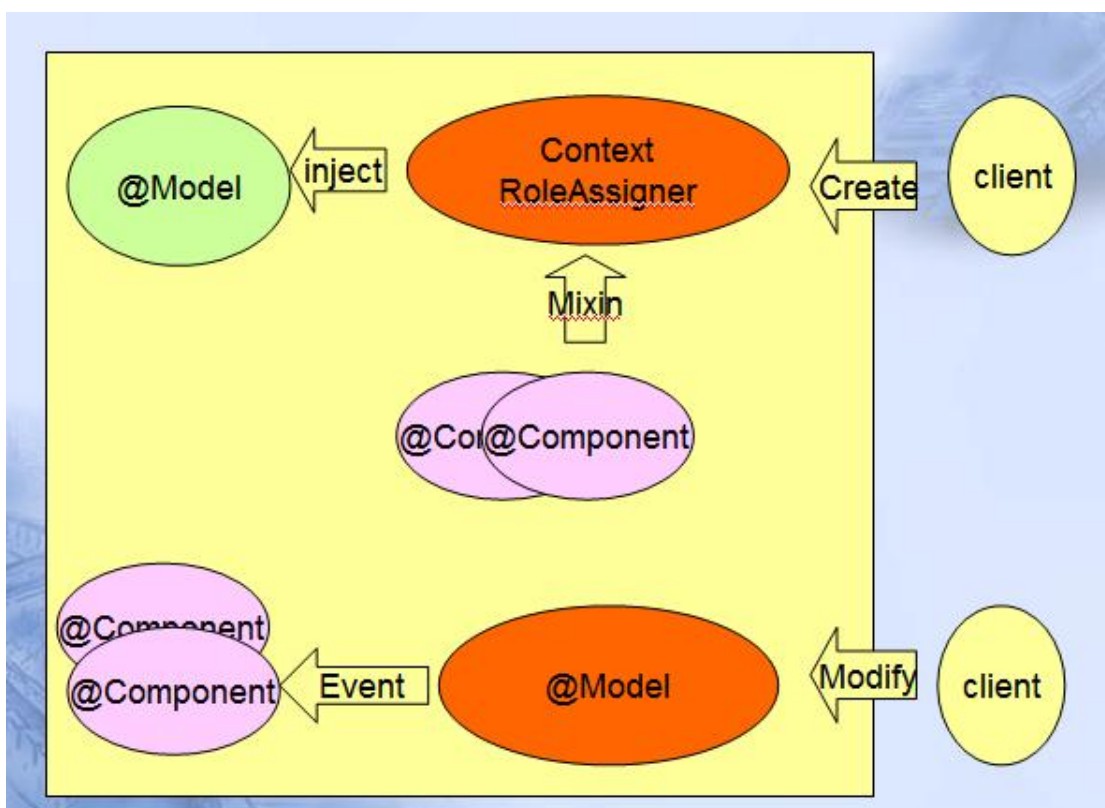
DCI：数据 Data，上下文(场景)Context，交互 Interactions 是由 MVC 发明者 [Trygve Reenskaug](#) 发明的。其核心思想是：

让我们的核心模型更加简单，只有数据和基本行为。业务逻辑等交互行为在角色模型中在运行时的场景，将角色的交互行为注射到数据中。

Jdon 框架提供了两种 DCI 风格实现：一种是将 DomainEvents 对象注射进入当前模型；还有一种是直接将数据模型和接口混合。这两种方式适合不同场景。

如果我们已经 Hold 住了一个领域对象，那么就直接通过其发出领域事件实现功能；比如模型的修改。在这种模式下，发出领域事件的领域模型本身已经隐含了场景。事件代表场景出头牵线。

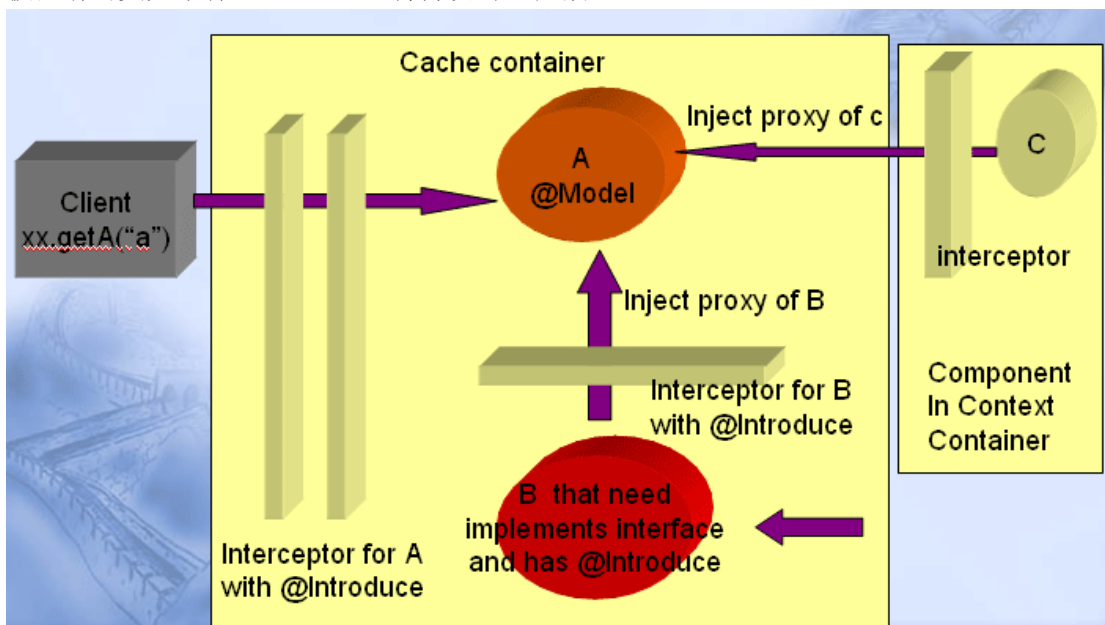
否则，我们显式创建一个上下文 Context，在其中通过 RoleAssigner 将角色接口注入到领域对象中。比如模型新增创建或删除。(对象不能自己创建自己)。



具体实现可见下面专门 DCI 案例章节。

## 依赖注入 DI

(一) @Model: 模型中可以通过字段的 @Inject 将其他类注射进入, 包括 @Component 类。被注射的类如果有 @Introduce, 将再次引入拦截器。



```

@Model
public class MyModel {

    private Long id;
    private String name;

    @Inject
    private MyModelDomainEvent myModelDomainEvent;

    @Inject
    private MyModelService myModelServiceCommand;

    @Introduce("message")
    public class MyModelDomainEvent {

        @Send("MyModel.findName")
        public ModelMessage asyncFindName(MyModel myModel) {
            return new ModelMessage(myModel);
        }
    }
}

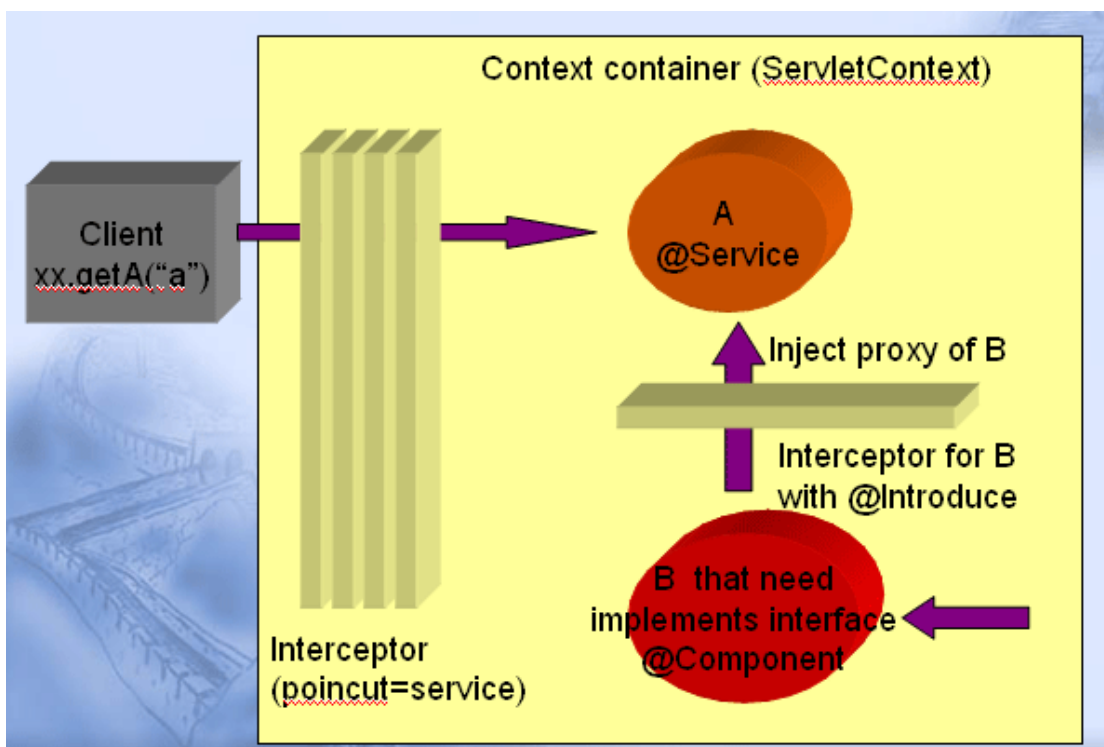
```

Inject

Introduce a interceptor

MyModel  $\xrightarrow{\text{invoke}}$  @Introduce(message)  $\xrightarrow{\text{invoke}}$  MyModelDomainEvent

(二) @Component: 技术架构中组件可以通过构造器直接注射，被注射的类如果有 @Introduce，将再次引入拦截器。



## AOP 拦截器

Jdon 框架拦截器在 6.2 版本以后得到加强和细化，不过和 AspectJ 等 AOP 框架有些区别，主要是方便性方面，这些 AOP 框架有专门的复杂 Poincut 表达式，需要再学习，而 Jdon 框架则还是使用 Java 完成复杂的拦截方式。

有两种拦截器写法，一种是写在被拦截的类代码中，而拦截器则无特别写法，解放了

拦截器；另外一种拦截语法写在拦截器中，解放了被拦截者，实际中可根据自己的情况灵活选用。

## @Introduce

@Introduce(名称) 为当前类引入拦截器，可以用在 @Model 或 @Component 中，名称是拦截器的 @Component(名称)。 例如 @Introduce("c") --- @Component("c")

在当前类具体方法上使用 @Before @After 和 @Around 三种拦截器激活的方式。注意使用 @Around 对拦截器有特定写法要求，其他无。

```
@Component("a")
@Introduce("c") // Introduce a interceptor that component name is "c"
public class A implements AInterface {

    @Before("testOne") // Interceptor will action before "myMethod "
    public Object myMethod(@Input() Object inVal, @Returning() Object ret) {
        System.out.println("this is A.myMethod is active!!!! ");
        int i = (Integer) inVal + 1;
        return i;
    }

    @After("testWo") // Interceptor will action after "MyMethod2"
    public Object myMethod2(Object inVal) {
        System.out.println("this is A.myMethod2 is active!!!! ");
        int i = (Integer) inVal + 1;
        return i;
    }
}
```

Annotations and their targets in the code above:

- `@Introduce("c")`: Introduces an interceptor named "c".
- `@Before("testOne")`: Specifies that the interceptor "testOne" should run before the `myMethod` method.
- `@After("testWo")`: Specifies that the interceptor "testWo" should run after the `myMethod2` method.
- `@Input()`: Indicates that the parameter `inVal` in `myMethod` is injected from the interceptor's `testOne` method.
- `@Returning()`: Indicates that the return value of `myMethod` is injected into the `testOne` method of the interceptor.

被引入的拦截器 c 的代码如下，没有特别要求，只要配置 @Component("c") 就可以，或者使用 XML 配置：<component name="c" class="xxxxxx" /> 也可以。

```
@Component("c")
public class C {

    //被拦截器的@Input 参数将注射到 inVal 中
    public Object testOne(Object inVal) {
        ....
    } //testOne 方法 return 结果将被注射到被拦截器的@return 中

    //被拦截器的方法 myMethod2 返回值将被引入此方法的 inVale
    public Object testWo(Object inVal) {
        .....
    }
}
```

@Introduce 的 @Around 则是一个特殊情况，因为 around 表示在被拦截点周围，实际是 Before + After 综合写法，@Around 需要对拦截器写法有一些特殊要求，如下：

```

@Interceptor(“aroundAdvice”)//1. 需要表明自己是 Inteceptor
public class AroundAdvice implements MethodInterceptor {

    //2.需要实现 org.aopalliance.intercept.MethodInterceptor 接口
    //3.完成接口的 Invoke 方法
    public Object invoke(MethodInvocation invocation) throws Throwable
        Object o = invocation.proceed();//在此方法前后写包围 around 代码。
}

```

## @Interceptor

@Interceptor 是将拦截定义在拦截器这边的语法，和 AspectJ Spring 写法类似，但是没有他们的复杂 pointcut 专门表达式。

```

//1. 指定被拦截的类是@Component(“a”)和@Component(“c”)
@Interceptor(name = "myInterceptor", pointcut = "a,c")
public class MyInterceptor implements MethodInterceptor {

    //2.需要实现 org.aopalliance.intercept.MethodInterceptor 接口
    //3.完成接口的 Invoke 方法
    public Object invoke(MethodInvocation methodInvocation) throws java.lang.Throwable {
        ...
    }
}

```

如果直接写@Interceptor(“myName”)那么就表示该拦截器的拦截点 pointcut 是针对所有 Service 或 POJO Services，相当于在 aspectj.xml 中的配置

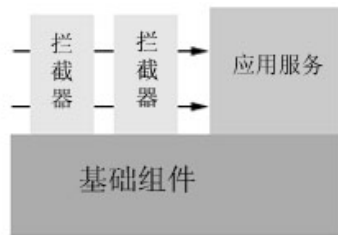
@Interceptor 中的 pointcut 是指被拦截的类名称，可以是多个类一起拦截。

## 框架内部组件机制

Jdon 框架可以实现几乎所有组件可配置、可分离的管理，这主要得益于 Ioc 模式的实现，Jdon 框可以说是一个组件（JavaBeans）管理的微容器。

在 Jdon 框架中，有三种性质的组件（JavaBeans）：框架基础组件；AOP 拦截器组件和应用服务组件。三种性质的组件都是通过配置文件实现可配置、可管理的，框架应用者替换这三种性质组件的任何一个。





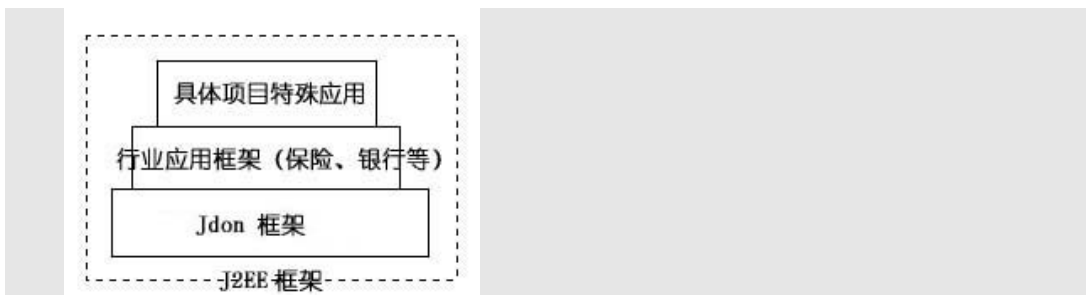
框架基础组件是 Jdon 框架最基本的组件，是实现框架基本功能的组件，如果框架应用者对 Jdon 框架提供的功能不满意或有意替换，可以编写自己的基础功能组件替代，从而实现框架的可彻底分离或管理。Jdon 框架功能开发基本思路是：当有新的功能加入 Jdon 框架时，总是让该功能组件实现可配置和可更换，以使得该功能代表的一类一系列其他功能有加入拓展的余地。

应用服务组件是框架应用者针对具体项目设计的组件，如用户管理 AccountService、订单服务 OrderService 等都属于应用服务组件。

AOP 拦截器组件主要是指一些应用相关的通用基础功能组件，如缓存组件、对象池组件等。相当于应用服务组件前的过滤器（Filter），在客户端访问应用服务组件之前，必须首先访问的组件功能。

这三种性质组件基本涵盖了应用系统开发大部分组件，应用服务组件是应用系统相关组件，基本和数据库实现相关，明显特征是一个 DAO 类；当应用服务组件比较复杂时，我们就可以从中重整 Refactoring 出一些通用功能，这些功能可以上升为框架基础组件，也可以抽象为 AOP 拦截器组件，主要取决于它们运行时和应用服务组件的关系。当然这三种性质框架组件之间可以相互引用（以构造方法参数形式），因为它们注册在同一个微容器中。

使用 Jdon 框架，为应用系统开发者提炼行业框架提供了方便，框架应用者可以在 Jdon 框架基本功能基础上，添加很多自己行业特征的组件，从而实现了框架再生产，提高应用系统的开发效率。



在 JdonFramework 中，所有组件都是在配置文件中配置的，框架的组件是在 container.xml 和 aspect.xml 中配置，应用系统组件是在 jdonframework.xml 中配置，应用系统组件和框架内部或外部相关组件都是在应用系统启动时自动装载入 J2EE 应用服务器中，它们可以相互引用（以构造器参数引用，只要自己编写的普通 JavaBeans 属于构造器注射类型的类就可以），好似是配置在一个配置文件中一样。

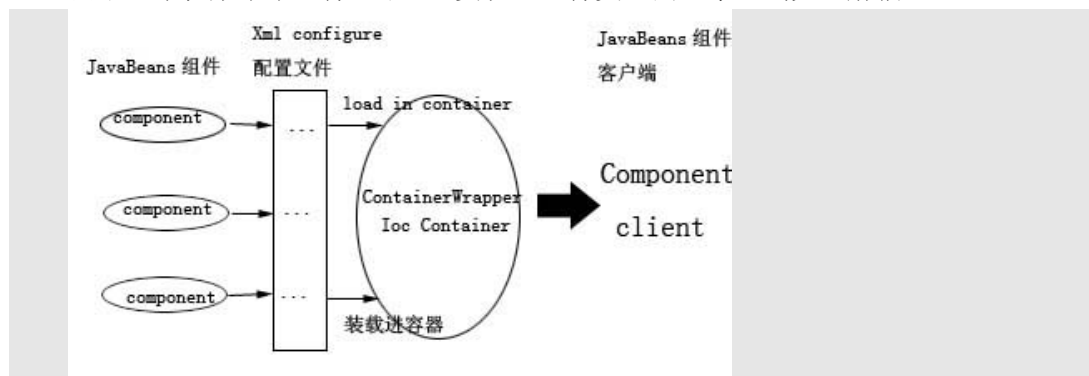
因此，组件配置主要有三个配置文件：应用服务组件配置 container.xml、AOP 拦截器组件 aspect.xml 和应用服务组件配置 jdonframework.xml 另外也有一套 Annotation 来起到和这些 XML 同样作用的配置。





## 内部原理

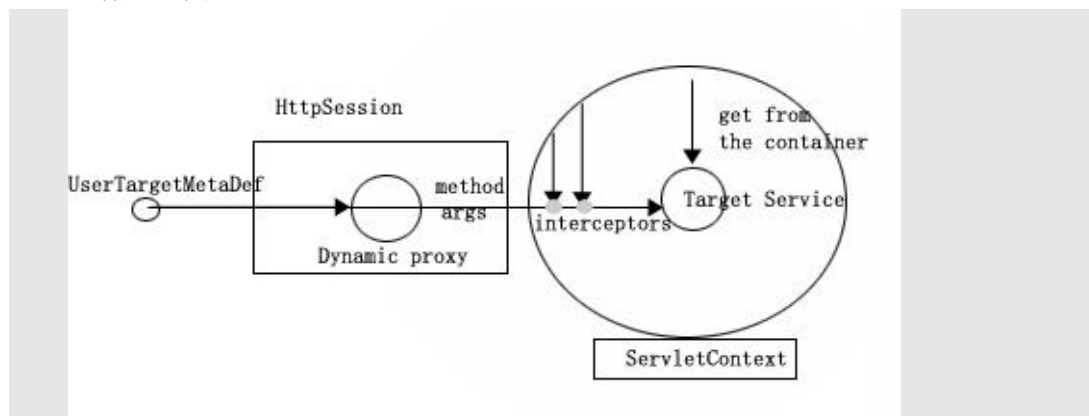
由于 Jdon 框架设计不断演进中，最新设计可见 [JdonFramework.ppt](#) 最新 PPT 文档  
由于整个框架中组件基于 Ioc 实现，组件类之间基本达到完全解耦。



从上图可以看出，任何 JavaBeans 组件只要在 XML 配置文件（container.xml aspect.xml 和 jdonframework.xml）中配置，由容器装载机制注册到 Ioc 容器中，这些组件的客户端，也就是访问者，只要和微容器（或者组件的接口）打交道，就可以实现组件访问。

因此，本框架中每个功能块都可从框架中分解出来，单独使用它们。用户自己的任意功能块也可以加入框架中，Jdon 框架成为一种完全开放、伸缩自如的基础平台。

运行原理图：



当客户端请求产生时，首先将访问目标服务参数包装在 userTargetMetaDef 对象中，该对象访问位于 HttpSession 中的动态代理实例，被动态代理将目标服务肢解成 Method 和方法参数 Args，然后被拦截器拦截，最后达到目标服务实例，当然，有的拦截器可能

就此完全拦截（堵住去路，如 `PoolInterceptor`），目标服务实例是事先被注册到容器中的，在被访问时，缺省情况是每次访问产生一个新的目标服务实例。

Jdon 容器如图是被保存在 `Servlet` 容器的 `ServletContext` 中的。

## 依赖注入组件包

`Container` 包主要是负责容器管理方面的功能，基于 `Picocontainer` 实现依赖注入功能。其他包中的组件都被写入配置文件 `container.xml`、`aspect.xml` 和 `jdonframework.xml` 中，而 `container` 包主要负责与这些配置文件打交道，从配置文件中获得其他包的组件，向容器中注册，并启动容器。

主要一个接口是 `ContainerWrapper`，`ContainerWrapper` 有两个主要方法：向容器注册组件；从容器查询获得组件。

`ContainerWrapper` 接口的缺省实现是 `PicoContainerWrapper` 和 `VisitorContainerWrapper` 两个子类，如下图：



`PicoContainerWrapper` 使用了著名的 `PicoContainer` 实现(<http://www.picocontainer.org>)，`ContainerWrapper` 接口主要从其抽象出来。

## 访问者模式

`VisitorContainerWrapper` 是观察者模式的实现，传统观察者模式中，对于 `Visitor` 角色一般有以下多个访问不同被访问者的方法：

```
visitAcomponent();
visitBcomponent();
.....
```

由于 `Acomponent`、`Bcomponent` 这些类已经注册到容器，因此，通过容器可以直接实现不同组件的访问，只需通过下面一个方法实现：

```
public ComponentVisitor getComponentVisitor(){
    return new ComponentOriginalVisitor(this);
}
```

而访问者 `Visitor` 则可以节省为单一方法即可：

```
visit(XXX xxx);
```

使用访问者模式的原因：主要为了实现缓存，提高一些组件运行性能，如果一些组件每次访问时，都需要 `new`，例如 `Proxy.newInstance` 如果频繁反复运行，将是十分耗费性能的，因此，使用缓存尽量避免每次创建将提高系统的运行性能。

`Visitor` 有两个子类实现：`ComponentOriginalVisitor` 和 `HttpSessionProxyVisitor`，这里又使用了装饰者模式，`Decoratee` 是 `ComponentOriginalVisitor`；而 `Decorator` 是

HttpSessionProxyVisitor, HttpSessionProxyVisitor 是 HttpSessionBindingListener, 也就是说, 我们使用了 HttpSession 作为缓存机制, HttpSession 的特点是以用户为 Key 的缓存, 符合 ♂ 的缓存机制, 当然, 我们以后也可以使用更好的缓存机制替换 HttpSession, 替换了 HttpSession 根本不必涉及到其他类的更好, 因为这里使用模式实现了彻底的解耦。

访问者模式另外一个重要角色: Visitable, 它则是那些运行结果需要缓存的组件必须继承的, 注意, 这里有一个重要点: 不是那些组件本省生成需要缓存, 而是它的运行结果需要缓存的。继承 Visitable 这些组件事先必须注册在容器中。

目前 Visitable 有两个子类:

- \* @see com.jdon.bussinessproxy.dyncproxy.ProxyInstanceFactoryVisitable

- \* @see com.jdon.bussinessproxy.target.TargetServiceFactoryVisitable

前者主要是动态代理的创建, 因为 Proxy.newInstance 频繁执行比较耗费性能, 第一次创建后, 将动态代理实例保存在 httpSession 中, 当然每个 Service 对应一个动态代理实例。

TargetServiceFactoryVisitable 主要是为了缓存那些目标服务的实例, 目前这个功能没有激活, 特殊情况下才推荐激活。

## 容器的启动

容器启动主要是将配置文件中注册的组件注册到容器中, 并启动容器, 这涉及到 container 包下 Config 和 Builder 等几个子包。

Config 包主要负责从 container.xml 和 aspect.xml 读取组件;

Builder 包主要负责向 ContainerWrapper 注册这些组件, 注册过程是首先从基础组件 (container.xml) 开始, 然后是拦截器组件 (aspect.xml), 最后是用户的服务组件 (jdonframework.xml)。

## 容器的生命周期

容器启动后, 容器本身实例是放置在 Web 容器的 ServletContext 中。

容器启动并不是在应用系统部署到 Web 容器时就立即启动, 而是该应用系统被第一次访问时触发启动, 这个行为是由 ContainerSetupScript 的 startup 触发的, 而 startup 方法则是由 ServletContainerFinder 的 findContainer 触发。

当应用系统从 Web 容器中销毁或停止, Jdon 框架容器也就此销毁, 最好将你的组件中一些对象引用释放, 只要继承 Startable, 实现 stop 方法即可。

## 容器的使用

客户端访问组件必需通过容器进行, 这个过程分两步:

从 Web 容器中获得框架容器 ContainerWrapper 实例。

从 ContainerWrapper 容器中查询获得组件实例, 有两者实例方式: 单例和多例。

关于具体使用方式可见前面章节“如何获得 POJO 实例”等。

这个容器使用过程是通过 finder 包下面的类完成, 主要是 ServletContainerFinder 类, ComponentKeys 类保存中一些 container.xml 中配置的组件名称, 这些组件名称可能需要在

框架程序中使用到，这就需要引起注意，这个类中涉及的组件名称不要在 container.xml 中随意改动（当然这个类在以后重整中争取去除）。

com.jdon.controller.WebAppUtil 是专门用于客户端对微容器的访问，这个客户端目前是 Http 客户端，主要方法：

```
public static Object getService(String name, HttpServletRequest request)
```

这是最常用的从容器中获得服务实例的方法。需要 HttpServletRequest 作为客户端，Jdon 框架将来会提供作为 Application 客户端调用的专门方法。

```
public static Object getComponentInstance(String name, HttpServletRequest request)
```

这是获得组件实例的方法。

```
public static String getContainerKey()
```

通过获得容器保存在 ServletContext 中的 Key。

## 从容器内部访问容器

上节“容器的使用”是指从容器外部（客户端）如何访问和使用容器，如果当前客户端是在容器内部，例如需要在一个 Service 服务类中访问容器的缓存机制等，该如何访问？

使用 com.jdon.container.finder.ContainerCallback，同时，该服务 POJO 类以 ContainerCallback 作为构造参数，当该 POJO 服务类注册到容器中时，容器的 Ioc 特性将会找到事先以及注册的 ContainerCallback 类。

通过 ContainerCallback 获得 ContainerWrapper 容器实例，然后通过 ContainerWrapper 下面两个方法：

```
public Object lookup(String name);
```

从容器中获得 container.xml 中注册的组件实例，这种方法每次调用获得的是同一个实例，相当于单例方式获得。

```
public Object getComponentNewInstance(String name);
```

每次从容器中获得 container.xml 中注册的组件的一个新实例，这种方法每次调用获得的是新的实例，该方法不适合一些单例资源的获得。

例如如果向在 POJOService 中访问容器的缓存机制，因为缓存整个容器只能有一处，因此必须以 lookup 方式获得，获得 ModelManager 的实例如下：

```
ModelManager modelManager = containerWrapper.lookup("modelManager");
```

其中“modelManager”字符串名称是从 Jdon 框架的 jdonFramework.jar 包中 META-INF 的 container.xml 中查询获知的。

## AOP 包

Jdon AOP 的设计目前功能比较简单，AOP 包下有几个子包：Interceptor、joinpoint 和 reflection，分别涉及拦截器、；拦截器切点和反射机制等功能。

Jdon 框架从架构角度来说，它是一种业务代理，前台表现层通过业务代理层访问业务服务层，使用 AOP 实现业务代理是一种新的趋势，因此本包功能是和 AOP 包功能交互融合的。

bussinessproxy 包中的 config 子包是获取 jdonframework.xml 中的两种服务 EJB 或

POJO 配置，然后生成 meta 子包中 TargetMetaDef 定义。

target 子包封装了关于这两种服务由元定义实现对象创建的工厂功能，服务对象创建透过访问模式使用了 HttpSession 作为 EJB 实例缓存，这个功能主要是为了 EJB 中有态会话 Bean 实现，这样，客户端通过 getService 获得一个有态会话 Bean 时，无需自行考虑保存这个 Bean 引用到 HttpSession 中这一使用细节了。无态会话 bean 从实战角度发现也可以缓存，提高了性能。

Jdon 框架既然是一种业务代理，那么应该服务于各种前台表现层，Jdon 框架业务代理还适合客户端为 Java 客户端情况下的业务服务访问。这部分功能主要是 com.jdon.bussinessproxy.remote 子包实现的。使用方式参考：  
<http://www.jdon.com/product/ejbinvoker.htm>

### TargetMetaDef

TargetMetaDef 是目标服务的元定义，TargetMetaDef 主要实现有两种：EJBTargetMetaDef 和 POJOTargetMetaDef，分别是 EJB 服务和 POJO 服务实现。

所谓目标服务元定义也就是程序员在 jdonframework.xml 中定义的 services。

Jdon 框架提供目标服务两种形式：目标服务实例创建和目标服务本身。

### ServiceFactory

目标服务创建工厂 ServiceFactory 可以根据目标服务元定义创建一个目标服务实例，这也是 WebAppUtil.getService 获得目标服务的原理实现，

com.jdon.controller.service.DefaultServiceFactory 是 ServiceFactory 缺省实现，在 DefaultServiceFactory 中，主要是通过动态代理获得一个服务实例，使用了 DefaultServiceFactory 来实现动态代理对象的缓存。不必每次使用 Proxy.newInstance 第一次执行后，将其结果保存在 HttpSession 中。

### Service

目标服务 Service 则是通过方法 Relection 运行目标服务，只要告知 Service 目标服务的类、类的方法、方法参数类型和方法参数值这些定义，除了方法参数值，其余都是字符串定义，这是 Java 的另外一种调用方式。

## 框架组件服务使用

### 应用服务配置

jdonframework.xml 是应用服务组件配置文件，文件名可自己自由定义，jdonframework.xml 中主要是定义 Model（模型）和 Service（服务）两大要素。

jdonframework.xml 最新定义由 <http://www.jdon.com/jdonframework.dtd> 规定。

<models>段落是定义应用系统的建模，一个应用系统有哪些详细具体的模型，可由 Domain Model 分析设计而来。<models>中的详细配置说明可见 数据模型增、删、改、查 章节。

<services>段落是定义服务组件的配置，目前有两种主要服务组件配置：EJB 和 POJO。EJB 服务组件配置如下：

```
<ejbService name="newsManager">
    <jndi name="NewsManager" />
    <ejbLocalObject class="news.ejb.NewsManagerLocal"/>
</ejbService>
```

每个 ejbService 组件有一个全局唯一的名字，如 newsManager，有两个必须子定义：该 EJB 的 JNDI 名称和其 Local 或 remote 接口类。

POJO 服务组件配置如下：

```
<pojoService name="userJdbcDao" class="news.container.UserJdbcDao">
    <constructor value="java:/NewsDS"/>
</pojoService>
```

POJO 服务也必须有一个全局唯一名称，如 userJdbcDao，以及它的 class 类定义。如果该 POJO 构造器有字符串变量，可在这里定义其变量的值，目前 Jdon 框架只支持构造器字符串变量注射。

如果该 POJO 服务需要引用其它服务，例如 UserPrincipalImp 类的构造器如下：

```
public UserPrincipalImp(UserDao userDao){

    this.userDao = userDao;

} .....
```

UserPrincipalImp 构造器需要引用 UserDao 子类实现，只需在 jdonframework.xml 中同时配置这两个服务组件即可，Jdon 框架会自动配置它们之间的关系：

```
<pojoService name="userJdbcDao" class="news.container.UserJdbcDao">
    <constructor value="java:/NewsDS"/>
</pojoService>
<pojoService name="userPrincipal" class="news.container.UserPrincipalImp"/>
```

上面配置中 news.container.UserJdbcDao 是接口 UserDao 的子类实现，这样，直接通过 userPrincipal 这个名称可获得 UserPrincipalImp 的实例。

6.0 版本以后可使用@Service(“myName”)替代以上 XML 配置。

## 基础组件配置说明

container.xml 是 Jdon 框架基础组件配置文件，container.xml 中包含的组件将由 Jdon 框架在启动时向微容器（PicoContainer）中注册，至于这些组件之间的依赖关系由微容器解决，称为 Ioc 模式。

container.xml 内容主要由每行配置组成，每行格式如下：

```
<component name="组件名称" class="POJO 类名称" />
```

如

```
<component name="modelHandler" class="com.jdon.model.handler.XmlModelHandler" />
```

代表组件 com.jdon.model.handler.XmlModelHandler，其名称为 modelHandler，如果需要在程序中调用 XmlModelHandle 实例，只需要以 modelHandler 为名称从微容器中获取即可。

组件配置也可以带有参数，例如下行：

```
<component name="cache" class="com.jdon.controller.cache.LRUCache" >
    <constructor value="cache.xml"/>
</component>
```

而 LRUCache 的类代码如下：

```
public class LRUCache implements Cache {
    public LRUCache(String configFileName) {
        PropsUtil propsUtil = new PropsUtil(configFileName);
        cache = new UtilCache(propsUtil);
    }
    .....
}
```

这样 LRUCache 中的 configFileName 值就是 cache.xml，在 cache.xml 中定义了有关缓存的一些设置参数。目前 Jdon 框架只支持构造器是纯字符串型，可多个字符串变量，但不能字类型和其它类型混淆在一起作为一个构造器的构造参数。如果需要多个类型作为构造参数，可新建一个包含字符串配置类，这个类就可和其它类型一起作为一个构造器的构造参数了。

一般在 container.xml 中的组件是框架基本功能的类，不涉及到具体应用系统。

6.0 版本以后可使用 @Component(“myName”) 替代以上 XML 配置。

## 拦截器组件配置说明

本篇为高级应用，初学者可以以后再深入研究。

aspect.xml 是关于拦截器组件配置，有两个方面：advice(拦截器 **Interceptor**) 和 pointcut (切入点) 两个方面配置，有关 AOP 的基本概念可见：  
<http://www.jdon.com/AOPdesign/aspectJ.htm>。

Jdon AOP 的设计目前功能比较简单，不包括标准 AOP 中的 Mixin 和 Introduction 等功能；Pointcut 不是针对每个 class 和方法，而是针对一组 class（如 POJO 组或 EJB 组），拦截粒度最粗，与许多复杂完整的 AOP 框架（如 AspectJ、Spring）不同的是：Jdon AOP 在粒度方面是最粗的，AspectJ 最细，Spring 中等，如果你需要粒度细腻的 AOP 功能，还是推荐使用 Spring 或 AspectJ。目前这样设计是主要有两个原因：

每个类在运行时刻都实现动态拦截，在性能上有所损失，这如同职责链模式缺点一样。

在实际应用中，可以通过代理模式 Proxy、装饰模式 Decorator 实现一些细腻度拦截，结合容器的 Ioc 特性，这两个代理模式使用起来非常方便，运行性能有一定提高。

Jdon AOP 主要针对拦截器 **Interceptor** 设计，它可以为所有 jdonframework.xml 中定义的 Service 提供拦截器；所有的拦截器被放在一个拦截器链 **InterceptorsChain** 中。

Jdon AOP 并没有为每个目标实例都提供拦截器配置的功能，在 JdonAOP 中，目标对象是以组为单位，而非每个实例，类似 Cache/Pool 等这些通用拦截器都是服务于所有目标对象。

JdonAOP 拦截器目标对象组有三种：全部目标服务；EJB 服务；POJO 服务（EJB 服务和 POJO 服务是在 JdonFramework.xml 中定义的 ejbService 和 pojoService）。从而也决定了 Pointcut 非常简单。以下是 aspect.xml 中的配置：

```
<interceptor name="cacheInterceptor"
class="com.jdon.aop.interceptor.CacheInterceptor"
pointcut="services" />
```

其中 `pointcut` 有三种配置可选：`services` ; `pojoServices` 和 `ejbServices`

拦截器配置也可以如组件配置一样，带有 `constructor` 参数，以便指定有关拦截器设置的配置文件名。

拦截器的加入不只是通过配置自己的 `aspect.xml` 可以加入，也可以通过程序实现，调用 `WebAppUtil` 的 `addInterceptor` 方法即可，该方法只要执行一次即可。

## 如何实现自己的拦截器？

本篇为高级应用，初学者可以以后再深入研究。

以对象池拦截器 `PoolInterceptor` 为例，对象池是使用 `Aapche Commons Pool` 开源产品，对象池主要是为了提高 `POJO Service` 的运行性能，在没有对象池的情形下，`POJO Service` 每次被访问时，要产生一个新的实例，如果并发访问用户量很大，`JVM` 将会频繁创建和销毁大量对象实例，这无疑是耗费性能的。

使用对象池则可以重复使用一个先前以前生成的 `POJO Service` 实例，这也是 `Flyweight` 模式一个应用。

对象池如何加入到 `Jdon` 框架中呢？有两种方式：1.替代原来的 `POJO Service` 实例创建方式，属于 `PojoServiceFactory` 实现（`Spring` 中 `TargetSource` 实现）；2.通过拦截器，拦截在原来的 `PojoServiceFactory` 实现之前发生作用，同时屏蔽了原来的 `PojoServiceFactory` 实现。`Jdon` 框架采取的这一种方式。

首先，为拦截器准备好基础组件。对象池拦截器有两个：对象池 `com.jdon.controller.pool.CommonsPoolAdapter` 和 对象池工厂 `com.jdon.controller.pool.CommonsPoolFactory`，这两个实现是安装 `Apache Pool` 要求实现的。

第二步，需要确定拦截器的 `Pointcut` 范围。前面已经说明，在 `Jdon` 框架中有三个 `Pointcut` 范围：所有服务、所有 `EJB` 服务和所有 `POJO` 服务，这种划分目标的粒度很粗糙，我们有时希望为一些服务群指定一个统一的拦截器，例如，我们不想为所有 `POJO` 服务提供对象池，想为指定的一些目标服务（如访问量大，且没有状态需要保存的）提供对象池，那么如何实现呢？这实际是如何自由划分我们自己的目标群（或单个目标实例）的问题

我们只要制作一个空接口 `Poolable`，其中无任何方法，只要将我们需要对象池的目标类实现这个接口即可，例如 `com.jdon.security.web.AbstractUserPrincipal` 多继承一个接口 `Poolable`，那么 `AbstractUserPrincipal` 所有的子类都将被赋予对象池功能，所有子类实例获得是从对象池中借入，然后自动返回。

5.6 版本以后，增加 `Annotation @Poolable` 替代接口 `Poolable`。

这种通过编程而不是配置实现的 `Pointcut` 可灵活实现单个目标实例拦截或一组目标实例拦截，可由程序员自由指定划分，非常灵活，节省了琐碎的配置，至于 `Pointcut` 详细到类方法适配，可在拦截器中代码指定，如缓存 `com.jdon.aop.interceptor.CacheInterceptor` 只拦截目标服务类中的 `getXXXX` 方法，并且该方法的返回结果类型属于 `Model` 子类，为了提高性能，`CacheInterceptor` 将符合条件的所有方法在第一次检查合格后缓存起来，这样，下次无需再次检查，省却每次检查。

第三步，确定拦截器在整个拦截器链条中的位置。这要根据不同拦截器功能决定，对象池拦截器由于是决定目标服务实例产生方式，因此，它应该最后终点，也就是在拦截器链中最后一个执行，`aspect.xml` 中配置拦截器是有先后的：

```
<interceptor name="cacheInterceptor"
class="com.jdon.aop.interceptor.CacheInterceptor" pointcut="services" />
```



```
<interceptor name="poolInterceptor"
class="com.jdon.aop.interceptor.PoolInterceptor" pointcut="pojoServices" />
```

拦截器链中排序是根据 `Interceptor` 的 `name` 值排序的，`cacheInterceptor` 第一个字母是 `c`，而 `poolInterceptor` 第一个字母是 `p`，按照字母排列顺序，`cacheIntercepotr` 排在 `poolInterceptor` 之前，在运行中 `cacheInterceptor` 首先运行，在以后增加新拦截器时，要注意将 `poolInterceptor` 排在最后一个，`name` 值是可以任意指定的，如为了使 `PoolInterceptor` 排在最后一个，可命名为 `zpoolInterceptor`，前面带一个 `z` 字母。

第四步，确定拦截器激活行为是在拦截点之前还是之后，或者前后兼顾，这就是 `advice` 的三种性质：`Before`、`After` 或 `Around`，这分别是针对具体拦截点 `jointcut` 位置而言。

虽然 `Jdon` 框架没有象 `Spring` 那样提供具体的 `BeforeAdvice` 和 `AfterReturningAdvice` 等接口，其实这些都可以有程序员自己直接实现。

`Jdon` 框架的拦截器和 `Spring` 等遵循 `aopalliance` 的 `AOP` 框架继承同一个接口 `MethodInterceptor`，也就是说，一个拦截器在这些不同 `AOP` 框架之间可以通用，具体实现方式不同而已。

例如，一个 `AroundAdvice` 实现如下代码，它们都只需要完成 `invoke` 方法内尔：

```
public class AroundAdvice    implements MethodInterceptor
{
    public Object invoke( MethodInvocation invocation)    throws Throwable
    {
        System.out.println("Hello world! (by " + this.getClass().getName() + ")");
        invocation.proceed();
        System.out.println("Goodbye! (by " + this.getClass().getName() + ")");
        return null;
    }
}
```

`beforeAdvice` 实现代码如下：

```
public class BeforeAdvice    implements MethodInterceptor
{
    public Object invoke( MethodInvocation invocation)    throws Throwable
    {
        System.out.println("Hello world! (by " + this.getClass().getName() + ")");
        invocation.proceed();
    }
}
```

由此可以注意到，`invocation.proceed()`类似一个 `jointcut` 点，这个方法类似 400 米接力比赛中的传接力棒，将接力棒传到下一个拦截器中，非常类似与 `servletFilter` 中的 `chain.doFilter(request, response)`；拦截器一些更详细说明可参考 `Spring` 相关文档，如下面网址：<http://www.onjava.com/pub/a/onjava/2004/10/20/springaop2.html>，和 `Jdon` 拦截器原理基本一致。

考察对象池拦截器功能，它实际是一个 `around advice`，在 `jointcut` 之前需要从对象池借用一个目标服务实例，然后需要返回对象池。`com.jdon.aop.interceptor.PoolInterceptor` 主要核心代码如下：

```
Pool pool = commonsPoolFactory.getPool(); //获得对象池
Object poa = null;
Object result = null;
try {
    poa = pool.acquirePoolable(); //借用一个服务对象
```

```

        Debug.logVerbose(" borrow a object:" + targetMetaDef.getClassName()
            + " from pool", module);

        //set the object that borrowed from pool to MethodInvocation
        //so later other Interceptors or MethodInvocation can use it!
        proxyMethodInvocation.setThis(poa); //放入 invocation，以便供使用
        result = invocation.proceed();
    } catch (Exception ex) {
        Debug.logError(ex, module);
    } finally {
        if (poa != null) {
            pool.releasePoolable(poa); //将服务对象归还对象池
            Debug.logVerbose(" realease a object:" + targetMetaDef.getClassName()
                + " to pool", module);
        }
    }
}
return result;

```

经过上述四步考虑，我们基本可以在 Jdon 框架动态 plugin 自己的拦截器，关于对象池拦截器有一点需要说明的，首先 EJB 服务因为有 EJB 容器提供对象池功能，因此不需要对象池了，在 POJO 服务中，如果你的 POJO 服务设计成有状态的，或者你想让其成为单例，就不能使用对象池，只要你这个 POJO 服务类不继承 Poolable 或没有 @Poolable 注释，它的获得是通过组件实例方式获得，参考后面“组件实例和服务实例”章节。

自己实现的拦截器是在 myaspect.xml 中配置，至于如何将 myaspect.xml 告诉 Jdon 框架，可见[启动自定义框架组件](#)一节。

6.2 版本以后可使用 @Interceptor(“myName”)替代以上 XML 配置。

## 组件替代

我们前面说过，Jdon 框架的一个最大特点是实现了对象的可替代性，Jdon 框架应用系统组件和框架本身组件都是可以更换的，这样，程序员可以根据自己开发的具体特点，通过编制自己的组件，丰富 Jdon 框架的一些功能。

那么自己编制的组件如何放入 Jdon 框架，让 Jdon 框架识别呢？

通过前面介绍已经可能知道，通过修改 container.xml 或 aspect.xml 这样的配置文件即可。

一般情况下，container.xml 是包含在 Jdon 框架的压缩包 jdonFramework.jar 中，不利于修改，我们可以用 winrar 等解压工具将 jdonFramework.jar 中 META-INF 目录下的 container.xml 或 aspect.xml 解压出来，解压出来的 container.xml 也必须放置在系统的 classpath 中，当然，你修改 container.xml 以后，再将修改后的 container.xml 通过 winrar 工具再覆盖 jdonFramework.jar 中原来的 container.xml 也可以。

具体修改办法，以 container.xml 下面一句为例子：

```
<component name="sessionContextSetup" class="com.jdon.security.web.HttpRequestUserSetup" />
```

其中 com.jdon.security.web.HttpRequestUserSetup 是 SessionContextSetup 接口的子类，如果我们要实现自己的 SessionContextSetup 子类，例如为 MyHttpRequestUserSetup，那么编制好 MyHttpRequestUserSetup 后，将配置中替代为 MyHttpRequestUserSetup 即可：

```
<component name="sessionContextSetup" class="com.mylib.MyHttpRequestUserSetup" />
```

注意，我们自己子类 MyHttpRequestUserSetup 也必须放置在系统的 classpath，最简单

的办法是：将 MyHttpRequestUserSetup 等类也打包成.jar 包，和 jdonFramework.jar 一起部署。

如果你需要增加自己的组件，那么只要定义 mycontainer.xml 和 myaspect.xml 即可，然后将这两个配置通过[启动配置](#)告诉 Jdon 框架。

## 组件实例和服务实例

在 Jdon 框架中， POJO 实例分为两种性质：组件实例和服务实例。

组件实例是指那些注册在容器中的普通 Javabeans，它们是单例的，也就是你每次获得的组件实例都是同一个实例，也就是说单态的。

组件 POJO 获得方法是通过 WebAppUtil 的 getComponentInstance 方法，例如在 container.xml 中有如下组件定义：

```
<component name="modelManager" class="com.jdon.model.ModelManagerImp" />
```

在程序中如果要获得 ModelManagerImp 组件实例的方法是：

```
ModelManager modelManager =
```

```
(ModelManager)WebAppUtil.getComponentInstance("modelManager", sc);
```

组件实例获得的原理实际是直接在微容器中寻找以前注册过的那些 POJO，相当于直接从 XML 配置中直接读取组件实例配置。

服务实例是指在 jdonframework.xml 中定义的 ejbService 和 pojoService，当然它们也可以组件实例方式获得，但是如果以组件实例方式获得，AOP 功能将失效；而以服务实例方式获得的话，在 aspect.xml 中定义的拦截器功能将激活。

EJB 服务一定是通过服务实例方式获得，只有普通 JavaBeans（POJO）才可能有这两种方式。

因此，除非特殊需要，一般推荐在应用系统中，通过获得服务实例方式来获得 jdonframework.xml 中定义的服务实例。

如在 jdonframework.xml 中有如下服务组件定义：

```
<pojoService name="userJdbcDao" class="news.container.UserJdbcDao">
```

```
    <constructor value="java:/NewsDS"/>
```

```
</pojoService>
```

获得服务实例的方法代码如下：

```
UserDao ud = (UserDao)WebAppUtil.getService("userJdbcDao", request);
```

UserDao 是 UserJdbcDao 的接口。注意，这里必须 getService 结果必须下塑为接口类型，不能是抽象类或普通类，这也是与 getComponentInstance 不同所在。

如果你在 aspect.xml 将 pojoServices 都配置以对象池 Pool 拦截器，那么上面代码将是 从对象池中获取一个已经事先生成的实例。

如何获得一个 POJO 服务实例？

在上面章节说明了服务实例和组件实例的区别，在 jdonframework.xml 中配置 POJO 既可以组件实例获得，也可以服务实例获得。

首先，我们需要在 jdonframework.xml 中定义自己的 POJO 服务，例如在 jdonframework.xml 有如下一行定义：

```
<pojoService name="userJdbcDao" class="news.container.UserJdbcDao">
```

那么在应用程序中需要访问 UserJdbcDao 实例有以下两种方式：

第一. 通过 WebAppUtil 工具类的 getService 方法获得服务实例，如：

```
UserDao ud = (UserDao)WebAppUtil.getService("userJdbcDao", request);
```

getService 方法每次返回的一个新的服务实例对象，相当于 new 一对象。如果对象池拦截器被配置，那么这里返回的就是从对象池中借用的一个对象。

第二. 通过 WebAppUtil 工具类的 getComponentInstance 方法获得组件实例，这也是获得 UserJdbcDao 一个实例，与服务实例不同的是，每次获得组件实例是同一个对象，因此，如果这个服务中如果包含上次访问的状态或数据，下次访问必须使用到这些状态和数据，那么就必须使用 getComponentInstance 方法获得服务实例。

注意，以上方式是假定你获得一个 POJO 实例，是为了使用它，也就是说，是为了访问它的方法，如访问 userJdbcDao 的 getName 方法，就要使用上述方式。

如果你不是为了使用它，而是作为别的 POJO 服务的输入参数，如构造器的输入参数，那么完全不必通过上述方式，你只要直接使用上述方式获得那个 POJO 服务的实例就可以，因为容器自动完成它们的匹配。

还有一点要求注意的是：使用 getService 获得服务实例，必须该服务类有一个接口，这样才能将 getService downcasting 下塑为其接口，否则只能是一个普通 Object，你获得后无法使用它。当然 getComponentInstance 没有这样限制。

如何编写一个 POJO 类？

既然在 Jdon 框架中获得 POJO 服务这么方便，那么 POJO 服务类的编写是否有特殊规定，回答是没有，就是一个普通的 Java 类，当然如果你需要在这个类引用其他类，最好将其他类作为构造器参数，如 A 类中引用 B 类，A 类的写法如下：

```
class A {  
    private B b;  
    public A(B b){  
        this.b = b;  
    }  
    ....  
}
```

这样，在 jdonframework.xml 中配置如下两行：

```
<pojoService name=" a" class="A">  
<pojoService name=" b" class="B">
```

如果你希望你的 POJO 服务能够以对象池形式被访问，那么你的类需要 implements com.jdon.controller.pool.Poolable 或者在类前加入 @Poolable 元注解 Annotation

例如 上面的 A 继承了 Poolable 或加入了 @Poolable 元注解 Annotation，那么当有并发两个以上客户端访问服务器时，对象池将同时提供两个以上 A 实例为客户端请求服务，提高了并发访问量。

注意，当 A 的实例同时为两个以上时，A 引用的 B 是否也是每个 A 实例拥有一个 B 实例呢？也就是说，B 实例是否也是两个以上？

这取决于 A 对 B 的调用方式，如果 A 和 B 都在 jdonframework.xml 中配置，也就是都被注册到 Jdon 容器中，那么此时 B 永远只有一个，也就是单例的。

如果 B 的创建是在 A 内部，如下：

```
class A {
```

```
private B b = new B();
....
}
```

这样，B 实例的创建是跟随 A 实例一起创建，因此，A 有多少个，B 就有多少个。

### 如何获得一个 POJO 服务的运行结果？

在应用系统中，我们不但可以通过[“如何获得一个 POJO 服务实例”](#)获得一个 POJO 服务实例，然后在通过代码调用其方法，获得其运行结果，例如写入代码：

```
UserDao ud = (UserDao)WebAppUtil.getService("userJdbcDao", request);
userJdbcDao.getName(); //获得一个 POJO 服务的运行结果
```

除此之外，Jdon 框架还可以直接获得 POJO 服务的运行结果，只要你告诉它 POJO 类名、需要调用的方法名和相关方法参数类型和值，借用 Java Method Reflection 机制，Jdon 框架可以直接获得运行结果。

实现这个功能，只要和接口 `com.jdon.controller.service.Service` 打交道即可：

```
public interface Service {

    public Object execute(String name,
                           MethodMetaArgs methodMetaArgs,
                           HttpServletRequest request) throws Exception;

    public Object execute(TargetMetaDef targetMetaDef,
                           MethodMetaArgs methodMetaArgs,
                           HttpServletRequest request) throws Exception;

}
```

Service 提供了两种获得某个 POJO 服务运行结果的方法。一个是以 Jdonframework.xml 中配置的 POJO 服务名称为主要参数，这是经常使用的一个情况。

MethodMetaArgs 是包含调用的方法名、方法参数类型和方法参数值。

这种调用方式适合于 POJO 服务配置式调用，也就是说，通过编写自己的 XML 配置文件也可以实现如同写代码一样的服务调用和运行。

## 事件模型实现

本章主要介绍如何使用 Jdon 框架实现领域事件 Domain Events，从这些事件实现中可以发现事件模型的优点和特点。

## Domain Events 开发

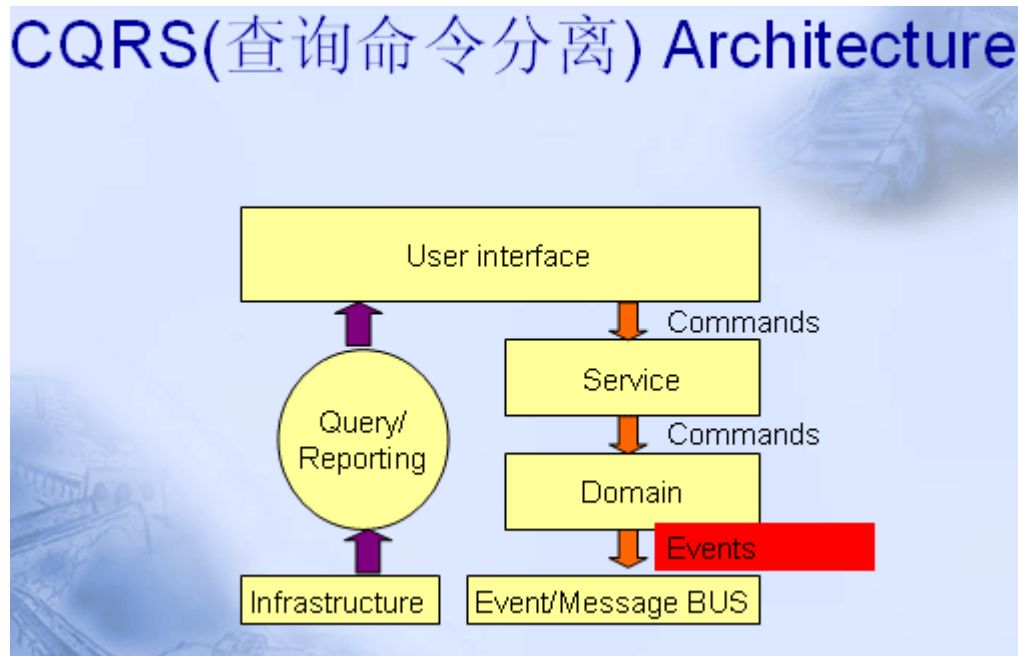
在 Evans DDD 实现过程中，经常会碰到实体和服务 Service 以及 Repository 交互过

程，这个交互过程的实现是一个难点，也是容易造成失血贫血模型的主要途径。

领域模型中只有业务，没有计算机软件架构和技术。不要将和技术相关的服务和功能组件注射到实体模型中，例如数据库 Dao 等操作。由领域模型通过 Domain Events 机制指挥 Domain 服务和其他功能组件，包括数据库操作。

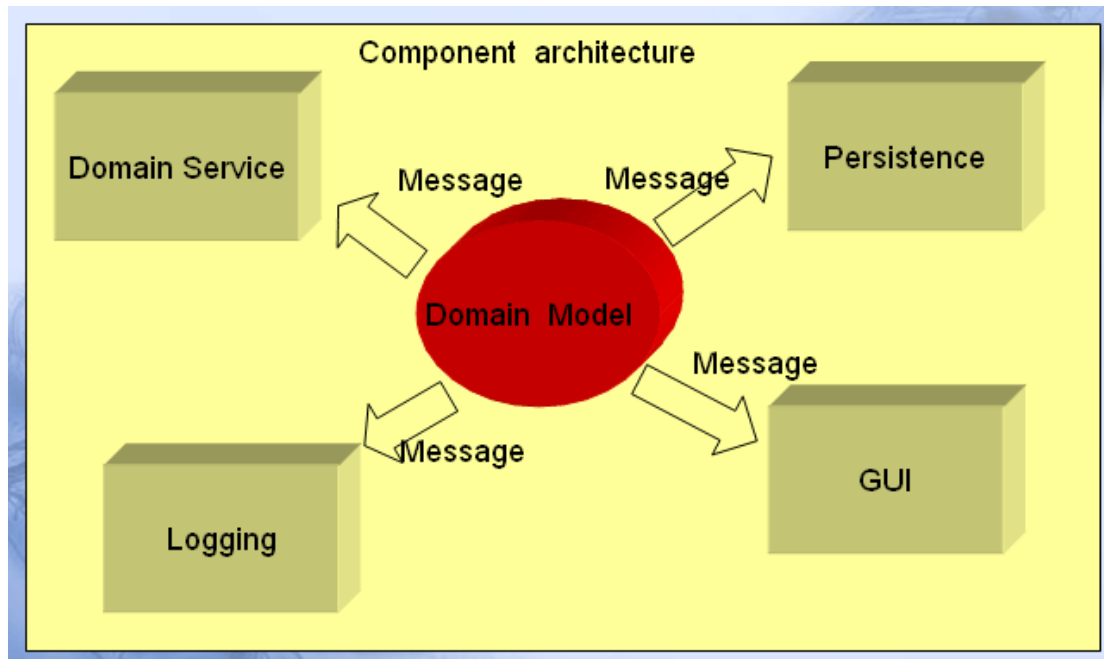
<http://jonathan-oliver.blogspot.com/search/label/DDD>

使用 DDD，常常需要[命令查询分离模式](#) CQS 架构，如下图：



图中 Domain 发出的 Events 就称为 Domain Events, 如果说 CQS 架构提出将 Command 命令和查询分离的模式，那么随着 Command 命令产生 Domain Events 则提出业务模型和技术架构分离的解决方案。

JF 提供的异步观察者模式为 Domain Event 实现提供更优雅的方案。详细文章见：  
<http://www.jdon.com/jivejdon/thread/37289>



## 让模型常驻内存

在开始 DomainEvents 开始之前，必须保证领域模型在内存中，分两步：

1. 使用 @Model 标注你的领域模型，如 User 是从需求中分析出的一个领域模型类：

@Model

```
public class User {  
    private String userId;  
    private String name;  
    ....  
}
```

源码见：

当领域模型类使用 @Model 标注后，意味着这个模型将会驻留在内存中 in memory, 缺省 Jdon 框架支持的是 Ehcache. 你也可以设置为 Key-value store 或其他内存数据网格 IMDG, [hazelcast 替代 JF 缺省的 ehcache](#)。

2. 虽然第一步标注了，但是模型不会自动驻留内存，还需要你在自己的应用代码中显式表明一下，方法是：在仓储 Respository 层，也就是封装了数据库 SQL 操作的层面，因为我们的领域模型一般都是从数据库获得，在仓储层进行组装，将一个数据表数据转换成领域对象，那么在这个转换性质的类和方法上要加一个元注释 @Introduce(“modelCache”) 和 @Around。

这一步确保领域对象每次加载在内存都是唯一的，In-memory。这对于后面使用 domain events 是必须关键重要步骤。



```

@Component("mymrepository")
@Introduce("modelCache")
public class RepositoryImp implements MyModel {

    @Around
    public MyModel getModel(Long key) {
        MyModel mym = new MyModel();
        mym.setId(key);
        return mym;
    }
}

```

Auto cache MyModel

这步源码见: [com.jdon.sample.test.domain.onecase.repository.RepositoryImp](http://com.jdon.sample.test.domain.onecase.repository.RepositoryImp)

下面开始 Domain Events 开发步骤, 分两步:

Domain Events 自 6.4 版本以后, 有两种使用方式, 一种是 6.3 版本以前的使用 JDK 并发包 `futurtask` 实现的方式; 一种是 6.4 版本以后新增引入最快并发框架 `Disruptor` 的 `@Consumer` 方式。首先谈一下这种新的方式:

### 消息生产者开发

第一步: 消息发送者的开发, 与 6.3 版本以前类似:

首先: 使用 `@Model` 和 `@Introduce("message")` 标注实体类。

然后: 使用 `@Send("mytopic")` 标注该实体中的发送方法。

如下:

```

@Model
@Introduce("message")
public class DomainEvent {
    private Long id;
    private String name;

    @Send("mychannel")
    public DomainMessage myMethod() {
        DomainMessage em =
            new DomainMessage(this.name);
        return em;
    }
}

```

源码见框架包中 [com.jdon.sample.test.domain.onecase.DomainEvent](http://com.jdon.sample.test.domain.onecase.DomainEvent)

注意点: `@Introduce("message")` 中 “message” 值表示引入 JF 配置 `aspect.xml` 中消息拦截器: `om.jdon.domain.message.MessageInterceptor`, 通过该配置让该模型类成为消息生产者或发送者。

`@Send("mytopic")` 中的 “mytopic” 是消息 topic 名称, 可自己取, 但是和消费者的标注 `@Consumer("mytopic")` 或 `@Compnent("mytopic")` 是一致的, 表示生产者将消息发往这个 topic 中;



在@send 标注的方法中，你还需要将你要传送给消息消费者使用的数据打包进入 DomainMessag 对象，该方法的返回类型必须是 DomainMessag.

## 消息消费者开发

第二步：消息消费者开发，分两种，@Consumer 和@Component

@Consumer; 可以实现 1:N 多个，内部机制使用号称最快的并发框架 Disruptor 实现。适合轻量；小任务；原子性；无状态。

@Componet; 直接使用普通组件类作为消费者，使用 jdk future 机制，只能 1:1，适合大而繁重的任务，有状态，单例。

@Consumer 和@Component 大部分情况下可互换使用。

@Consumer 方式：

标注消费者 @Consumer("mytopic"); 消费者类必须实现接口 com.jdon.domain.message.DomainEventHandler。

```
@Consumer("mychannel")
public class MyDomainEventHandler implements DomainEventHandler
{
    public void onEvent(EventDisruptor event, boolean endOfBatch)
        throws Exception {
        System.out.println("DomainEventHandler action " +
            event.getDomainMessage().getEventSource());
    }
}
```

源码见 [com.jdon.sample.test.domain.onecase.MyDomainEventHandler](#)

如果有多个消费者订阅了同一个主题 Topic，那么这些消费者之间的运行顺序是按照他们的类名字母先后的。如 AEventHandler 先于 BEventHandler 先于 CEvent…等。

消费者的主要内容写在 onEvent 中，可通过 event.getDomainMessage()获得生产者你要传递的数据。

以上两个步骤的领域事件@Consumer 消费者开发完成。

## 返回事件处理结果

如果要返回事件的处理结果，在 DomainEventHandler 的 onEvent 方法 将结果设置如 event 的 DomainMessage 中，如下：

```
void onEvent(EventDisruptor event, boolean endOfBatch) throws Exception {
    //返回结果 “eventMessage=hello”
    event.getDomainMessage().setEventResult("eventMessage=" + myModel.getId());
}
```

JdonFramework 客户端可以是在 Web 的 Servlet 或 Jsp 中调用，也可以在 Application 调用，调用以上领域事件代码如下：

```
ContainerSetupScript css = new ContainerSetupScript();
css.prepare("com.jdon.jdonframework.xml", da);
AppUtil appUtil = new AppUtil("com.jdon.jdonframework.xml");

IServiceSample serviceSample = (IServiceSample) appUtil.getService("serviceSample");
//返回结果 "eventMessage=hello"
String res = (String) serviceSample.eventPointEntry("hello");
Assert.assertEquals(res, "eventMessage=hello");
```

源码见:[com.jdon.SampleAppTest](http://com.jdon.SampleAppTest)

### @Component 消费者

下面谈一下 6.3 以前版本但是 6.4 以后延续使用的 1:1 队列@Component 方式的开发，@Component 方式适合大任务，繁重计算，不必返回结果，是单例，而@Consumer 非线程方式：每次启动都启动新的实例线程。

1. 创建模型类如 UserModel(需要 @Model 标注)，在 UserModel 引入自己的 UserDomainEvents

@Inject

private UserDomainEvents userDomainEvents;

2. 创建 UserDomainEvents 类

3. 创建 com.jdon.domain.message.MessageListener 实现子类

如下图所示步骤：

该案例代码可在 <http://www.jdon.com/jdonframework/SimpleJdonFrameworkTest.rar> 下载。

```

@Model
public class UserModel {

    private String userId;
    private String name;

    private UserCountValueObject ageVO;

    @Inject
    private UserDomainEvents userDomainEvents;

    public void update(UserModel userParameter) {
        this.name = userParameter.getName();

        ageVO.preloadData();
        userDomainEvents.save(this);
    }
}

@Introduce("message")
public class UserDomainEvents {

    @Send("saveUser")
    public DomainMessage save(UserModel user) {
        return new DomainMessage(user);
    }
}

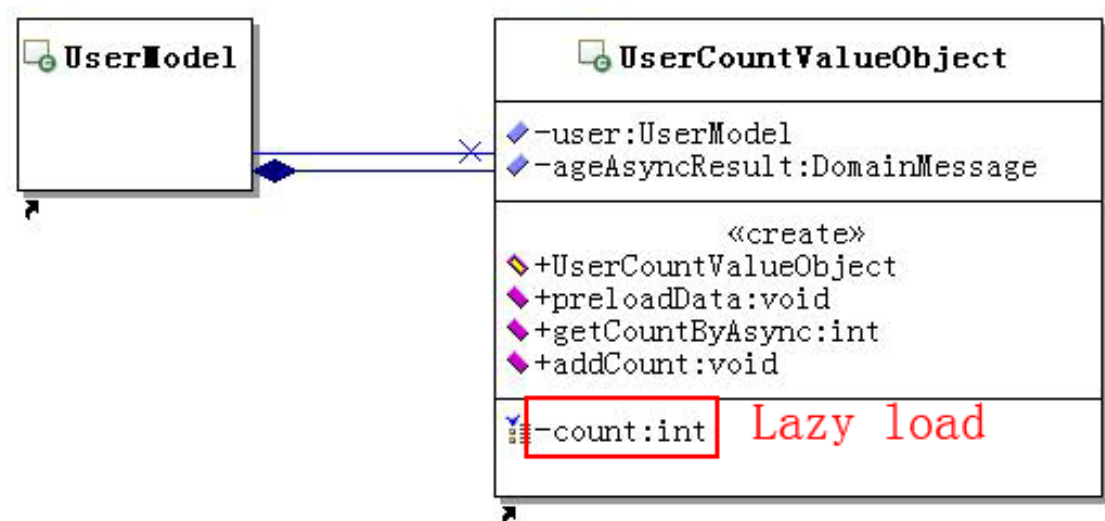
@Component("computeCount")
public class ComputeCountListener implements MessageListener {

    public void action(DomainMessage message) {
        UserModel user = (UserModel) message.getEvent();
        int age = userRepository.getAge(user.getUserId());
    }
}

```

使用 Domain Events 可实现异步懒加载机制，对模型中任何字段值根据需从数据库加载，即用即取，这种方式有别于 Hibernate 的惰加载 lazy load 机制，更加灵活。

如下图，我们为了实现 UserCount 值对象中 count 数据的懒加载，将其封装到一个值对象中，这个值对象其他字段和方法都是为懒加载服务的。



我们看看 getCountByAsync 方法内部：

```

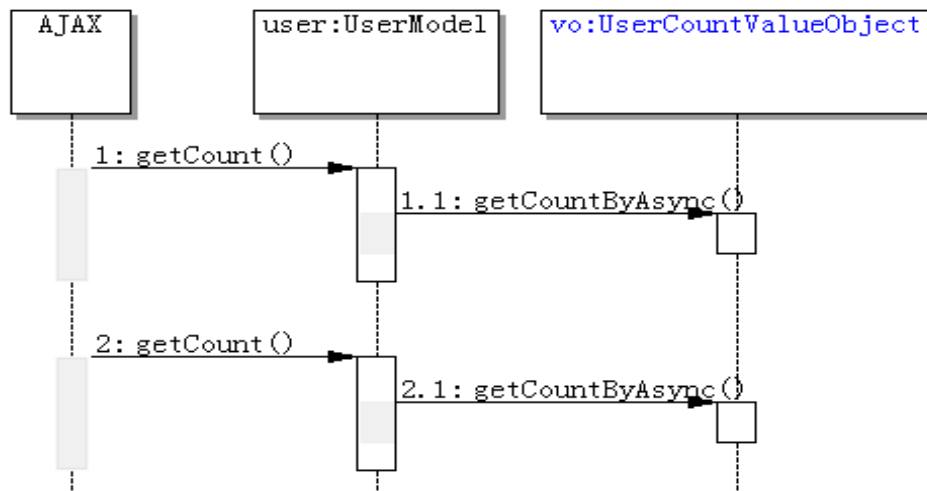
public int getCountByAsync() {
    if (count == -1) { // lazy load
        if (ageAsyncResult == null)
            ageAsyncResult = user.getUserDomainEvents().computeCount(user);
        else
            count = (Integer) ageAsyncResult.getEventResult();
    }
    return count;
}

```

Count 初始值是-1，如果该方法第一次调用，count 必为-1，那么 ageAsyncResult 也应该为空，这时就激活 computeCount 这个耗时耗 CPU 的任务，当第二次访问 getCountByAsync 时，ageAsyncResult 应该不为空，因为耗时任务应该计算完毕，如果还没有，会在这里等待。一旦获得计算结果，count 就不会为-1。

UserCount 值对象随同 UserModel 生活在内存中，因此，这样耗时任务计算只需要一次，以后，基本直接从对象的字段 count 直接获得。

调用顺序图如下：



## 懒加载 Laziness

Account 中有一个计算该用户发帖总数字段 messageCount:

```

public class Account{

    private int messageCount;

    public int getMessageCount(){
        return messageCount;
    }

}

```

这个 messageCount 是通过查询该用户所有发帖数查询出来的，也许你会问，为什么不将用户发帖总数设置为数据表一个字段，这样用户发帖时，更新这个字段，这样只要直接查询这个字段就可以得到 messageCount？

没有设立专门持久字段的原因如下：

1. 从模型设计角度看：messageCount 字段归属问题，messageCount 其实是一个表示 Account 和 Message 两个实体的关系关联统计字段，但是这种关联关系不属于高聚合那种组合关系的关联，不是 Account 和 Message 的必备属性，根据 DDD 的高聚合低关联原则，能不要的关联就不要，因此放在哪里都不合适。

2. 从性能可伸缩性角度看：如果我们将 messageCount 放在第三方专门关联表中，那么用户增加新帖子，除了对 Message 对应的表操作外，还需要对这个关联表操作，而且必须是事务的，事务是反伸缩性的，性能差，如果象 messageCount 各种统计字段很多，跨表事务边界延长，这就造成整体性能下降。

3. 当然，如果将 messageCount 硬是作为 Account 表字段，由于整个软件的业务操作都是 Account 操作的，是不是将其他业务统计如 threadCount 等等都放到 Account 表中呢？这会造成 Account 表变大，最终也会影响性能。

那么 messageCount 每次都通过查询 Message 对应表中用户所有发帖数获得，这也会导致性能差，表中数据越多，这种查询就越费 CPU。

使用缓存，因为 Account 作为模型被缓存，那么其 messageCount 值将只有第一次创建 Account 执行查询，以后就通过缓存中 Account 可直接获得。

所以，根据 DDD，在 AccountRepository 或 AccountFactory 实现数据表和实体 Account 的转换，Account 中的值都是在这个类中通过查询数据表获得的。

当访问量增加时，这里又存在一个性能问题，虽然一个 Account 创建时，messageCount 查询耗时可能觉察不出，但是如果是几十几百个 Account 第一次创建，总体性能损耗也是比较大的，鉴于我们对可伸缩性无尽的追求，这里还是有提升余地。

从设计角度看，由于 messageCount 不是 Account 的必备字段，因此，不是每次创建 Account 时都要实现 messageCount 的赋值，可采取即用即查方式。所以，我们需要下面设计思路：

```
public class Account{

    private int messageCount = -1;

    public int getMessageCount(){
        if(messageCount == -1)
            //第一次使用时即时查询数据表
            return messageCount;
        }

    }
```

怎么实现这个功能呢？使用 Hibernate 的懒加载？使用 Lazy load 需要激活 Open

Session In View，否则如果 Session 关闭了，这时客户端需要访问 messageCount，就会抛 lazy Exception 错误，但是 OSIV 只能在一个请求响应范围打开，messageCount 访问可能不是在这次请求中访问，有可能在后面请求或其他用户请求访问，所以，这个懒加载必须是跨 Session，是整个应用级别的。

实际上，只要 Account 保存在缓存中，对象和它的字段能够跨整个应用级别，这时，只要在 messageCount 被访问即时查询数据表，就能实现我们目标，其实如此简单问题，因为考虑 Hibernate 等 ORM 框架特点反而变得复杂，这就是 DDD 一直反对的技术框架应该为业务设计服务，而不能成为束缚和障碍，这也是一山不容二虎的一个原因。

这带来一个问题，如何在让 Account 这个实体对象中直接查询数据库呢？是不是直接将 AccountRepository 或 AccountDao 注射到 Account 这个实体呢？由于 AccountDao 等属于技术架构部分，和业务 Account 没有关系，只不过是支撑业务运行的环境，如果将这么多计算机技术都注射到业务模型中，弄脏了业务模型，使得业务模型必须依赖特定的技术环境，这实际上就不是 POJO 了，POJO 定义是不依赖任何技术框架或环境。

POJO 是 Martin Fowler 提出的，为了找到解决方式，我们还是需要从他老人家方案中找到答案，模型事件以及 Event 模式也是他老人家肯定的，这里 Account 模型只需要向技术环境发出一个查询 Event 指令也许就可以。

那么，我们就引入一个 Domain Events 对象吧，以后所有与技术环境的指令交互都通过它来实现，具体实现中，由于异步 Message 是目前我们已知架构中最松耦合的一种方案，所以，我们将异步 Message 整合 Domain Events 实现，应该是目前我们知识水平能够想得到的最好方式之一，当然不排除以后有更好方式，目前 JdonFramework 6.2 已经整合了 Domain Events + 异步消息机制，我们就可以直接使用。

这样，Account 的 messageCount 即用即查就可以使用 Domain Events + 异步消息实现：

```
public int getMessageCount(){
    if (messageCount == -1) {
        if (messageCountAsyncResult == null) {
            //向技术环境发出查询获得 messageCount 值的命令，
            //这个命令是在另外新线程实现，因此结果不一定立即返回
            messageCountAsyncResult =
domainEvents.computeAccountMessageCount(account.getUserIdLong());
        } else {
            //当客户端再次调用本方法时，可以获得查询结果，
            //如果查询过程很慢，还是没有完成，会在这里堵塞等待，但概率很小
            messageCount = (Integer) messageCountAsyncResult.getEventResult();
        }
    }
}
```

messageCount 最后获得，需要通过两次调用 getMessageCount 方法，第一次是激活异步查询，第二次是获得查询结果，在 B/S 架构中，一般第二次查询是由浏览器再次发出请求，这浏览器服务器一来一回的时间，异步查询一般基本都已经完成，这就是充分利用 B/S 架构的时间差，实现高效率的并行计算。

所以，并不是一味死用同步就能提高性能，可伸缩性不一定是指单点高性能，而是指整个系统的高效率，利用系统之间传送时间差，实现并行计算也是一种架构思路。这种思

考思路在实际生活中也经常会发生。

最后，关于 `messageCount` 还有一些有趣结尾，如果浏览器不再发第二次请求，那么浏览器显示 `Account` 的 `messageCount` 就是-1，我们可以做成不显示，也就看不到 `Account` 的发帖总数，如果你的业务可以容忍这个情况，比如目前这个论坛就可以容忍这种情况存在，`Account` 的 `messageCount` 第一次查询会看不到，以后每次查询就会出现，因为 `Account` 一直在缓存内存中。

如果你的业务不能容忍，那么就在浏览器中使用 `AJAX` 再次发出对 `getMessageCount` 的二次查询，那么用户就会每次

都会看到用户的发帖总数，`JiveJdon` 这个论坛的标签关注人数就是采取这个技术实现的。这样浏览器异步和服务器端异步完美结合在一起，整个系统向异步高可伸缩性迈进一大步。

更进一步，有了 `messageCount` 异步查询，如何更新呢？当用户发帖时，直接对内存缓存中 `Account` 更新加一就可以，这样，模型操作和数据表操作在 `DDD + 异步架构` 中完全分离了，数据表只起到存储作用(`messageCount` 甚至没有专门的存储数据表字段)，这和流行的 `NoSQL` 架构是同一个思路。

由于针对 `messageCount` 有一些专门操作，我们就不能直接在 `Account` 中实现这些操作，可以使用一个专门值对象实现。如下::

```
public class AccountMessageVO {

    private int messageCount = -1;

    private DomainMessage messageCountAsyncResult;

    private Account account;

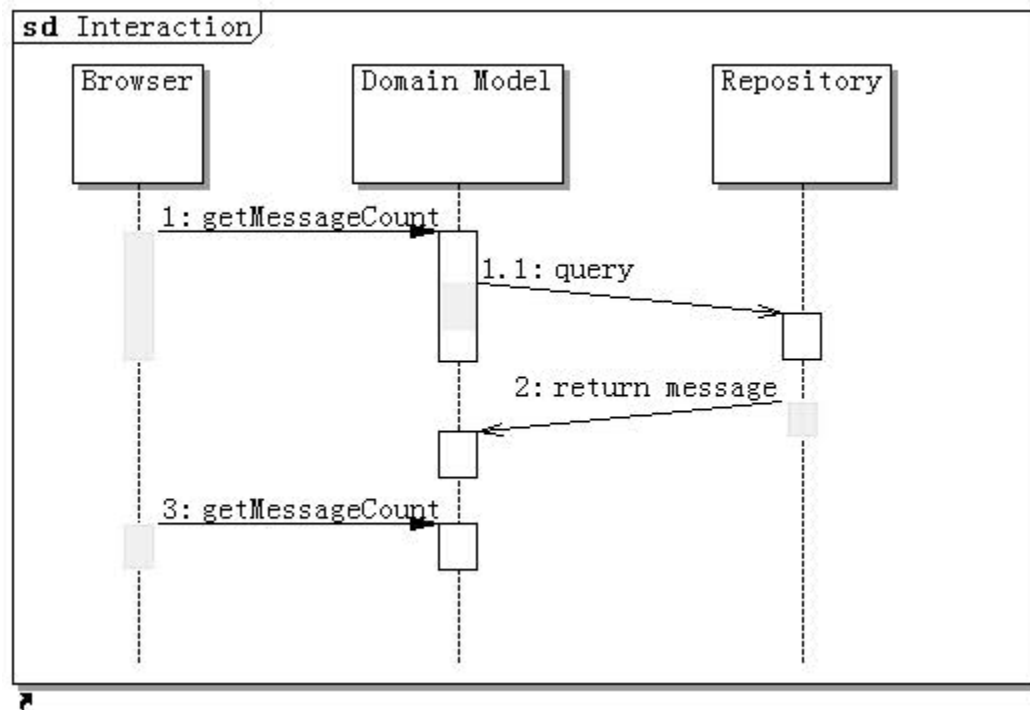
    public AccountMessageVO(Account account) {
        super();
        this.account = account;
    }

    public int getMessageCount(DomainEvents domainEvents) {
        if (messageCount == -1) {
            if (messageCountAsyncResult == null) {
                messageCountAsyncResult =
                domainEvents.computeAccountMessageCount(account.getUserIdLong());
            } else {
                messageCount = (Integer) messageCountAsyncResult.getEventResult();
            }
        }
        return messageCount;
    }

    public void update(int count) {
        if (messageCount != -1) {
            messageCount = messageCount + count;
        }
    }
}
```

```
}  
}
```

调用顺序图如下：



## 即用即加载

比如图片，只有显示图片时才会需要图片的二进制数据，这个数据比较大，所以，一般从持久层加载图片时，只加载图片其他核心文字信息，如图片 ID，名称等，当需要显示时，再加载二进制数据输出真正图片。

```
public byte[] getData() {  
    byte[] bytes = super.getData();  
    if (bytes != null)  
        return bytes;  
    preloadData();  
    bytes = (byte[]) imgDataAsyncResult.getEventResult();  
    this.setData(bytes);  
    return bytes;  
}  
  
//预加载 可在 JSP 即将显示图片之前发出事件激活该方法  
public void preloadData() {  
    if (imgDataAsyncResult == null && domainEvents != null)
```



```
imgDataAsyncResult = this.domainEvents.loadUploadEntity(this);  
}
```

传统意义上，懒加载和异步都是好像不被人接受的，会带来比较差的性能，高延迟性，属于边缘技术，这其实是被误导了：并发策略可以解决延迟

懒加载和异步代表的并发策略实际是一种潮流趋势，特别是作为并行计算语言 Scala 和 erlang 的新亮点：函数式编程 functional programming 的特点

而最新发布 JiveJdon3.9 使用传统 Java，基于 Jdon 框架 6.2 实现了领域模型层的懒加载和异步，可以完全克服 Hibernate 等 ORM 框架带来的 Lazy 懒加载错误问题。

## 顺序执行

Jdon 框架的事件机制是并发的多线程，好处是能充分应用 CPU，提高性能，带来的问题是如果业务上希望事件是顺序执行的，如何实现呢？

通过领域模型发出事件指挥技术架构为之服务，这些事件是异步的，如果在第一个事件响应还没有处理完成，第二个事件就要基于第一事件的处理结果进行进一步处理，如何协调他们的前后一致的关系？

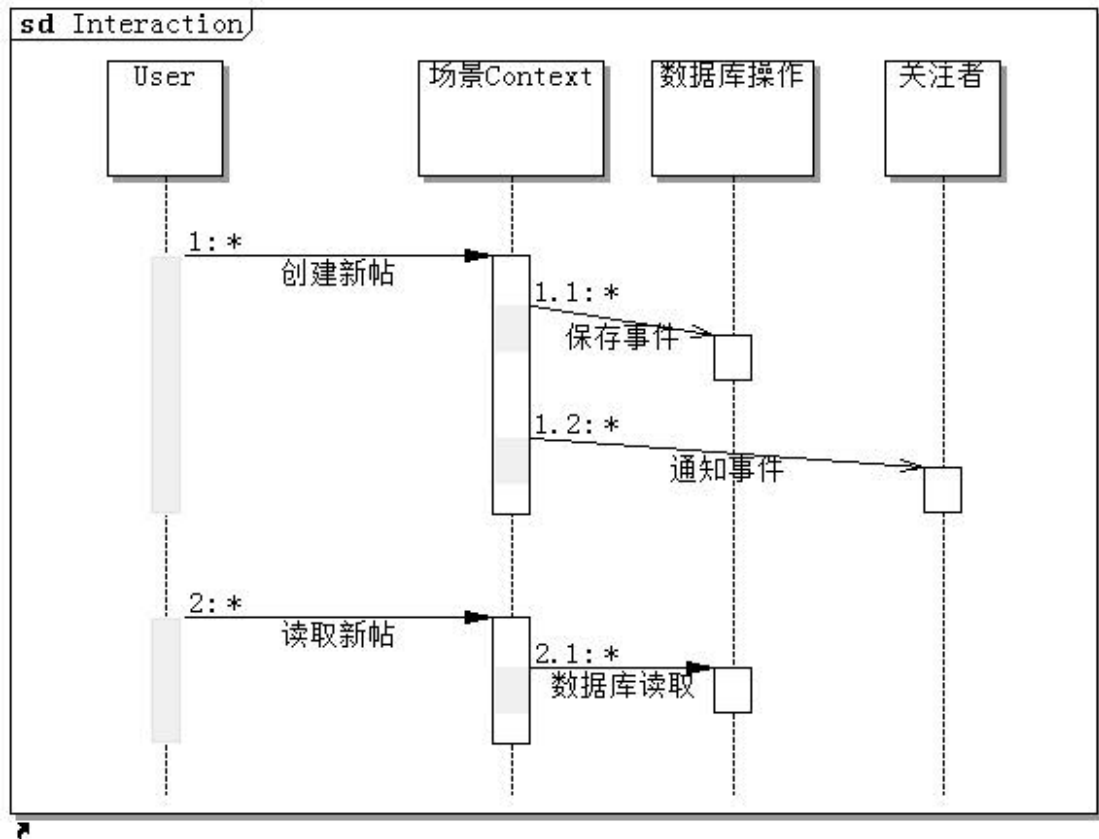
如果说我们普通编程是在一个线程内进行顺序同步编程，那么 Jdon 框架的事件编程实际是一种并行异步编程模型，普通编程需要花力气让程序并行运行；而事件编程则要注意让其顺序执行。

Jdon 框架事件模型提供两种种顺序执行的方式：

1. 根据事件订阅者类的名称顺序执行，比如 ADomainHandler 在 BDomainHandler 之前执行，BDomainHandler 在 CDomainHandler 之前执行。
2. 将上一个事件的处理结果传入下一个事件，然后进行堵塞等待方式。下面主要介绍这种方式。

以 JiveJdon 中帖子创建为案例，当用户提交新的帖子时，发出领域事件进行数据库保存，用户然后就会立即再次从数据库读取这个新帖子，而在另外一个线程中数据库保存还没有完成，那么从数据库是无法读到这个新帖子的。

我们将用户提交新帖子后的处理顺序图如下：



用户有两个动作：创建新帖后再读取新帖，这两个动作次序是有先后次序的，否则读取新帖就无法正确读取。

所以，我们希望这个过程是一个顺序执行的：

创建新帖；然后发出保存事件，数据库保存完成新帖后；再执行通知事件；如果在这一过程中，用户又几乎同时发出读取新帖，那么一定要等待创建新帖的过程全部完成才能执行。

为了实现这种顺序执行过程，我们使用 Jdon 框架的 DomainMessage 的中一种堵塞读取方法，将发出保存事件的 DomainMessage 传递给下一个通知事件，然后在通知事件激活后堵塞读取上一次事件结果 DomainMessage。

```

RepositoryRoleIF repositoryRole = (RepositoryRoleIF) roleAssinger.assign(forumMessage, new
RepositoryRole());
//获得数据库保存的事件结果
DomainMessage domainMessage = repositoryRole.addTopicMessage(forumMessage);
//将事件结果保留，以便读取新帖时使用
transactions.put(forumMessage.getMessageId(), domainMessage);

ThreadRoleIF threadRole = (ThreadRoleIF) roleAssinger.assign(forumMessage, new ThreadRole());
//将数据库保存的事件结果作为通知事件的输入参数
threadRole.aftercreateAction(domainMessage);
  
```

在通知事件的订阅者中代码如下：

```

//获得传入的输入参数，也就是数据库保存事件结果
DomainMessage lastStepMessage = (DomainMessage) event.getDomainMessage().getEventSource();
//对数据库保存事件进行堵塞读取，一直等待数据库保存完成
Object lastStepOk = lastStepMessage.getBlockEventResult();
  
```

```

if (lastStepOk != null) { //如果数据库保存完成
    ForumMessage forumMessage = (ForumMessage) lastStepOk;
    boolean isReplyNotifyForAuthor = forumMessage.isReplyNotify();
    forumMessage = forumFactory.getMessage(forumMessage.getMessageId());
    forumMessage.getForum().addNewThread(forumMessage);

    messageNotifyAction(isReplyNotifyForAuthor, forumMessage);
}

```

在通知事件的实现中，将对上次发出的事件处理结果进行堵塞读取处理结果，如果没有处理结果，就一直等待直至有结果。

那么在上次数据库保存的事件处理中如何表示自己的事件处理完成，有了处理结果呢？

实际上，关键是，在数据库保存事件的订阅者中，只要在处理保存数据库完成后，将一个非空对象赋值给 `DomainMessage` 即可。如下：

```
domainMessage.setEventResult(forumMessage);
```

因为 `forumMessage` 是非空，当调用 `DomainMessage` 的 `setEventResult` 方法时，此方法中有一个新的 `Disruptor` 队列，而通知事件中堵塞读取等待的就是这个新的专门用来传递处理结果的 `Disruptor` 队列，当 `Disruptor` 队列消息时，堵塞读取立即被激活，获得 `forumMessage`，被堵塞的下一个事件就能够知道上一步事件已经处理完成。

在 `JiveJdon` 中，数据库保存事件订阅者是在 `com.jdon.jivejdon.repository.MessageTransactionPersistence` 中使用 `@OnEvent` 方法标注实现的。

```

@OnEvent("addTopicMessage")
public ForumMessage insertTopicMessage(ForumMessage forumMessage) throws Exception {
    logger.debug("enter createTopicMessage");
    try {
        jtaTransactionUtil.beginTransaction();
        messageRepository.createTopicMessage(forumMessage);
        logger.debug("createTopicMessage ok!");
        jtaTransactionUtil.commitTransaction();
    } catch (Exception e) {
        jtaTransactionUtil.rollback();
        String error = e + " createTopicMessage forumMessageId=" + forumMessage.getMessageId();
        logger.error(error);
        throw new Exception(error);
    }
    //返回一个非空对象，表示当前处理过程完成
    return forumMessage;
}

```

由于事件订阅者没有直接使用原始的事件订阅者方式，也就是使用 `@Consumer("addTopicMessage")`，以及实现接口 `com.jdon.domain.message.DomainEventHandler`。

而是使用 `Jdon` 框架更为灵活的 `@OnEvent`，这时只要被标注 `@OnEvent` 的方法必须返回一个非空对象，那么在 `Jdon` 框架中有一个缺省的 `DomainEventHandler` 实现类，就会将 `@OnEvent` 标注的方法返回值自动执行 `DomainMessage` 的 `setEventResult` 方法。

由此可见，只要我们在下一个事件处理之前，对上一步事件处理结果进行堵塞等待，

就能确保事务顺序依次执行。

下面谈谈读取新帖这个事件如何堵塞读取第一个处理事件呢？

我们注意到，在将第一个事件数据库保存事件处理结果传入第二个事件之前，我们已经将第一次事件处理结果保存：

```
transactions.put(forumMessage.getMessageId(), domainMessage);
```

那么，当有读取新帖这中随时可能发生的事件时，我们只要从保存的第一次事件处理结果中也是堵塞读取确认第一个事件处理结束再继续：

```
public boolean isTransactionOk(Long messageId) {
    if (transactions.size() == 0)
        return true;
    if (!transactions.containsKey(messageId)) {
        return true;
    }
    //获得第一次事件
    DomainMessage message = transactions.get(messageId);
    //堵塞读取第一次事件处理结果
    if (message.getBlockEventResult() != null) {
        transactions.remove(messageId);
        return true;
    }
    return false;
}
```

## DCI 实现

DCI：数据 Data，场景 Context，交互 Interactions 是由 MVC 发明者 Trygve Reenskaug 发明的。见 DCI 架构是什么？DCI 让我们的核心模型更加简单，只有数据和基本行为。业务逻辑等交互行为在角色模型中 在运行时的场景，将角色的交互行为注射到数据中。

Jdon 框架提供了两种 DCI 实现风格：没有领域事件的纯粹注入；另外一种领域事件的注入；这两种模型适合不同的场景：

如果我们已经 Hold 住了一个领域对象，那么就直接通过其发出领域事件实现功能；比如模型的修改。否则，我们创建一个上下文 Context，在其中通过 RoleAssigner 将角色接口注入到领域对象中。比如模型新增创建或删除。(对象不能自己创建自己)。

下面谈谈这两种 DCI 模型的实现：

### 角色分配场景

数据模型如下：

```

@Model
public class MyModel {
    private Long id;
    private String name;
    No domain events with @Inject
    public Long getId() {
        return id;
    }
}

```

DCI 的 Role 实现：角色必须有一个接口，否则无法实现 Mixi 注入模型的功能，这是实现 DCI 必要条件。

```

@Introduce("message")
public class RepositoryRole implements RepositoryRoleIF {
    // interface needed

    @Send("save")
    public DomainMessage save(MyModel myModel) {
        return new DomainMessage(myModel);
    }
}

```

Jdon 框架提供了一种角色分配器：com.jdon.domain.dci.RoleAssigner 是一个角色分配器，可以向任何模型中注入任何带有接口 (Mixin)的实现类。

当使用 RoleAssigner, 我们就没有必要从带有元注释 @Introduce(“modelCache”) 和 @Around 的仓储中首先获得一个模型对象，这是模型对象从一个主动核心让位于包含有 RoleAssigner 的 Context 场景上下文，模型成为被注射的被动对象了；而在领域事件驱动中，模型是发出领域事件，是一种主动核心对象。

RoleAssigner 可以手工对任何一个模型对象从外部进行事件注入或角色分配。

下面是使用角色分配器实现的 DCI 场景代码：

```

@Component("repositoryContext")
public class RepositoryContext {
    private final RoleAssigner roleAssigner;

    public RepositoryContext(RoleAssigner roleAssigner) {
        this.roleAssigner = roleAssigner;
    }

    public void interact() {
        MyModel myModel = new MyModel();
        RepositoryRoleIF r =
            (RepositoryRoleIF)roleAssigner.assign(myModel, new RepositoryRole());
        r.save(myModel);
    }
}

```

注意这个 RepositoryContext 必须标注以@Component, 表示其生存在组件边界中，Jdon

框架有两个边界：

@Model 的模型对象是生活在缓存内存中；而 @Component/@Service 生活在应用容器中，比如 Web 容器的 ServletContext 中。

很多情况下，在模型对象还没有生活到内存中时，应用容器总是首先存在，应用容器如同天地，万物之母。

比如模型对象的新增创建时，由于这个模型对象在数据库仓储中还不存在，因此在这种情况下，我们不能 Hold 住还没有创建的模型对象，这时模型对象内部的领域事件就派不上用处。

但是应用容器以及存在，因此，我们可以使用已经生存在应用容器中的 com.jdon.domain.dci.RoleAssigner 对新创建的模型进行功能注入，让这个从页面提交过来的新的模型对象具有能够自己持久化(save())或其他必要的业务功能。

这样，我们通过获取模型时的自动注入和 RoleAssigner 的手工注入两种手段，灵活地根据不同上下文实现风格的 DCI。

源码见：[SimpleJdonFrameworkTest.rar](#)

这种以角色分配器为主要的场景模式的 DCI 比较适合脱离领域模型内部的外部操作，Jdon 框架认为世界基本上有三个部分组成：边界内的事物内部；边界；边界外的事物外部。事物的创建删除一般由事物外部作用，因为一个事物不可能在自己内部创建自己，因为它自己还不存在。

角色分配器的注入 DCI 比较适合模型的创建或删除，在开源 JiveJdon 的 com.jdon.jivejdon.service.imp.message. MessageKernel, ForumMessage 的创建删除是利用角色分配器实现，而 ForumMessage 的修改则是利用模型的领域事件实现。

## 领域事件的 DCI

这种模式的 DCI 实现特点是隐含了场景，凡是领域模型发出事件的地方就是一种 DCI 中的场景上下文。这时领域模型占据了核心位置，场景是被隐含在领域模型内部，这不同于上一种角色分配器中场景和领域模型是分离的。

下图是领域模型在运行时领域事件被 Jdon 框架自动注入的原理图，这个运行时是指，我们从 Repository 仓储(带有元注释 @Introduce(“modelCache”) 和 @Around 的仓储)获得领域模型对象时发生的。

```

@Model
public class MyModel {

    private Long id;
    private String name;

    @Inject
    private MyModelDomainEvent myModelDomainEvent;

    @Inject
    private MyModelService myModelServiceCommand;
}

```

Inject

```

@Introduce("message")
public class MyModelDomainEvent {

    @Send("MyModel.findName")
    public ModelMessage asyncFindName(MyModel myModel) {
        return new ModelMessage(myModel);
    }
}

```

Introduce a interceptor

MyModel <sup>invoke</sup> ---> @Introduce(message) <sup>invoke</sup> ---> MyModelDomainEvent

上图中领域事件 MyDomainEvent 的注入是在如下代码被调用时发生的：

```

@Component("mymrepository")
@Introduce("modelCache")
public class RepositoryImp implements MyModelService {

    @Around
    public MyModel getModel(Long key) {
        MyModel mym = new MyModel();
        mym.setId(key);
        return mym;
    }
}

```

Auto cache MyModel

也就是当如下场景执行时，将会发生事件注入：

```
//当从仓储 RepositoryImp 获得模型对象时，事件被注入到 myModel 对象中
MyModel myModel = repository.getModel(new Long(100));

//通过模型发出事件。
myModel .myModelDomainEvent.asyncFindName(this);
```

上图中,不但可以注射 MyModelDomainEvent 到模型 MyModel 中,偶尔也可以将 Jdon 框架中任何一个以 @component 或 @Service 标注的组件如 MyModelService 注射到 MyModel 中,这种方式相比 DomainEvents 的异步,这是同步方式。

模型注入功能实际上有助于实现 DCI 架构,因为 DCI 架构关键是将角色注入数据模型中。而事件的发布者也就是生产者其实就是一个具有业务行为的角色。

为更清楚说明 DCI, 下面以 JdonFramework 案例说明。

领域模型是 DCI 的 Data, 只有数据和基本行为, 更简单, 但注意不是只有 setter/getter 的贫血模型。如下:

```
@Model
public class UserModel {

    private String userId;
    private String name;
    //注意 这是一个带 Domain Events 注入的
    @Inject
    private ComputerRole computerRole;
```

Domain Events 事件或消息的生产者也就是 DCI 中的角色 Role, 比如我们有一个专门进行计数计算的角色, 实际上真正计算核心因为要使用关系数据库等底层技术架构, 并不真正在此实现, 而是依托消息消费者 @Consumer 实现, 那么消息生产者可以看出是一个接口, 消费者看成是接口的实现:

```
@Introduce("message")
public class ComputerRole {

    @Send("computeCount")
    public DomainMessage computeCount(UserModel user) {
        return new DomainMessage(user);
    }

    @Send("saveUser")
    public DomainMessage save(UserModel user) {
        return new DomainMessage(user);
    }
}
```



DCI 第三个元素是场景 Context，在这个场景下，ComputeRole 将被注入到模型 UserModel 中，实现计数计算的业务功能：

```
public class ComputeContext {

    private DomainMessage ageAsyncResult;

    public void preloadData(UserModel user) {
        if (ageAsyncResult == null)
            ageAsyncResult = user.getUserDomainEvents().computeCount(user);
    }

    public int loadCountNow(UserModel user) {
        preloadData(user);
        return (Integer) ageAsyncResult.getEventResult();
    }

    public int loadCountByAsync(UserModel user) {
        if (ageAsyncResult == null)
            ageAsyncResult = user.getUserDomainEvents().computeCount(user);
        else if (ageAsyncResult != null)
            return (Integer) ageAsyncResult.getEventResult();
        return -1;
    }
}
```

上述是 UserModel 中有一个 @Inject:

```
@Inject
private ComputerRole computerRole;
```

## 模型驱动开发(MDD)案例

DDD 是领域驱动设计(Domain-Driven Design )的简称，DDD 是一种分析设计建模方法，它倡导统一语言，提出了实体和值对象 以及聚合根等概念，借助 DDD 我们能够在结构理清需求中领域模型。[DDD 专题](#)

DCI: Data 数据模型，Context 上下文或场景，Interactions 交互行为是一种新的编程范式，由 MVC 发明人 Trygve Reenskaug 提出。 [DCI 架构是什么？](#)

DCI 的关键是：

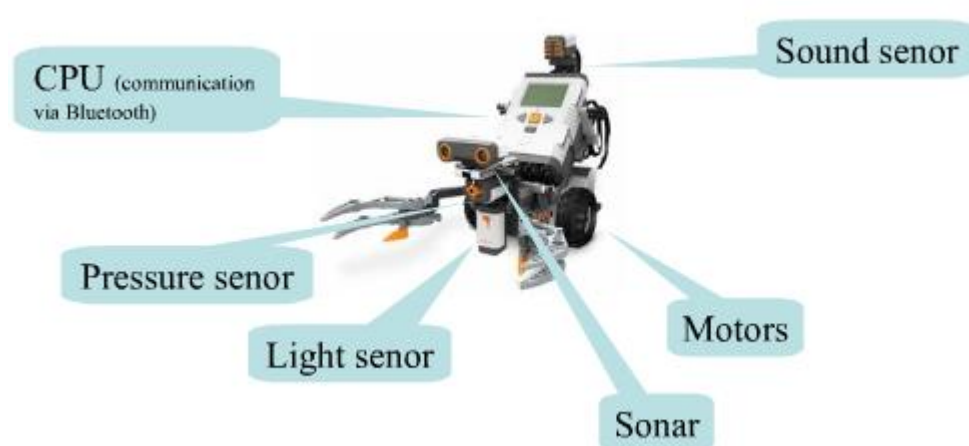
1. 要让核心模型非常瘦。

## 2. 逻辑或行为应该放在角色这个类中

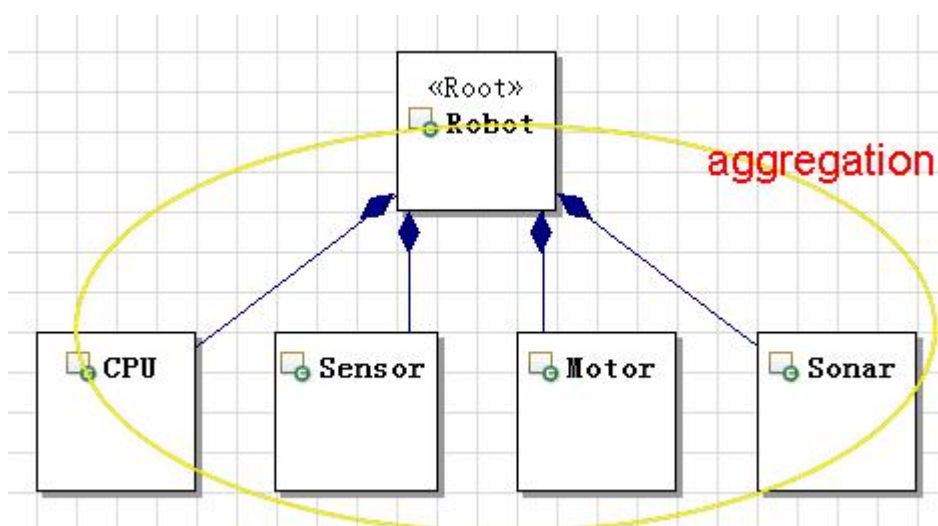
Event Sourcing 是由 Martin Fowler 提出，是将业务领域精髓（尤其是最复杂的）与技术平台的复杂性实现脱钩的天作之合。[为什么要用 Event Sourcing?](#)或 [Domain Events – 救世主](#)

下面我们将演示如何将上述三者在实践中结合在一起？

以机器人 Robot 这个需求为案例，下图是 Robot 描述，我们根据这种图通过 DDD 建模。



从上面这张图中，我们根据 DDD 的实体聚合根等定义，得出如下类图：



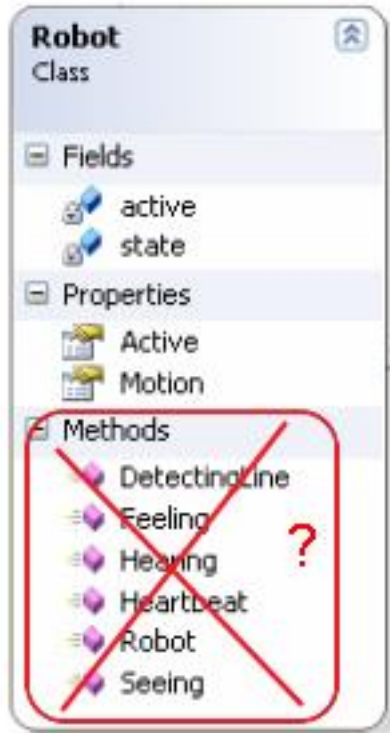
可见，Robot 作为一个聚合集合的根实体，它有四个重要部分组成，这些组成部分与 Robot 形成一种高凝聚的组合关系。缺一不可。

有了这样的结构关系，我们还将细化方法行为，根据[对象的责任与职责](#)，也就是职责驱动开发方法论，它提出一种角色职责的模型：roles-and-responsibilities，见 [Object Design: Roles, Responsibilities, and Collaborations](#)，什么是职责呢？职责就是那些 knowing what; doing what; deciding what.

那么 Robot 如果作为一个智能机器人，它应该有哪些职责呢？

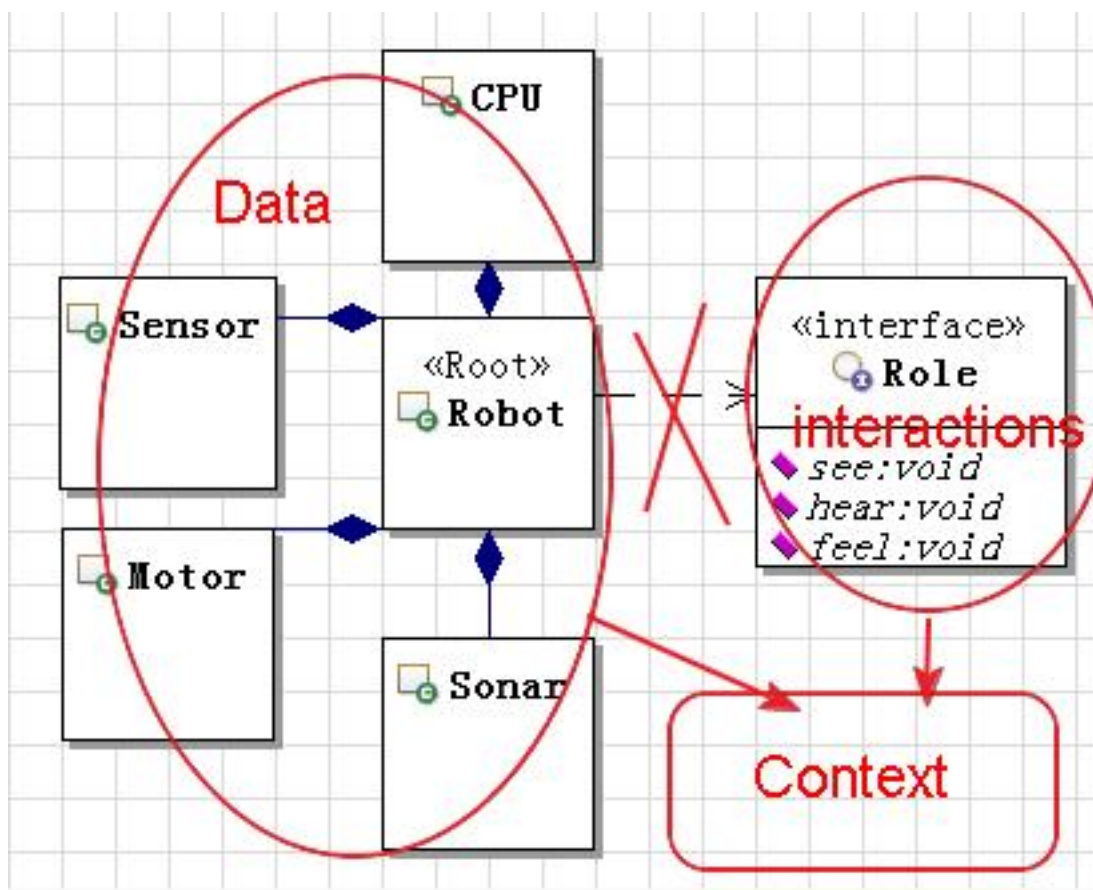
Robot 的职责功能是能听，能看或能感觉，听 看 感觉这些都是其作为一个智能机器人角色的职责。

那么，下一步关键是，如何实现这些职责呢？是否是将这些职责设计作为 Robot 实体类的方法呢？如下：



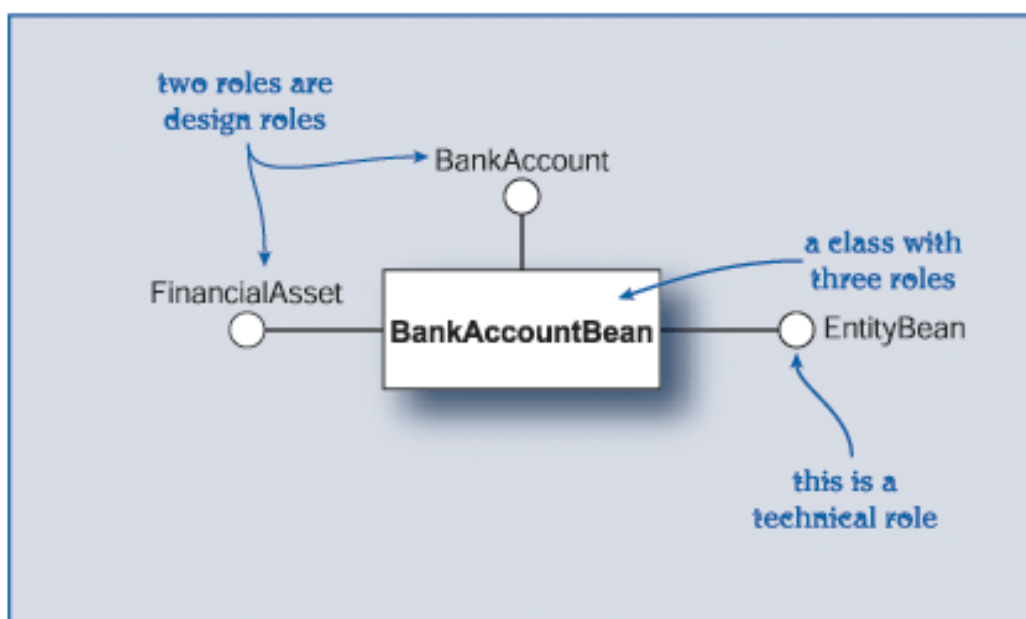
这样设计以后，可能会导致 Robot 实体类非常臃肿，是一个庞大的对象，这有违背 DCI 要旨。

DCI 认为要保持模型的精简，听 看 感觉这些行为是 Robot 作为一个智能机器人，不是普通机器人，这样一个特殊角色具备的职责，应此，应该将这些行为放入一个叫智能机器人的角色中。当在运行时需要的场景 context 时，我们将这个角色中的职责行为注入到精简的数据模型中，如下图：



这样的例子很多，一个人在家是父亲，在单位是经理，父亲和经理都是角色，是不是要将这些角色行为比如签字 烧菜这些和具体业务场景有关的职责放入“人”这个类中呢？显然不是。

又比如银行账户 `Account` 有三种角色：两个是设计角色 `BankAccount` 银行账户 和 `FinancialAsset` 理财账户，另外一个角色是技术角色，它又是 EJB 的实体 `Bean` 专门用来实现持久化保存。



那么如何将上面 DCI 设计或职责驱动落实为代码呢？特别是 Robot 实体类和角色智能机器人的行为如何在运行时场景结合呢？这非常类似桥模式：

```
public String hello(String id) {
    Robot robot = robotRepository.find(id);
    //将角色智能机器人 IntelligentRole 的行为注入到 Robot 数据对象中
    IntelligentRole intelligentRobot = (IntelligentRole) roleAssigner.assign(robot, new IntelligentRobot());
    //得到一个混和 robot 将具有听 看 感觉等能力行为
    return "Hello, " + intelligentRobot.hear();
}
```

类似 Account 的实体持久化角色，，Robot 也有一个保存自己到数据库的技术职责，Robot 保存自己应该是首先由自己发出这样意愿，而不是被保存，是主动保存，其次保存数据库这个动作耗时，影响性能，因此，我们使用领域事件 Domain Events 来间接实现。

一个 PublisherRole 是保存事件发送者 Robot 可以扮演这样一个角色发出保存事件。：

```
@Introduce("message")
public class PublisherRoleImp implements PublisherRole {

    @Send("saveme")
    public DomainMessage remember(Robot robot) {
        return new DomainMessage(robot);
    }
}
```

保存事件的接受方就是 DDD 中定义的仓储 Respository:

```
@Component
@Introduce("modelCache")
public class RobotRepositoryInMEM implements RobotRepository {
    .....

    //保存事件的订阅者 真正实现数据库保存
    @OnEvent("saveme")
    public void save(Robot robot) {
        memDB.put(robot.getId(), robot);
    }
    .....
}
```

.那么 Robot 在什么时候扮演事件发送者发出保存自己的命令呢？可以在任何时候，下面是一个 context：

```
public void save(Robot robot) {  
    PublisherRole publisher = (PublisherRole)  
roleAssigner.assign(robot, new PublisherRoleImp());  
    publisher.remember(robot);  
}
```

至此，我们通过机器人 Robot 案例展示了 DDD DCI 和事件模型等分析设计实现的过程，当然复杂项目将比这个过程更加复杂，需要敏捷迭代，精炼出符合客观规律的核心模型。

以上 Robot 实现源代码下载：[bot.zip](#)

关于 DDD DCI CQRS 等详细 PPT 文档下载。[DDD DCI CQRS.PPT](#)

## 不变性设计原则

不变性是统领业务分析和高性能架构重要法门，通过业务上不变性分析设计，可以实现代码运行的并发高性能和高扩展性。

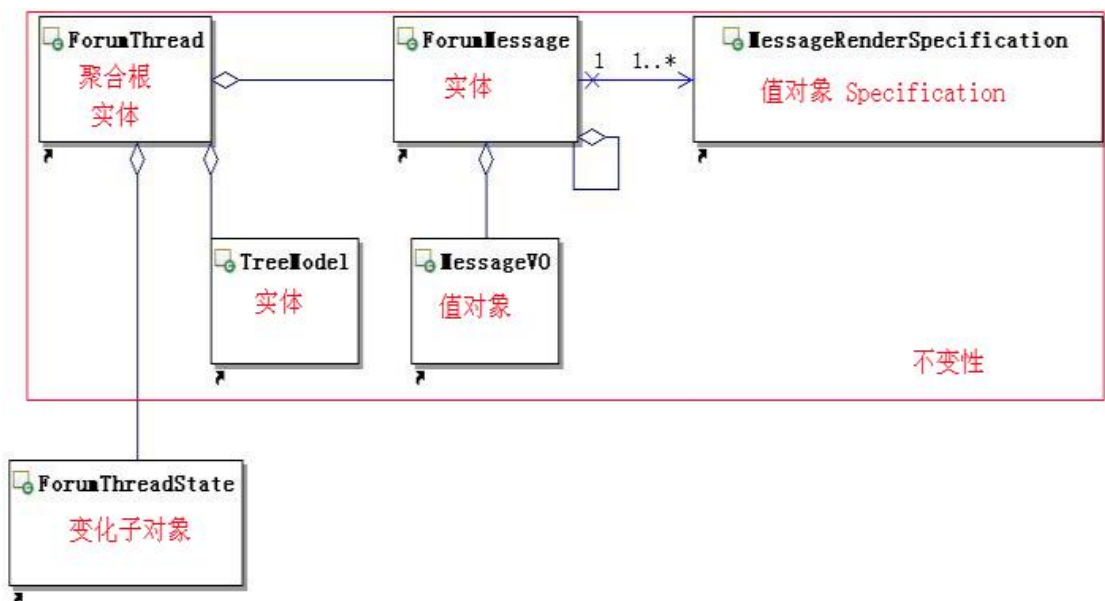
不可变性是一种抽象，它并不在自然界中存在，世界是可变的，持续不断变化。所以数据结构是可变的，他们是描述真实世界某段时间内的状态，这个看上去类似 OOP 设计，核心概念是：在内存中对象 `object in RAM` 并不一定等于真实的对象，数据会有不准确，带来延迟。

在大部分琐碎的需求中将一切看成是可变的也许没有问题，但是当下面情况就不同了：

- 1.数据结果被并发线程同时修改
- 2.多个用户对一个共享对象进行冲突性修改。
- 3.用户提供了无效数据，应该被回滚。

在自然可变的模型下，你会在这些情况下发生不一致性或破碎 `crushing`，显然可变性很容易导致错误。需要的是一种新视野，新角度，一种事务 `transaction` 视野，当你从事务性程序中看世界时，一切都是不可变的。相关文章：[删因与引用透明](#)。

实战 DDD(Domain-Driven Design 领域驱动设计 :Evans DDD) :  
<http://www.jdon.com/mda/ddd.html> 中提出状态是一种值对象，而值对象的特征是不可变性，如果发生变化，就全部替换。



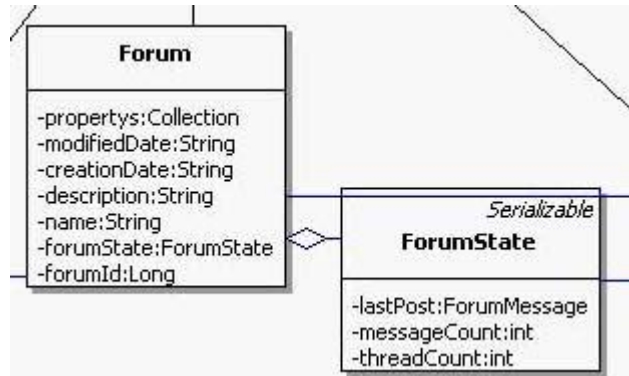
值对象状态的这种不可变性还为我们进行海量数据处理提供了一种优雅的架构：如何打败 CAP 定理？<http://www.jdon.com/jivejdon/thread/42886#23136931>

数据有两个重要属性：首先数据是基于时间的，数据是表达一段时间内一个逻辑为真的事实。另外一个属性是数据本质上是不可变的，因为和时间有关，我们是不能回到过去改变数据的真实性。

我们把基于时间的数据准确称为状态，状态是表达一段时间内一个逻辑为真的事实，状态是不可变的，因为我们不能回到过去改变状态。

针对状态的不可变性，在应用中需要针对状态改变进行一种最佳实践，也就是说：如果状态发生改变，我们只要重新构建一个新的状态对象，而不是在原有状态对象内部去修改字段。

根据这个宗旨，对 Jivejdon 4 版本中没有对状态做到不变性替换的问题进行了重构。以 Forum 的 ForumState 为案例，FormThread 的 ForumThreadState 也是同理，如下：



下面是根据不变性新设计的 ForumState:

```
public class ForumState {
    private final AtomicLong threadCount;

    /**
     * the number of messages in the thread. This includes the root message. So,
     * to find the number of replies to the root message, subtract one from the
     * answer of this method.
     */
    private final AtomicLong messageCount;

    private final ForumMessage lastPost;

    private final Forum forum;

    private final SubscribedState subscribedState;

    public ForumState(Forum forum, ForumMessage lastPost, long messageCount, long threadCount) {
        super();
        this.forum = forum;
        this.lastPost = lastPost;
        this.messageCount = new AtomicLong(messageCount);
        this.threadCount = new AtomicLong(threadCount);
        this.subscribedState = new SubscribedState(new ForumSubscribed(forum));
    }
}
```

与以前的 ForumState 区别主要是：将其中所有字段都加了 final，表示不能修改，构建 ForumState 时就必须指定其中的数据值。

专门创建一个 FormStateFactory 负责 FormState 的生命周期维护：



```

public interface ForumStateFactory {

    void init(Forum forum, ForumMessage lastPost);

    void addNewMessage(Forum forum, ForumMessage newLastPost);

    void addNewThread(Forum forum, ForumMessage newLastPost);

    void updateMessage(Forum forum, ForumMessage forumMessage);

}

```

过去 ForumState 是直接进行修改其中的字段，如下：

```

public synchronized void addNewMessage(Forum forum) {
    forumState.addMessageCount();
    forumState.setLastPost(forumMessageReply);
    forumMessageReply.setForum(this);
    this.publisherRole.subscriptionNotify(
}

```

```

local: addNewMessage(ForumMessageReply)
public void addNewMessage(ForumMessageReply forumMessageReply) {
    forumStateManager.addNewMessage(this, forumMessageReply);
    this.publisherRole.subscriptionNotify(
}

```

forumState.setLasPost 这些方法是直接修改，现在重构成使用 ForumStateFactory 的 addNewMessage 来实现，其内部方法具体代码如下：

```

public void addNewMessage(Forum forum, ForumMessage newLastPost) {
    try {
        long newMessageCount = forum.getForumState().addMessageCount();
        //重新创建新的 ForumState
        ForumState forumState = new ForumState(forum, newLastPost, newMessageCount,
forum.getForumState().getThreadCount());
        //替换 Forum 原来旧的 forumState
        forum.setForumState(forumState);
        newLastPost.setForum(forum);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

通过这种不变性的重构，好处是去除了原来 addNewMessage 方法的 synchronized 同

步锁，通过不变性规避了共享锁的争夺，从而获得了更好的并发性能。

需要注意的是：不同的事件引起不同的状态，如读取事件引发读状态的变化，比如帖子的浏览数；而写事件引发写状态的变化；越是频繁发生的事件，其不变性的时间周期 **Scope** 就越短，实现的手段就有区别。

帖子浏览数是帖子的一种读取次数状态，频繁发生，如果每次读取事件发生，我们象上面采取值对象不变性原则，每次都构造一个新的值对象，无疑会有大量垃圾临时对象产生，从而频繁触发 **JVM** 的垃圾回收机制，那么在这种情况下，我们可以采取其他并发措施，比如使用 **JDK** 的原子类型 **AtomicLong** 等，通过其提供的自增获取的原子功能实现并发。

还有一种状态是结构关系状态，比如帖子之间的相互回复关系可以组成一个二叉树的模型，如果有新帖写事件发生，会增加一个新帖 **ID** 在这个树结构中。这是一种难以避免的在原来数据上进行修改情况。

## Jdon 框架高级使用

从前面章节已经清楚：Jdon 框架快速开发主要体现在 CRUD 和批量查询这两个基本功能上，大量数据库系统基本都由这两个基本功能组成。

Jdon 框架的这两个功能横跨 J2EE 多个层次，主要简化工作体现在表现层，这一层主要是简化了 Struts 一些烦琐的配置和代码过程。

在这一章，主要介绍在深入使用 Jdon 框架中碰到的一些问题和解决方案，从而达到更加灵活地使用 Jdon 框架。

### 内嵌对象(Embedded Object)缓存设计

请看下面这段代码：

```
public class Category extends Model{
    String Id;
    Product product;    //内嵌包含了一个 Product 对象
}
```

Category 这个 Model 内嵌了 Product 这个 Model，属于一种关联关系。

目前 Jdon 框架提供的缺省缓存是扁平式，不是嵌入式或树形的（当然也可以使用 JbossCache 等树形缓存替代），因此，一个 ModelB 对象（如 Product）被嵌入到另外一个 ModelA 对象（如 Category）中，那么这个 ModelB 对象随着 ModelA 对象被 Jdon 框架一起缓存。

假设实现 ModelA 已经在缓存中，如果客户端从缓存直接获取 ModelB，缓存由于不知道 ModelB 被缓存到 ModelA 中（EJB3 实体 Bean 中是通过 Annotation 标注字段），那么缓存可能告知客户端没有 ModelB 缓存。那么有可能客户端会再向缓存中加一个新的 ModelB，这样同一个 ID 可能有两份 ModelB，当客户端直接调用的 ModelB 进行其中字段更新，那么包含在 ModelA 中的 ModelB 可能未被更新（因为 ModelA 没有字段更新）。

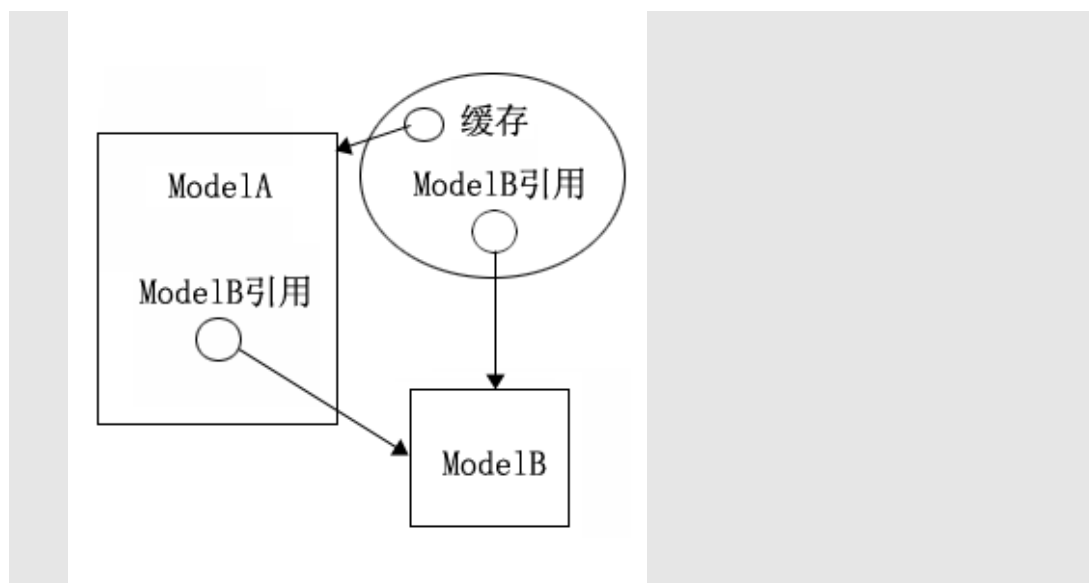
有两种解决方案：

第一．最直接方式，通过手工清除 ModelA 缓存方式来更新，或者耐心等待 ModelA 缓存自动更新。手工清除缓存见下章。

注意：下面这种做法将也会导致不一致现象发生：

在 DAO 层读取数据库。生成 ModelA 时，直接读取数据库将 ModelB 充填。

第二.在进行 ModelA 和 ModelB 的相关操作服务设计时，就要注意保证这两种情况下 ModelB 指向的都是同一个。如下图：



为达到这个目的，只要在 Service 层和 Dao 层之间加一个缓存 Decorator，服务层向 Dao 层调用的任何 Model 对象都首先经过缓存检查，缓存中保存的 ModelA 中的 ModelB 是一个只有 ModelB 主键的空对象，在服务层 getModelA 方法中，再对 ModelA 的 ModelB 进行充实，填满，根据 ModelA 中的 ModelB 的主键，首先再到缓存查询，如果有，则将缓存中 ModelB 充填到 ModelA 的 ModelB 中，这样上图目的就可以实现了。

相关实现可参考 JiveJdon 的代码，Forum/ForumMessage 都属于这种情况，com.jdon.jivejdon.repository.\*

## Model 缓存使用

Jdon 框架通过两种方式使用 Model 缓存：

CRUD 框架内部使用，如果你使用 Jdon 框架提供的 CRUD 功能，那么其已经内置 Model 缓存，而且会即时清除缓存。

通过 CacheInterceptor 缓存拦截器，如果你不使用 Jdon 框架的 CRUD 功能，缓存拦截器功能将激活，在向 Service 获取 Model 之前，首先查询当前缓存器中是否存在该 Model，如果有从缓存中获取。当你的 Model 中数值更改后，必须注意自己需要手工清除该 Model 缓存，清除方法如下介绍。

Jdon 框架除了提供单个 Model 缓存外，还在持久层 Dao 层提供了查询条件的缓存，例如如果你是根据某个字段按照如何排列等条件进行查询，这个查询条件将被缓存，这样，下次如果有相同查询条件，该查询条件将被提出，与其相关的满足查询条件一些结果（如符合条件总数等）也将被从缓存中提出，节省翻阅数据库的性能开销。

### 手工访问缓存

在一般情况下，前台表现层通过 getService 方法访问服务层一个服务，然后通过该服

务 Service 获得一个 Model，这种情况 Jdon 框架的缓存拦截器将自动首先从缓存读取。

但是，有时我们在服务层编码时，需要获得一个 Model，在这种情况下，Jdon 框架的缓存拦截器就不起作用，这时可能我们需要手工访问缓存。

因为所有服务类 POJO 都属于 Jdon 框架的容器内部组件，这实际是在容器内访问容器组件的问题。

使用 `com.jdon.container.finder.ContainerCallback`，同时，该服务 POJO 类以 `ContainerCallback` 作为构造参数，当该 POJO 服务类注册到容器中时，容器的 Ioc 特性将会找到事先以及注册的 `ContainerCallback` 类。

通过 `ContainerCallback` 获得 `ContainerWrapper` 容器实例，然后通过 `ContainerWrapper` 下面方法：

```
public Object lookup(String name);
```

从容器中获得 `container.xml` 中注册的组件实例，这种方法每次调用获得的是同一个实例，相当于单例方式获得。

```
ModelManager modelManager =
```

```
(ModelManager)containerWrapper.lookup("modelManager");
```

其中“modelManager”字符串名称是从 Jdon 框架的 `jdonFramework.jar` 包中 `META-INF` 的 `container.xml` 中查询获知的。

获得 `ModelManager` 后，我们基本可以访问 Model 有关的功能安排，如 `ModelManager` 的 `getCache` 方法。

下节的手工清除缓存中，我们是通过 `WebAppUtil.getComponentInstance` 获得 `ModelManager` 实例，这是一种从容器外获得容器组件的方式，本节介绍从容器内获得容器组件的方式，这两种方式可根据我们实际需要灵活使用，关键是弄清你需要在哪里触发组件调用？

## 手工清除缓存

注意：手工清除缓存不是必要的，因为缓存中对象是有存在周期的，这在 `Cache.xml` 中设置的，过一段时间缓存将自动清除那些超过配置时间不用的对象，这样你修改的数据将被从数据库重新加载。如果你等不及这些内在变化，可以手工处理：

有两种情况需要手工清除缓存，首先，在持久层的 Dao 类中，总是需要手工清除查询条件的缓存（注意不是 Model 缓存，是查询条件的缓存），只要在相应的增删改方法中调用 `PageIteratorSolver` 的 `clearCache` 方法既可。

如果你不实行这种缓存清除，那么你更改一个 Model 数据或新增一个新的 Model 数据，你在批量查询时，将看不到任何变化：Model 数据没有被修改；新的 Model 没有出现在查询页面中。

其次，单个 Model 缓存在不使用 Jdon 框架的 CRUD 功能下也必须手工清除，如果你使用 1.2.3 以后版本，可以调用 `com.jdon.strutsutil.util.ModelUtil` 类的 `clearModelCache` 方法，该方法一般是 Action 中调用后台增删改服务之前被激活调用：

注意，手工清除 Model 缓存代码关键是：

```
modelManager.removeCache(keyValue);
```

`keyValue` 是 Model 的主键值，例如 User 的主键 `userId` 值是“2356”，那么 `keyValue` 就是“2356”。简化代码如下：

```
//获得 ModelManager 实例
```

```
ModelManager modelManager = (ModelManager)
WebAppUtil.getComponentInstance(ComponentKeys.MODEL_MANAGER, request);
modelManager.removeCache(keyValue);
```

上面代码是在容器外访问获得 **ModelManager**，使用上节容器内访问组件方式也可以获得 **ModelManager**；前者适合在表现层使用；后者适合在服务层使用。

最后，介绍一下清除全部缓存的方式，调用 **ModelManager** 的 **clearCache** 方法（Jdon 框架 1.3 以上版本），这样实际上将整个 Jdon 框架缓存全部清零。

前面两种清除缓存方式前提是首先获得 **ModelManager**，特别是服务层需要清除缓存时，需要以容器内访问组件方式获得 **ModelManager**，这只适合 POJOService 构成的服务层，如果我们使用 EJB 的 Session Bean 作为服务层实现，这时当前版本的 Jdon 框架容器不会在 EJB 容器中加载，因此，在 Session Bean 中无法访问到容器，无法获得 **ModelManager** 了，在这种情况下，可以通过设置 Model 的 **setModified** 属性为 **True**，表示该 Model 已经修改更新，这样当表现层获取该 Model 时，Jdon 框架缓存拦截器拦截时，发现该 Model 已经被修改，也就不会从缓存中获取。

Model 还有另外一个方法 **setCacheble**，当设置为 **false** 时，该 Model 将不会被保存到缓存中。如果你不希望某个 Model 被框架自动存入缓存，那么使用此功能，**setCacheble** 和 **setModified** 区别是，前者一旦设置为真，相当于缓存失效，以后再也不能用缓存；而后者则是当前 Model 表示被修改过，这样当有任何再次（限一次）试图从缓存中读取这个 Model 时，都会被阻挡，从而可直接从数据库获得，然后再保存到缓存中，这样缓存中的 Model 数据就是新鲜的了。

明白 Jdon 框架 **setModified** 这个神奇作用，当你设置一个 Model 的 **setModified** 为真，那么你再读取缓存时，Jdon 框架内部将忽略你这个读取，返回一个 **null**；这样你就根据返回是否为空，再从数据库直接获得，获得后，别忘记再保存到缓存中，已覆盖前次修改的旧数据，保证以后每次从缓存中读取的都是新鲜数据。

＝总结如下：手工缓存清除共有两种方式：

直接操作缓存，从缓存中清除；

如果无法操作到缓存体系，那么设置 Model 的 **setModified**。

## 配置 encache 作为缓存

从 JdonFramework5.1 以后，在该项目目录下有一个 **componenets/encache** 项目，缺省 Jdon 框架的缓存是使用一个简单的 **com.jdon.util.UtilCache**，这种缓存是可以更换的，如果想更换为 **encache**，步骤简单，如下：

1. 更改 JdonFramework.jar 包中 META-INF 的 **container.xml**（方法可通过 winrar 打开 JdonFramework.jar，将 **container.xml** 解压更改后，再拖放回去覆盖原来的）：

```
<!-- 将原来缺省这行注释掉
comment/delete this line in jdonframework.jar /META-INF/container.xml
<component name="cache" class="com.jdon.util.LRUCache" >
    <constructor value="cache.xml"/>
</component>
-->
```

```

        <!--加入下面行
        active EnCache see project : components/encache: add these lines in jdonframework.jar
/META-INF/container.xml -->
        <component name="cache" class="com.jdon.components.encache.EncacheProvider" />

        <component name="ehcacheConf" class="com.jdon.components.encache.EhcacheConf" >
            <constructor value="ehcache.xml"/>
            <constructor value="sampleCache1"/>
        </component>

```

2. 将项目目录 components/encache/dist 下的 jar 包文件 jdon-encache.jar ehcache-1.2.4.jar commons-logging.jar 和 JdonFramework.jar 放在一起。
3. 重新启动 JBoss 或 Tomcat

## 配置 Hazelcast 作为缓存

1. 下载 hazelcast-1.9.2.jar 和 hazelcast-hibernate-1.9.2.jar。
2. 增加新类 HazelcastProvider。
3. 更改 CacheableWrapper，如下：  
public class CacheableWrapper implements java.io.Serializable
4. 在 container.xml 中增加：  
<component name="cache" class="com.jdon.components.hazelcast.HazelcastProvider" />
5. 增加文件 hazelcast.xml
6. 另外 web 工程中的 hibernate.cfg.xml 需要更改如下（基于 hibernate3.2）：  
<property  
name="hibernate.cache.provider\_class">com.hazelcast.hibernate.provider.HazelcastCacheProvider</property>  
<property name="hibernate.cache.useminimalputs">true</property>

更多问题讨论：

<http://www.jdon.com/jivejdon/thread/39870>

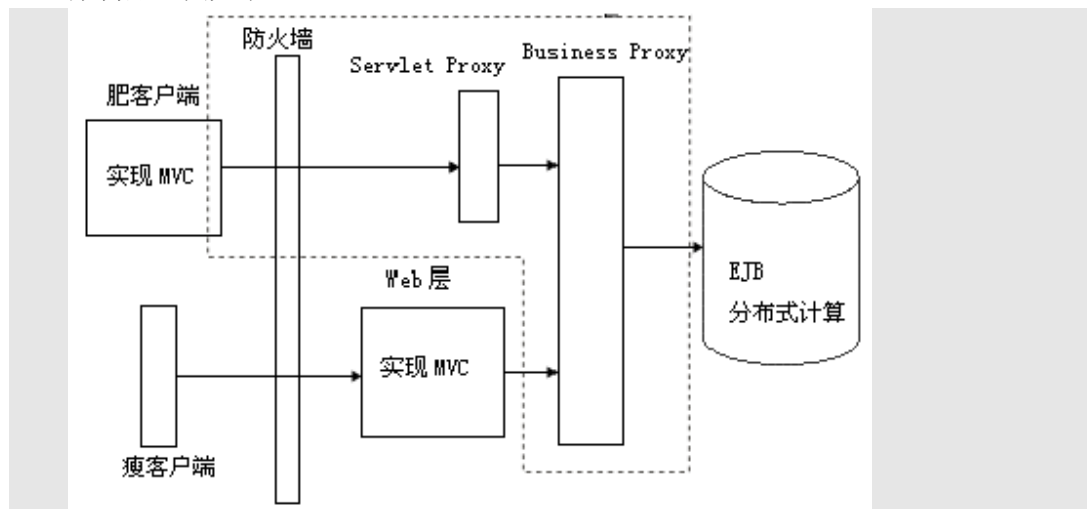
## C/S 架构下胖客户端远程调用

Jdon 框架不但是一个 B/S 架构框架，还支持 C/S 架构，客户端可以是 Java 类型的任何客户端，如 Swing、Applet 和 SWT 等。

使用 Jdon 框架，可以在无需改变业务服务层代码的情况下，将这些服务 Service 提供给远程 Java 客户端调用，这是一种基于 Http 协议的 RPC，它区别于 Web 服务的最大不同是：Jdon 框架将客户端请求打包对象传送；而不是转换成 XML；省却了两端转换解析步骤，提高了性能。

有两种使用方式，一种见下面，还有一种更方便方式，使用 Hessian，可见 [Jdon 框架 5.8 版本增加功能](#)。

架构原理图如下：



## 客户端配置

客户端调用远程服务器的服务核心代码很简单，如下：

```
MySessionLocal mySessionLocal = (MySessionLocal)serviceFactory.getService(  
    FrameworkServices.MySessionEJB);  
mySessionLocal.insert();
```

客户端调用服务就如同与服务在同一个 JVM 中一样，但是因为客户端在远程，所以需要进行对远程服务器的一些配置，比如规定远程服务器的 IP 地址和 Servlet Prox 名称。

具体客户端代码可见框架案例 Samples 中的 remoteClient 代码。

## 服务器端配置

首先激活框架系统的远程访问，在 web.xml 加入：

```
<servlet>  
    <servlet-name>EJBInvokerServlet</servlet-name>  
    <display-name>接受远程客户端访问</display-name>  
    <servlet-class>com.jdon.bussinessproxy.remote.http.InvokerServlet</servlet-class>  
    <init-param>  
        <param-name>configList</param-name>  
        <param-value>jdonframework.xml</param-value>  
    </init-param>  
</servlet>
```



```
<servlet-mapping>
  <servlet-name>EJBInvokerServlet</servlet-name>
  <url-pattern>/auth/EJBInvokerServlet</url-pattern>
</servlet-mapping>
```

然后激活容器基于 HTTP 的基本验证，配置 web.xml：

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>SecurityRealm</realm-name>
</login-config>
```

在 web.xml 中定义访问权限如下：

```
<security-constraint>
  <display-name>User protected</display-name>
  <web-resource-collection>
    <web-resource-name>Collection1</web-resource-name>
    <url-pattern>/auth/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>User</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>NONE</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

这样，访问/auth /EJBInvokerServlet 将得到授权保护。下面在 web.xml 中分别配置该系统的角色和 JNDI 配置。

```
<security-role>
  <role-name>Admin</role-name>
</security-role>
<security-role>
  <role-name>User</role-name>
</security-role>
<ejb-local-ref>
  <ejb-ref-name>ejb/EJBControllerLocal</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>com.jdon.security.auth.ejb.SecurityFacadeLocalHome</local-home>
  <local>com.jdon.security.auth.ejb.SecurityFacadeLocal</local>
  <ejb-link>SecurityFacade</ejb-link>
</ejb-local-ref>
```

在 jboss-web.xml 配置中使用容器安全机制：

```
<jboss-web>
  <security-domain>java:/jaas/SecurityRealm</security-domain>
</jboss-web>
```

然后在 JBoss 的 login-config.xml 中配置 SecurityRealm

## Struts+Jdon 表现层编程

通常，一个 Jsp 页面可能显示很多各种内容，涉及到肯定不只一个数据库，也就是多个组件的结合，但是 Struts 配置文件中一般又是一个 Action 一个 ActionForm 一个 Jsp，他们之间的关系是 1:1:1。

如何在这样一个体系下实现灵活的一个页面多个组件编程模式呢？

首先，要区分两种情况：一个 Action 一个 ActionForm，一个 ActionForm 中装载多个组件；一个 Action 一个 ActionForm 一个组件；

### 一个 ActionForm 多个组件

第一种情况：如果同一个 Action 就可以对付这些组件，那么在这种情况下又有两个办法：

1.将这多个组件装入一个 ActionForm 中，如使用 MapForm 等机制；

一般多个组件在同一个 Jsp 页面显示情况比较经常发生在批量查询中，对于批量查询，因为 Jdon 框架规定必须有一个 ModelListForm，在 ModelListForm 中最多只能装载一个 Model，如果你有多个组件，也就是多个 Model，如何装入 ModelListForm 呢？

可以使用 Jdon 框架的 `com.jdon.controller.model.DynamicModel` 装载多个 Model，其实 DynamicModel 中有一个 Map，然后将 DynamicModel 装入 ModelListForm 的 oneModel 中。

Struts 对于 Map 可以直接访问，以 DynamicModel 为例子：

Action 中代码：

```
DynamicModel dynamicModel = new DynamicModel();
Test test = new Test();
test.setName("oooooooooooooooooooooo");
//装入 Test 组件，可以装入更多其他组件。”test”为 Test 实例的 key 值
dynamicModel.put("test", test);
modelListForm.setOneModel(dynamicModel);
```

Jsp 中显示代码：

```
<bean:define id="dynamicModel" name=" modelListForm " property="oneModel"/>
<bean:define id="test" name="dynamicModel" property="value(test)"/>
<bean:write name="test" property="name"/>
```

其中 `value(test)`是这样解释：`value` 是 `dynamicModel` 方法 `getValue` 方法的简写，`test` 是之间保存 Test 实例的 key 值，然后从 test 对象中取出其属性 `name` 的值。

2.手工将多个组件装入 request/session 等 scope 中，然后根据其名称在 jsp 中获得。这种方式比较传统和原始，类似 Jsp 编程方式，学习难度小，方便。

### 一个 ActionForm 一个组件

前面情况，多个组件装载在一个 ActionForm 中，隐含的意思实际是：所有功能都可以在一个 Action 中实现，在这个 Action 中，我们将多个组件装入当前对应的 ActionForm，

但是，如果这些组件必须有预先由不同的 Action 来处理，每个组件必须经过 Action -->ActionForm 流程，那么在这种情况下也有两种办法：

1. 使用 Tiles，不同流程输出到同一个页面的不同区域。是一种并行处理方式。
2. 对多个流程首尾相连，第一 Action forward 结果是第二个 Action，最后输出一个 Jsp，在这个 jsp 中就可以使用前面多个流程的多个 ActionForm 了，这属于串行方式。

例如，当前页面是使用 Jdon 框架的批量查询实现，Jdon 框架的批量查询一定是一个 ModelListAction 对应一个 ModelListForm，这是一个比较固定的流程了。如果在当前这个批量查询页面中还要显示其他 Model 的批量查询，也就是两个或多个 Model 的批量查询在同一个 Jsp 页面中显示，在这种情况下有上述两个方案选择。

第一个方案前提是你必须使用 Struts+Tiles 架构，使用 Tiles 可以将页面划分成任意块，这样多个批量查询在页面任何位置以任何方式组合显示，完全灵活，缺点是多加入 Tiles 概念。

在一般小型应用中，我们只需采取第二方案，将多个批量查询的 Action 的 struts-config.xml 配置进行首尾连接既可，这种方式类似是按照 Jsp 页面先上后下的顺序串联的。

例如：threadPrevNexListForm 和 messageListForm 是两个批量查询的 ModelListForm，那么 struts-config.xml 配置如下：

```
<action path="/message/messageList"
type="com.jdon.jivejdon.presentation.action.ThreadPrevNexListAction"
name="threadPrevNexListForm" scope="request"
validate="false">
  <forward name="success" path="/message/messageListBody.shtml"/>
</action>

<action path="/message/messageListBody"
type="com.jdon.jivejdon.presentation.action.MessageListAction"
name="messageListForm" scope="request"
validate="false">
  <forward name="success" path="/message/messageList.jsp"/>
</action>
```

这样，当 url 调用/message/messageList.shtml 时，最后导出/message/messageList.jsp 这个页面，在 messageList.jsp 中可以访问 threadPrevNexListForm 和 messageListForm 两个 ActionForm 了。

上述组件构造方法可应用在复杂的 Master-Details 编程中，将 Master 看成一个对象 Model，Details 是一系列子 Model 的集合。

## 常用 API 说明

Jdon 框架提供了丰富的案例源码，一般简单情况下可以通过参考案例源码的用法明白一些用法，但是在深入使用过程中，需要熟悉一些 API 的用法，上节缓存的使用实际已经涉及了 Jdon 框架的 API 使用。

## JdbcTemp API

`com.jdon.model.query.JdbcTemp` 提供了一个简单的持久层解决方案，通过 `JdbcTemp`(JDBC 模板)可以完成增/删/改/查等 SQL 语句操作。

一般我们会选择 `Hibernate/Ibatis` 等作为持久层架构，但是这些框架需要另外学习和复杂的配置，对非常轻量的简单应用，直接使用 `JdbcTemp` 可节省大量时间，随着项目复杂和成熟，还是可以使用这些专门框架来替代 `JdbTemp` 的。

`JdbcTemp` 主要帮助开发者省却复杂的数据库 JDBC 操作语句（如打开/关闭数据库连接等），只要告诉它你要操作的 SQL 语句和参数数值，通过如下一句：

```
jdbcTemp.operate(queryParams, sql);
```

其中 `queryParams` 参数是一个 `List` 集合类型，里面装载的是参数集合，是有一定顺序的；`sql` 是一个字符串类型的 SQL 语句，如：

```
String sql = "INSERT INTO testuser (userId , name)  VALUES (?, ?)";
List queryParams = new ArrayList();
queryParams.add(userTest.getUserId());
queryParams.add(userTest.getName());
jdbcTemp.operate(queryParams,sql);
```

`queryParams` 中装载的是 `sql` 语句中两个问号参数，顺序也是和问号的顺序对应的，第一个问号是对应 `testuser` 的 `userId` 字段；第二个问号对应 `testuser` 的 `name` 字段。

数据库的 `INSERT UPDATE DELETE` 操作都是可以通过 `operate` 方法实现。

目前 `JdbcTemp` 的 `operate` 方法只支持有限几种类型：`String Integer Float Long Double Bye Short`，也就是说，当你需要更新数据库的参数类型符合这几种，可以使用 `operate` 方法，简化 JDBC 语句编写。

但是，如果这几种类型不符合你的参数类型，只能直接使用 JDBC 或使用专门的持久层框架等，`Jdon` 框架提供这个简单 `operate` 方法是为了简化一般系统的开发工作。

`JdbcTemp` 也提供多种查询操作语句：`queryMultiObject` 或 `querySingleObject` 方法，例如：

```
jdbcTemp.queryMultiObject (queryParams, sql);
```

```
jdbcTemp.querySingleObject (queryParams, sql);
```

`querySingleObject` 是查询返回一个对象，适合如下 SQL 语句：

```
select name from testuser where userId = ?
```

```
select count(1) from testuser where userId = ?
```

返回的是数据表一个字段，`querySingleObject` 则返回的相应类型，如字符串等，整数型则是整数型对象，如 `int` 则返回的是 `Integer`，`bigInt` 则返回的是 `Long`。

`queryMultiObject` 是最常用的查询方法，与 `querySingleObject` 返回单个字段相反，它返回的是多个字段，当然也可能是多行多个字段。类如 SQL 语句：

```
select userId, name from testuser where userId = ?
```

这个返回的两个字段，而且结果一般是一行，就是一条记录，但是下面 SQL 语句：

```
select userId, name from testuser where name = ?
```

可能返回多行符合查询条件的记录，每行记录包含两个字段。

这两种形式都是可以采取 `queryMultiObject` 调用，`queryMultiObject` 返回的是一个 `List` 结果，在这个 `List` 结果中按照查询顺序装载着查询结果，这个 `List` 中每个元素是一个 `Map` 对象，`Map` 的 `key` 是数据表字段名称，`Map` 的 `value` 是数据表字段名对应的查询出来的值。

因此，从 `queryMultiObject` 结果中取值的写法如下：

```

List list = pageIteratorSolverOfUser.queryMultiObject(queryParams, GET_FIELD);
Iterator iter = list.iterator();
if (iter.hasNext()) { //如果有多行记录，这里是 while
    Map map = (Map) iter.next();
    ret = new UserTest();
    ret.setName((String) map.get("name"));
    ret.setUserId((String) map.get("userId"));
}

```

List 结果中装载内容结构如下：

字段名	字段值
字段名	字段值

## JdbcTemp 使用前提

只要将 DataSource 传给 JdbcTemp 即可，DataSource 可以来自容器的 JNDI；也可以来自自己编码的连接（自己编码的连接代码需要向外界提供获得 DataSource 方法）；也可来自 Hibernate 等持久层框架，在这些框架配置中配置数据库连接，然后通过这些框架的 API 获得 DataSource 赋值给 JdbcTemp 即可。

目前 JdbcTemp 的 operate 方法只支持有限几种类型：String Integer Float Long Double Byte Short，也就是说，当你需要更新数据库的参数类型符合这几种，可以使用 operate 方法，简化 JDBC 语句编写。

这样做的理由基于“将所有数据表字段尽可能都设计成字符串类型”：  
<http://www.jdon.com/jive/thread.jsp?forum=91&thread=23875>

## DaoCRUDTemplate API

本功能只有 Jdon 框架 5.1 以后才有。使用本功能可以无需你写 CRUD 的持久层代码了，大大减轻开发工作量。

在 Struts+Jdon +Hibernate3.2 的案例 samples\_hibernate.zip 中，无论任何对象的持久化，都可以使用如下一个 JDBC DAO 接口实现：

```

public void insert(Object o) throws Exception;

public void update(Object o) throws Exception;

public void delete(Object o) throws Exception;

```

```
public Object loadModelById(Class classz, Serializable Id) throws Exception;
```

这些 CRUD 方法已经在 Jdon 框架的 com.jdon.persistence.DaoCRUDTemplate 模板中已经实现，你只需直接调用就可以了。如你自己的 JdbcDaoImp 可以如下实现：

```
public class JdbcDaoImp extends DaoCRUDTemplate implements JdbcDao{  
    ....  
}
```

当然，如果要使用 DaoCRUDTemplate，还需要激活 CloseSessionInView，在 web.xml 配置 CloseSessionInViewFilter，如下：

```
<filter>  
    <filter-name>hibernateFilter</filter-name>  
    <filter-class>com.jdon.persistence.hibernate.CloseSessionInViewFilter</filter-class>  
</filter>  
<filter-mapping>  
    <filter-name>hibernateFilter</filter-name>  
    <url-pattern>/*</url-pattern>  
</filter-mapping>
```

这样可以实现一个请求结束后再进行 Session 关闭，这样，Hibernate3.2 缺省的懒加载功能就可以正常使用。更多可见：[5.1 新功能章节](#)。

## 多态、有态和单态

目前在 Jdon 框架 1.3 以上版本中，提供三种接口让你的应用服务 Service 类继承：

com.jdon.controller.pool.Poolable //对于主要很大的类推荐使用，使用对象池提高你的服务实例运行性能

com.jdon.controller.service.Stateful //有状态，当你需要处理有状态数据使用，如购物车

com.jdon.container.visitor.data.SessionContextAcceptable //需要获得容器的登陆信息，见下节。

上述三种接口选择根据具体业务需求决，SessionContextAcceptable 可以和前面两种混合使用，Poolable 和 Stateful 不能并列使用，只能二选其一。

5.6 版本以后，分别提供以上三种 Annotation 元注解，这样不用你的代码实现这些接口，从而减少所谓侵略性。

假设你的一个应用服务类的接口名为 MyServiceIF，类实现为 MyServiceImp，那么当客户端进行如下服务调用：

```
MyServiceIF ms = (MyServiceIF) WebAppUtil.getService("myService ", request);
```

这时，ms 这个实例产生和 MyServiceImp 所继承的上述三种接口有关系，如果 MyServiceImp 实现了 Poolable，那么 ms 就是 MyServiceImp 多态实例池中的任何一个；如果 MyServiceImp 实现了 Stateful 接口，那么，每个客户端每次获得的 ms 都是同一个 MyServiceImp 实例；如果 MyServiceImp 不继承任何接口，那么每次调用上述服务获得的 ms 实例都是一个新的实例，相当于 new 一个实例。

如果希望每次获得 ms 实例是单态实例，也就是是一个实例，直接通过上述 WebAppUtil.getService 语句是不行的，但是有一个变通办法，将 MyServiceImp 嵌入在其

他服务类中，例如嵌入一个叫 `OurService` 类中，例如：

```
public class OurService implements OurServiceIF {  
  
    private MyServiceIF myservice;  
  
    public OurService(MyServiceIF myservice) {  
        this.myservice = myservice;  
    }  
  
    public MyServiceIF getMyService() {  
        return myservice;  
    }  
}
```

这样，首先获得 `OurService` 实例：

```
OurServiceIF os = (OurServiceIF) WebAppUtil.getService("ourService ", request);  
MyServiceIF ms = os. getMyService();
```

这样，每次获得 `ms` 就是同一个实例，或者称为单态，原因是：`MyServiceImp` 是通过 `Ioc` 容器注射进入 `OurService` 中的，而 `Ioc` 容器每次注射的都是单态实例。

## 用户注册登陆

用户注册登陆系统是每个系统必须模块，主要包括两个功能块：用户资料注册和修改部分；用户登陆和访问权限 `ACL` 部分。

前一个部分主要是用户资料的增删改查 `CRUD`，可以使用 `Jdon` 框架轻易完成；如 `Jpstore` 中的 `SignAction` 类。

第二个部分有两种实现方式：基于容器和手工定制（`Container-based` vs. `custom security`），这两种实现方式都可由用户自己选择，这里主要说明一下，如何将用户登陆后的信息如用户 `ID` 共享给业务模型，例如：论坛发言者登陆后，发表一个帖子，这个帖子应该有一个用户 `ID`，如何将发言者的用户 `ID` 赋值给帖子里？依据不同的实现方式有不同的捷径：

手工定制就是开发者自己编制代码完成，为需要保护的资源设立 `filter`，或者在每个 `Jsp` 页面 `include` 一个安全权限检查 `Jsp` 模块，在这个方式下，用户登陆信息一般是保存在 `HttpSession` 中，那么在表现层继承 `ModelHandler` 实现一个子类，在这个 `Handler` 中能够访问 `HttpSession`，并取出事先保存在 `HttpSession` 中的用户信息如 `ID`。

当然，基于容器的实现方式也可以采取上述办法，例如 `JdonNews` 的 `NewsHandler` 类中的 `serviceAction` 方法代码，如下：

```
User user = (User) ContainerUtil.getUserModelAfterLogin(request);  
if (user != null) {  
    News news = (News) em.getModel();  
    news.setUser(user);  
}  
else
```

```
Debug.logVerbose(" not found user in any scope ", module);
```

这样，将用户登陆信息 `user` 直接赋值到 `news` 实例中。

这种方式是通用方式，缺点是需要专门实现一个 `ModelHandler` 类，需要编写代码，还有一个不需要编写代码的方式：

### 获得基于容器登陆用户 Principle

下面介绍在服务层能够直接获得基于容器的登陆用户 `UserPrinciple` 名称的方式，这种方式只适合 `POJO Service` 架构，而且必须是基于容器实现方式，相关基础知识点见：

<http://www.jdon.com/idea/jaas/06001.htm>

使用本方式目的在于简化表现层代码，在 `Jdon` 框架中，表现层基本都是通过 `jdonframework.xml` 配置实现，一般情况不必编码，如果不使用本方式，那么就必须自己实现一个 `ModelHandler` 子类，现在则不必，可以在业务服务层服务 `Service` 对象中直接获得 `request.getPrinciple()` 方法结果。

直接简单的使用方式：

1. 让 `Service` 对象继承 `com.jdon.container.visitor.data.SessionContextAcceptable`，该接口目前有两个方法：

```
void setSessionContext(SessionContext sessionContext);  
SessionContext getSessionContext ();
```

这样这个 `Service` 类必须完成接口上述两个方法，实际是 `Jdon` 容器在运行时，通过 `com.jdon.aop.interceptor.SessionContextInterceptor` 这个拦截器在运行时注射到被调用的这个 `Service` 对象中。

目前 `com.jdon.container.visitor.data.SessionContext` 中已经包含 `String` 类型的 `principleName`，这个值来源于 `request.getUserPrincipal()` 获得的 `Principal` 的 `Name` 值，如下：

```
Principal principal = request.getUserPrincipal();  
String principleName= principal.getName();
```

上述这段代码是在 `Jdon` 框架中执行，这样通过这种方式，在服务层服务对象中我们可以容易获得保存在 `SessionContext` 中用户登陆信息 `Principle Name`，使用这个方式虽然方便，但是必须有下列注意点：

1. 继承 `com.jdon.container.visitor.data.SessionContextAcceptable` 的服务层服务 `Service` 对象必须被客户端直接调用，调用方式是代码或配置方式都可以，客户端代码方式如下：

```
FrameworkTestServiceIF frameworkTestService = (FrameworkTestServiceIF) WebAppUtil.getService(  
    "frameworkTestService", request);  
frameworkTestService.getName()
```

或在 `jdonframework.xml` 中配置 `model` 部分指向 `frameworkTestService` 调用也可以，这是一种配置方式调用。

只有这个服务 `Service` 对象直接面对客户端调用，拦截器的注射功能才能将 `SessionData` 注射进入它的“体内”。

2. 访问服务层服务 `Service` 对象的客户端必须在容器安全体系保护下，一般这样客户端都是 `Jsp` 或 `Servlet/Action`，都会有 `Url`，例如 <http://localhost:8080/MyWeb/Admin/XXXX.do>，在 `web.xml` 以及配置 `/Admin/*` 都是必须被特地角色才能访问。

因为只有这样，`Web` 容器的 `Request` 对象中 `getUserPrinciple` 才会有值，这是 `J2EE` 规范规定的。载有角色值的 `Request` 访问 `/Admin/XXXX.do`，`XXXXX.do` 是 `Struts` 的一个 `Action`，再通过 `Action` 访问特定的服务 `Service` 对象。



在上述两种情况同时满足情况下，服务层服务 Service 对象中 sessionContext 中的 principleName 才会有该用户的登陆信息，注意这个登陆信息令牌可以是用户 ID 或用户名，取决于你容器配置如 JBoss 的 login-config.xml，只有密码验证通过 Web 容器才会有值。

一旦你在 Service 中获得 principleName，你就可以再通过查询数据库获得该用户的完整资料，从而将用户信息共享给所有业务服务层。

具体代码见 Jivejdon 3.0 的 com.jdon.jivejdon.service.imp. ForumMessageServiceImp。

## SessionContext 用处

上述容器登陆信息 Principle Name 通过保存在 SessionContext 中使得服务层服务对象能够获得用户的登陆信息。

SessionContext 是保存在 Web 容器的 HttpSession 中一个载体，当在服务层服务对象中需要缓存一些数据，这些数据的 Scope 是 Session 性质，这样可以通过 SessionContext 实现数据共享。Jdon 框架还提供 Stateful 提供另外一种与 Session 有关状态载体选择。

如果希望缓存或共享的数据的 scope 是 Request，可以使用 JDK 新语言特性 ThreadLocal，服务层的服务对象就能够和 Jdon 容器以外的 Request 请求相关发生联系，例如我们可以做一个 ServletFilter，将某些数据保存在 ThreadLocal 中，然后在服务层服务对象中从 ThreadLocal 中获取，这属于一般应用技巧，这里不详细描述，只是做一个应用提示。

如果缓存或共享的数据 scope 是 Application，就将其注册到 Jdon 容器中，因为 Jdon 容器的 scope 是 Application 的，这样整个服务层也可以访问。

目前 Jdon 框架通过 SessionContextSetup 将一些用户相关信息保存到 SessionContext，当应用系统通过 Jdon 框架调用任何服务时，将激活 SessionContextSetup，SessionContextSetup 的缺省实现是 com.jdon.security.web. HttpRequestUserSetup，

目前在 HttpRequestUserSetup 中，实现两个功能：将用户通过容器登陆的 Principle 和 remote Address 地址保存在 SessionContext，这样，在业务层中的所有服务将可以通过 SessionContext 获得这两个数值。

HttpRequestUserSetup 是通过在 container.xml 中配置激活的：

```
<component name="sessionContextSetup" class="com.jdon.security.web.HttpRequestUserSetup" />
```

如果你还有更多数值需要供业务层服务使用，可以提到这行配置，见[组件替换](#)。

例如：缺省 HttpRequestUserSetup 中是从 request 中获得用户登录的 Principle，如果你想自己实现用户登录，使用 ServletFilter 来实现登录验证，验证通过后，将验证后的用户信息保存在 Request 或 ThreadLocal 中，这样，你就需要实现自己的 SessionContextSetup，用来从 Request 或 ThreadLocal 中获取用户信息给 Jdon 框架。

## cookie 自动登录

本节是针对基于 Web 容器登录体系，login.jsp 登录页面如下：

用户	<input type="text"/>	自动登陆 <input checked="" type="checkbox"/>
密码	<input type="password"/>	<input type="button" value="登陆"/>
<a href="#">新用户注册</a> <a href="#">忘记密码?</a>		

通常基于容器的登录，在 login.jsp 中提交的 action 是 j\_security\_check，为了激活“自动登录”功能，我们需要使用自己的 servlet，然后再转发到 j\_security\_check，如下：

```
<form method="POST" action="<%=request.getContextPath()%>/login"
....
</form>
```

这里/login 是一个 Servlet，在 web.xml 中配制如下：

```
<servlet>
  <servlet-name>login</servlet-name>
  <servlet-class>com.jdon.security.web.LoginServlet</servlet-class>
  <init-param>
    <param-name>login</param-name>
    <param-value>/account/login.jsp</param-value>
  </init-param>
  <init-param>
    <param-name>logout</param-name>
    <param-value>/account/logout.jsp</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>login</servlet-name>
  <url-pattern>/login</url-pattern>
</servlet-mapping>
```

com.jdon.security.web.LoginServlet 是 JdonFramework 中的一个类，主要功能：

如果是从 login.jsp 提交，LoginServlet 判断自动登录是否选中，如果是，将用户和密码加密后保存在客户端 cookie 中，然后再转发到 j\_security\_check。

如果是直接/login?logout 调用，表示退出，我们应用的退出按钮连接需要指向这个连接，退出主要是 session 失效和 cookie 失效。

如果是直接无参数/login 调用，首先判断当前 cookie 是否保存用户和密码，如果是，使用这对用户密码发往 j\_security\_check 自动登录；如果不是，推出 login.jsp 页面。

LoginServlet 需要配置两个参数：login 和 logout，表示你的应用的 login.jsp 位置和 logout.jsp 位置，LoginServlet 将根据条件推出这些页面。

以上是配置之一，还有下面的 web.xml 配置。

在 web.xml 的 form-login-page 中，不要直接将登录页面/account/login.jsp 写入，这样自动登录功能就不能激活，或者说，通过前面步骤我们配置激活 cookie，cookie 中保存了我们的用户和密码，但是具体使用时还必须到 cookie 中查询，因此需要配置 form-login-page 为/login，也就是 LoginServlet。

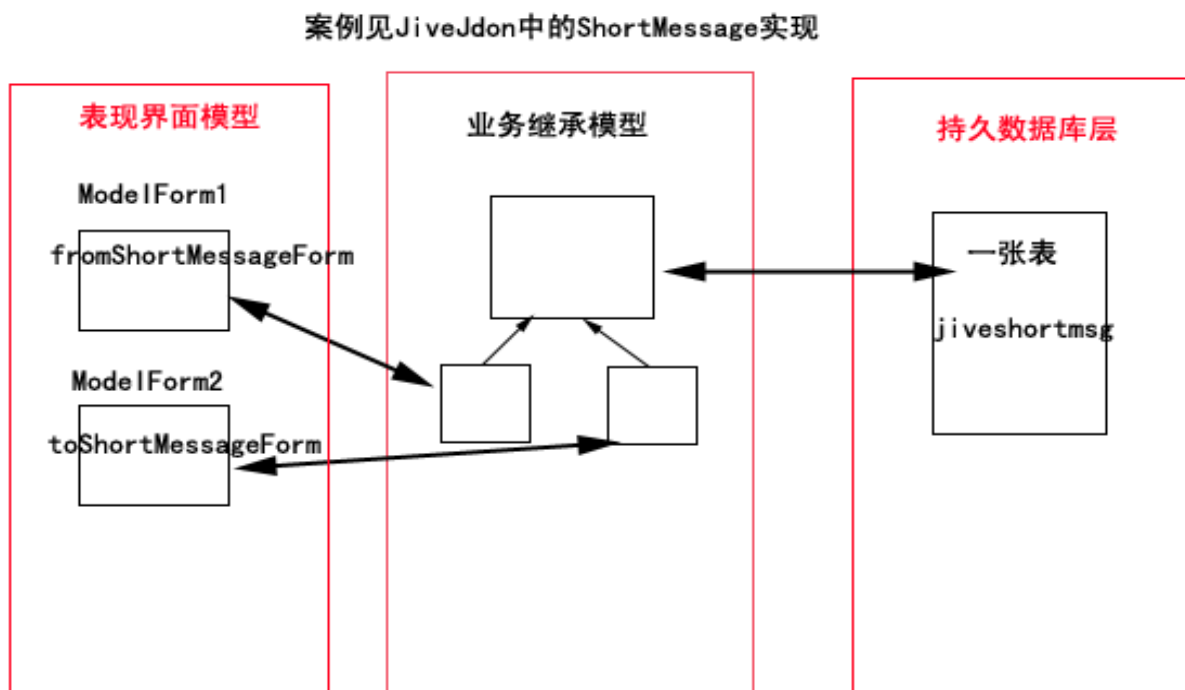
```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login</form-login-page>
    <form-error-page>/account/login_error.jsp</form-error-page>
```

```
</form-login-config>
</login-config>
```

## 继承模型最佳实践

Jdon 框架除了可以实现根据 Evans DDD 的关联模型以外，也可以方便实现继承关系的模型，下面是来自 JiveJdon 的继承模型最佳实践，具体案例见 JiveJdon 的 ShortMessage 代码。

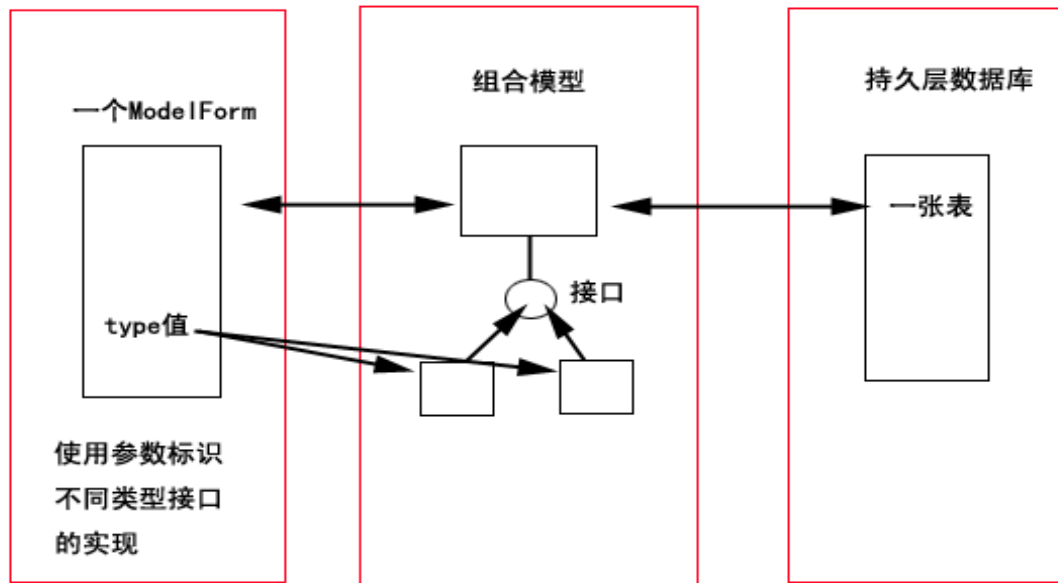
消息模型是一个父子继承关系，这样父子继承关系的模型如何使用 Jdon 框架来简单明了的实现，也就是不破坏领域层的继承关系，将这种继承关系延伸到非对象化的表现层和持久层，下面根据消息模型的实践总结为最佳实践如下图：



## 组合模型最佳实践

关注订阅模型是 JiveJdon 的一个功能，该模型的特点是 Subscription 根模型中有一个接口关联，而接口有不同实现，将来可能要扩展，但是在具体运行中，该接口肯定是其具体实现的一个子类，如何协调表现层模型来实现这一点，不破坏组合模型的特点和优点，JiveJdon 的关注 Subscription 模型提供一条最佳实践方式。具体见其代码。

案例见JiveJdon中的Subscription实现



struts-config.xml 的配置一个 ActionForm:

```
<form-bean name="subscriptionForm"
  type="com.jdon.jivejdon.presentation.form.SubscriptionForm" />
```

在 model.xml 模型的 CRUD 配置中,也配置 ActionForm 和 Model 一对一.

```
<model key="subscriptionId"
  class="com.jdon.jivejdon.model.subscription.Subscription">
  <actionForm name="subscriptionForm"/>
  <handler>
    <service ref="subscriptionService">
      <getMethod name="getSubscription"/>
      <createMethod name="createSubscription"/>
      <deleteMethod name="deleteSubscription"/>
    </service>
  </handler>
</model>
```

微妙之处在 `com.jdon.jivejdon.presentation.form.SubscriptionForm` 代码中.:存在 `subscribeType` 这个参数, 而模型 `Subscription` 中没有 `subscribeType` 这个参数, 只有数据库中有 `subscribeType` 这个字段,转换关系:

表现层: `subscribeType` <----> 模型中 `Subscribe` 类型 <----> 数据表 `subscribeType`  
 通过这种转换,可保证模型层使用 OO 面向对象方式设计,将数据完美表现为对象.

`SubscriptionForm` 中 `subscribeType` 和 `Subscribe` 转换主要两个方法如下:

```
/**
```

```
    * transfer Form to Model join fields into Subscribed;
```

```
    * Form 拷贝到 Model 时,将自动执行 Form 的 getSubscribe,将结果通过
    调用 Model 的 setSubscribe 方法,赋值到 Model 中.
```

```
    * @return
```

```

*/

public Subscribed getSubscribe() {
    return SubscribedFactory.create(subscribeType, subscribeId);
}

/**
 * transfer Model to Form separate Subscribed into fields;
 * 模型拷贝到 Form 时,将自动执行模型的 getSubscribe 方法,将其结果通过 Form
的 setSubscribe 方法赋值其中.
 * @return
 */

public void setSubscribe(Subscribed subscribe) {
    this.subscribeType =
SubscribedFactory.getSubscribeType(subscribe);
    this.subscriptionId = subscribe.getSubscribeId();
}

```

## Domain Event 模式

在 Evans DDD 实现过程中,经常会碰到实体和服务 Service 以及 Repository 交互过程,这个交互过程的实现是一个难点,也是容易造成失血贫血模型的主要途径。

Domain Event 提出解决方案, Jdon 框架提供的[异步观察者模式](#)为 Domain Event 实现提供更优雅的解决方案。

详细文章见: <http://www.jdon.com/jivejdon/thread/37289>

使用 [6.2 版本的领域消息](#)可以彻底实现领域模型和技术架构的耦合,领域模型任何事件只要通过消息就可以和技术架构比如持久化 领域服务进行交互,而且是异步的,没有任何事务单一线程。

## Startable 接口

框架提供 com.jdon.container.pico.Startable 接口, Startable 接口有两个方法:

start(); 当应用开始运行时, 激活这个方法;

stop(); 当应用 undeploy 或容器正常 shutdown 时, 激活该方法。

可以自己做一个类, 实现 Startable 接口, start 方法是在类的构造方法以后才执行, 可以用于启动一些重量任务, 如启动线程等, 而 stop 方法则是在应用从服务器中 undeploy 时自动执行, 可以在其中执行一些将内存中数据写入数据库持久化的任务。

## 版本新增或改变功能

### 1.X 版本

#### 1.4 版本

增加 ModelIF 接口，其意义等同于以前 Model 类，是继承 Model 类或实现 ModelIF 接口，可由框架应用者自行决定。

服务调用命令模式，见[这里](#)。

批量分页查询，提供新的批量读取数据库算法，增加批量查询时缓存的重用效率。

#### 1.5 版本

Fixed 一些 1.4 版本的 BUG。

update 02:解决主键类型是非 String 时，可能带来的问题，在 1.4 及其以前版本，如果主键类型是非 String，Service 接口中查询模型方法如 getMessage(String messageId)方法，该方法参数 messageId 必须为 String，这会带来初学者一些歧义，现在 1.5 以后这个方法参数 messageId 主键类型可以和模型主键类型统一了，如模型主键类型是 Long 方法，那么 Service 接口可以为 getMessage(Long messageId)。

在 JDK5.0 下编译通过，将 JF 应用案例全部在 JDK5.0 下调试通过。

暂时没有在 1.5 版本中加入 JDK5.0 特殊语法，换言之，1.5 版本还可以在 JDK1.4 平台编译通过。

1.5 版本推荐使用平台： JDK5.0 + JBoss 3.2.8

### 5.x 版本

#### 5.0 版本

升级到 JavaEE5.0 平台，本框架只能在 JDK5.0 以上平台编译和运行，引入了 5.0 高质量的并行并发概念，提高 Jdon 框架在高访问下的线程安全性和并发性。

增加对 EJB3 服务的支持，如果你的普通 JavaBean 架构需要升级到 EJB2/EJB3，无需更改客户端或表现层代码，直接修改 JdonFramework 配置即可。

## 5.1 版本

JdonFramework 5.1 主要变化是增加了 Hibernate3 支持，并且将原来一个 jdonFramework.jar 包分为三个包：jdonFramework.jar jdon-struts1x.jar 和 jdon-hibernate3x.jar

如果使用 struts+jdon+Hibernate 架构，这三个包需要完全使用。如果只使用 Jdon 框架作为业务层框架，只需要 jdonFramework.jar，可以使用 Jdon 框架的 IOC 管理功能，不过 Jdon 框架提供的 CRUD 流程简化将无法实现。

JdonFramework 5.1 重点是增加 Hibernate3 整合，特别是 Hibernate3 的懒加载支持，每个使用 Jdon+Hibernate 项目都需要在 web.xml 配置 CloseSessionInViewFilter，如下：

```
<filter>
    <filter-name>hibernateFilter</filter-name>
    <filter-class>com.jdon.persistence.hibernate.CloseSessionInViewFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>hibernateFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

这样可以实现一个请求结束后再进行 Session 关闭，这样，Hibernate 缺省的 Lazy=true 功能就可以正常使用。

懒加载功能可以大幅度提高 Hibernate 关联性能(前提是基于 DDD 分析设计)，是 Hibernate 必须使用的功能，但是单纯使用 Hibernate 却无法激活懒加载，致使很多人关闭懒加载 lazy="false"。

Spring 也提供了 Hibernate 的 OpenSessionInView 功能，但是它是将 Hibernate 的 Session 再表现层打开和关闭，而 JdonFramework 5.1 只是检查是否打开的 Session，如果有则关闭，这样缩短 Session 无谓开启时间，降低出错率,同时简化事务(Spring+Hibernate 架构 Spring 事务缺省是 read-only 只读，只有配置显式 create 方法事务为非 read-only 后，才能使用 Hibernate 保存创建新资料，非常不方便);

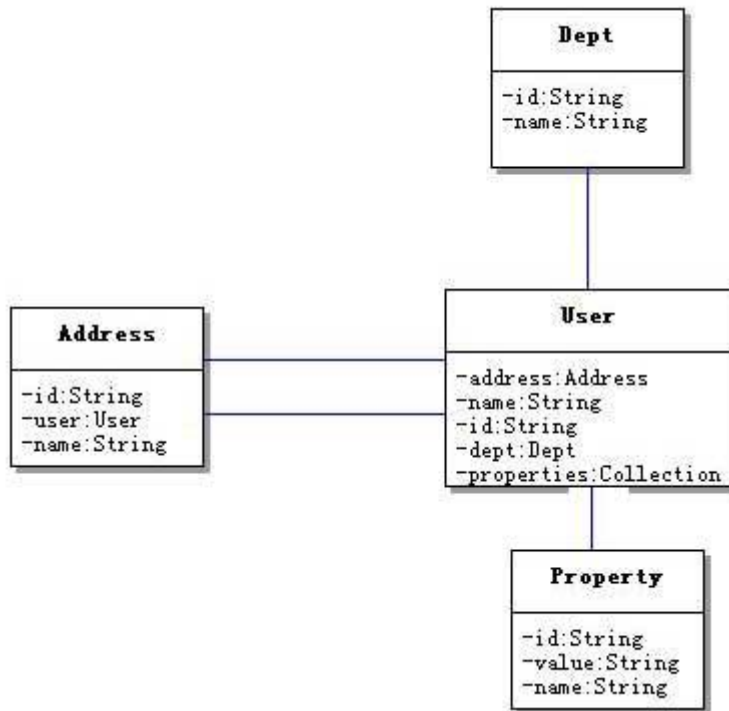
Struts+Jdon+Hibernate(简称 SJH)和 Struts+Spring+Hibernate(简称 SSH)最大的特点是：

- 1.SJH 对 Hibernate 无侵略性，Hibernate 全局配置还是在自己的 hibernate.cfg.xml 中配置，而不似 SSH 需要在 Spring 中配置。这样从设计上减少彼此依赖性，实现真正分层松耦合。jdon-hibernate3x.jar 可以单独使用。
- 2.SJH 以更少的代码快速实现增删改查和批量分页查询。
- 3.Jdon 框架内置缓存+Hibernate 二级缓存+懒加载最大化提高缓存使用效率，性能优异，有效实现数据库零访问和必要访问。

JdonFramework 5.1 推出同时，也推出 struts+jdon+Hibernate3 的整合应用案例  
[http://www.jdon.com/jdonframework/download/samples\\_hibernate.zip](http://www.jdon.com/jdonframework/download/samples_hibernate.zip)

该案例领域模型图：





## 5.5 版本

将框架与 `HttpRequest` 和 `HttpSession` 分离，框架不一定需要用在 WEB 中，也可应用在 Application，直接运行源码包中的 `runTest.bat` 测试。

Jdon 框架可以只作为一个 IOC/DI 框架，使用在 JavaEE 或普通 Java 或 J2ME/JME 中，拓展用途。

5.5 版本经过严格的并发性能测试，解决了以往框架中可能存在的内存泄漏漏洞，使用 `ThreadLocal` 以及 `java.concurrent Callable Future` 等 JDK5.0 以上新的并行功能，增强了并行计算能力。

5.5 版本 10 分钟内发出近万次并发请求，没有任何报错和问题，测试客户端见 `JiveJdon3.5` 源码包中 `jmeterTest.jmx`，可以使用 Jmeter 打开，服务器端运行 Jprofiler 和 JBoss 服务器，注意：进行并发测试时，需要将 `concurrent_myaspect` 和 `concurrent_web.xml` 提到原来的 `myaspect.xml` 和 `web.xml`，失效 JiveJdon 的防 SPAM 功能。

5.5 版本的源码经过并发性能重构，增强稳健型和快速性，是目前 Jdon 框架中最稳定的版本。

限制 `sessionContext` 中最多只能放置 10 个对象，超过就自动清楚，这样，不要将重要状态放入 `sessionContext` 中，也不要放置超过 10 个对象，`sessionContext` 最大个数 10 设置是在 `container.xml` 中。

使用 ehcache 作为 Jdon 框架缺省的缓存，以增强兼容性和伸缩性，分布式 Cache Terracotta <http://www.terracotta.org/> 可以将 encache 自动进行分布到多台服务器，这样，Jdon 框架也能支持分布式缓存和计算(无需借助 EJB <http://www.jdon.com/article/34888.html>)。

Jdon 框架部署在集群环境下，由于集群环境，`HttpSession` 将被在多台服务器间同步，而 Jdon 框架利用 `HttpSession` 进行动态代理优化缓存，这时可能会引起



**NotSerializableException** 问题，采取办法可以失效优化缓存。

确认用 **winrar** 打开 **JdonFramework.jar**，将 **META-INF** 的 **container.xml** 拖到某个文件夹下。将其中下面这一行中 **true** 改为 **false**，保存后再将 **container.xml** 拖回 **winrar** 中即可。

```
<component name="httpSessionVisitorFactorySetup"
class="com.jdon.container.visitor.http.HttpSessionVisitorFactorySetup">
<constructor value="40" />
<!-- it is count of the cached instances that saved in a sessionContext instance, -->
<constructor value="true" /> <!-- disable all session cached except SessionConext -->
</component>
```

正常单机情况下，也可以将这一设置改为 **false**，可以节省内存，经过测试，稳定运行后性能差异不大。

## 5.6 版本

5.6 版本主要是增加 **Annotation** 元注释来替代原来一些接口，减少框架的侵略性。

**Jdon** 框架默认提供了三种拦截器，当然你可以自己增加定制，这三种拦截器分别是对象池 **Session** 周期和有态，以前为让你的 **service** 实现对象池等这三个功能，需要特别实现三个接口 **Poolable** **SessionContextAcceptable** **Stateful**，而现在有一个可替代方法，就是使用元注解 **Annotation**，例如：

```
@Poolable
```

```
public class ForumMessageShell implements ForumMessageService
```

表示 **ForumMessageShell** 将以对象池形式存在，这样对于一个主要代码很多的类，可以起到提高性能，防止资源无限消耗。对象池优点见：<http://www.jdon.com/jivejdon/thread/35191.html>，注意不要将小类做成 **Pool**，因为对象池本省也损耗一定性能。

另外一个元注释是 **Model**，原来所有 **Domain Model** 需要实现接口 **ModelIF** 或继承 **Model** 类，现在可以使用 **@Model** 来替代了，如：

```
@Model
```

```
public class Account{ }
```

目前 **Jdon** 框架是 **Annotation** 和 **XML** 配置组合使用，**Jdon** 框架并没有使用 **Annotation** 全面替代 **XML** 配置文件，因为我们对于元注解 **Annotation** 的观点是：适当但是不可乱用，是一把双刃剑。当所有组件都可以自由拆开 **IOC/DI**，运行时也可自由指定顺序 **AOP**，那么我们就能够更快地跟随变化，甚至无需重新编译整个项目，只要带一个组件到现场，**XML** 配置改写一下，就能上线运行，注意，我这里提到了 **XML** 配置，如果你完全使用 **Annotation** 就达不到这个目的。

<http://www.jdon.com/jivejdon/forum/messageList.shtml?thread=35373&message=23119938#23119938>

## 5.8 版本

增加基于 **Http** 的远程 **remote** 远程调用，可以方便开发出基于 **RIA** 富客户端的多层架

构 C/S 系统。见案例 remote\_javafx

案例 remote\_javafx 是使用 RIA 技术 JavaFX + Hessian + Jdon 框架的一个 Demo 案例，开发简要步骤：

在 web.xml 配置 Hessian Servlet Proxy 如下：

```
<servlet>
<servlet-name>Hessian2Jdon</servlet-name>
<servlet-class>com.jdon.bussinessproxy.remote.HessianToJdonServlet
</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>Hessian2Jdon</servlet-name>
<url-pattern>/remote/*</url-pattern>
</servlet-mapping>
```

这样，当远程客户端调用 <http://localhost:8080/remote/helloService>，那么这个 Servlet 将在 jdonframeowrk.xml 查找服务名称为 helloService 的服务：

```
<app>
  <services>
    <pojoService name="helloService" class="sample.HelloServiceImpl"/>
  </services>
</app>
```

远程客户端代码如下：

```
HessianProxyFactory factory = new HessianProxyFactory();
HelloService _service = (HelloService)
factory.create(HelloService.class, _url);
_service.hello(s);
```

客户端调用 HelloService 的 hello 方法时，将激活 HelloServiceImpl.hello 方法。Demo 在线演示网址：<http://www.jdon.com:8080/jdonremote/>

## 6.x 版本

### 6.0 版本

（一）增加 Annotation 注射功能， @Service 或 @Component 替代了原来 XML 配置：

```
<pojoService name="给自己类取的名称" class="完整类的名称"/>
或者在具体类代码上加入： @Service("给自己类取的名称")

<component name="给自己类取的名称" class="完整类的名称"/>
或者在具体类代码上加入 @Component("给自己类取的名称")
```

Service 和 Component 都表示组件构建概念，可以是一个类，也可以是一个类为主的多个类，如果这个组件供客户端调用，那么就称为服务。两者在 Jdon 框架中没有区别，

只是使用时称呼不同而已。

@Component(“给自己类取的名称”)也可以简化为@Component，这样，Component 的名称就是该类的完整类名称(getClass().getName());

@Service 则不可以这样，因为是供 Jdon 框架容器以外的外部客户端调用的，一定要取一个名称。

因为需要被客户端调用，那么就要指定服务的名称，所以，@Service 与@Component 的区别就是，@Service 必须规定一个名称@Service(“给自己类取的名称”);而@Component 则没有 name 属性，缺省是类名称 getClass().getName()名称，如果客户端有时需要临时直接调用@Component 标注的组件，也可以按类名称调用。

开发 Jdon 框架变得异常简单，只需要两步：

第一步：将存在依赖关联关系的两个类用@Service 或@Component 标注：

@Poolable

@Service("helloService")

```
public class HelloServiceImpl implements HelloService{
    UserRepository userRepository;
    public HelloServiceImpl(UserRepository userRepository){
        this.userRepository = userRepository;
    }
}
```

..

}

@Component

```
public class UserRepositoryInMEM implements UserRepository {
```

客户端调用代码:

```
HelloService helloService = (HelloService) WebAppUtil.getService("helloService", req);
String result = helloService.hello(myname);
```

无需 jdonframework.xml 配置文件和相关 XML 配置。全部演示代码见 JdonFramework 源码包目录下 examples/testWeb

(二).增加@Singleton Annotation，缺省客户端调用 getService 时，每次请求获得一个新的 Service 实例，也可以使用@Poolable 对象池来对代码比较大的 Service 类进行性能提高，现在又提供了另外一种方法，使用单例@Singleton。

单例和对象池区别主要是：对象池可以进行实例最多个数控制，这样，能够保护服务器不会被尖峰冲击造成资源耗尽。

在 Service 服务代码比较简单小的情况下，Service 实例三个创建方式：单例、对象池以及每次请求生成一个新实例，在性能上几乎没有太大差别，已经使用框架源码包中 examples 目录下的 testWeb 经过测试了。

(三).没有将有关 Action 的配置转为 Annotation，原来的配置如下继续保留：

```
<model key="username"
        class="com.jdon.jivejdon.model.Account">
```

```

<actionForm name="accountForm"/>
<handler>
    <service ref="accountService">
        <initMethod    name="initAccount"/>
        <getMethod     name="getAccountByName"/>
        <createMethod name="createAccount"/>
        <updateMethod name="updateAccount"/>
        <deleteMethod name="deleteAccount"/>
    </service>
</handler>
</model>

```

因为这个配置是有关 MVC 表现层，而表现层不是 Jdon 框架的核心，所以，Jdon 框架核心是业务层，对 Domain Model 和 Service 服务组件进行管理，所以，不能在业务层耦合具体表现层。上述配置可以认为是表现层和业务层的桥连接配置。

## 6.1 版本

增加异步事件处理功能，可以在当前处理过程同时，抛出另外一个过程处理其他事务，不影响当前主要处理过程，例如，发送邮件是一个很费时的过程，如果一个业务过程需要发送 Email，那么将发送 Email 这件事情通过异步，放到另外一个线程中处理，从而能实现处理效率性能的提供。

6.1 版本根据 Jdk 6.0 的并发 Concurrent 模型实现了异步事件处理功能 com.jdon.async.EventProcessor。

该类有两个构造参数，第一个是你分配几个线程同时来处理的异步事件，一般是设置一个，Queue 队列性质，一对一，如果你希望多个线程并发同时处理，发挥并发性能，那么可以选择更大数字，不过要注意，你的业务是否能够被多个线程同时执行，比如 Email 发送如果多个线程发送，那么就多发几次，一次就够了，因此设置为 1。

第二个参数，是允许同时几个线程在处理事务，因为你是将一个个事件或消息逐个放到 EventProcessor 队列中，如果每个事件或消息处理起来很费时，在之前事件消息还没处理完，你是否同意系统再启动其他线程处理后面的事件消息，这个参数需要根据你的硬件系统处理能力而定，如果处理能力不高，那么设置小一些。

6.1 版本还基于 EventProcessor 提供了观察者模式，用来实现对模型中数据变动的监测。

通常模型中数据变动，我们是通过直接关联或依赖关系，来通知另外一个模型同时变动，如下：

```

void changeValue(OtherModel om){
    this.value = "new Value";
    om.setValue("new Value too");//通过这个方法通知另外一个模型修改
}

```

这种方式适合与核心模型关系紧密的模型使用，比如同属于一个聚合模型边界内的模型对象们，但是现实中，还有一些关系并不如聚合关联那么紧密，是和核心模型有关联，但是是一种非常松散的关联，如何实现他们之间变动事件的传递？

使用观察者模式，传统的 JDK 观察者 API 实现是同步的，如下：

```

void changeValue(OtherModel om){
    this.value = "newValue";
    setChanged(); //设置变化点
    notifyObservers();//通过这个方法通知观察者事件
    doothers(); //必须等上面观察者实现了其响应行为才执行本方法。
}

```

而使用 `com.jdon.async.observer.ObservableAdapter` 和 `com.jdon.async.observer.TaskObserver` 则可以实现异步的观察者，这样，上面方法 `dooters` 就不必等待观察者 `Observer` 做完它的事情才能继续，两者同时运行。

异步观察者模式具体使用见 JiveJdon3.7 版本中的 Subscription 关注订阅模型，当主题有新内容时，通知关注者，我们在 `ForuThread` 的 `addNewMessage` 方法中加入观察激活点，当有人在这个主题下发新回帖时，激活这个观察点，从而异步激活订阅通知处理功能，由它来检查哪些人关注订阅了这个主题，然后给他们逐个发送消息或邮件，这个过程就不会影响发新回帖处理过程，保证发新回帖处理过程快速完成。

异步观察者模式步骤总结：

1. 继承 `TaskObserver`，实现其 `action` 方法，这是激活后所要实现的方法。
2. 将观察者 `TasjObserver` 加入 `ObservableAdapter`。
3. 将设置观察点，在被观察或监听的类中，调用 `ObservableAdapter`，在具体激活方法中调用 `ObservableAdapter` 的 `notifyObservers` 方法。

上面三个部分是松耦合，可以分别在代码三个地方执行，有时也可以合并一起：

```

private void sendComposer(EmailTask emailTask) {
    //创建一个观察者 继承 TaskObserver
    EmailTaskListener emailTaskListener = new
        EmailTaskListener(emailTask);
    //创建一个 ObservableAdapter
    ObservableAdapter subscriptionObservable = new
        ObservableAdapter(ep);
    //将观察者 TasjObserver 加入 ObservableAdapter
    subscriptionObservable.addObserver(emailTaskListener);
    //激活出发
    subscriptionObservable.notifyObservers(null);
}

```

关于领域模型设计和相关监视事件，见有关 Qi4j 讨论，Jdon 框架和 Qi4j 一样已经考虑到此类实践问题：<http://www.jdon.com/jivejdon/thread/37186>

## 6.2 版本 AOP

6.2 版本架构改变比较大，最大变化是增强了 AOP 功能，具体如下：

1. 使用 `CGLIB` 替代 `JDK` 的动态代理，性能有所提高：

<http://www.jdon.com/jivejdon/thread/37330>

增强了 AOP 功能，提供基于 `Annotation` 的方法拦截功能，但是 AOP 功

和 Spring/Aspect 的有些区别，主要是在被拦截类进行定义，如下：

```
@Introduce("c")
public class A implements AInterface {

    @Before("testOne")
    public Object myMethod(@Input() Object inVal, @Returning() Object returnVal) {
        System.out.println("this is A.myMethod is active!!!! ");
        return inVal + "myMethod" + returnVal;
    }

    @After("testWo")
    public Object myMethod2(Object inVal) {
        System.out.println("this is A.myMethod2 is active!!!! ");
        return "myMethod2" + inVal;
    }

}
```

通过元注解@Introduce 引入拦截器 c，c 是一个组件的名称，用 Component(“c”)定义的类，@Before("testOne")表示 c 这个类的 testOne 方法将在当前方法 myMehtod 之前执行，@After 则表示是以后执行。

使用这种引入式定义的拦截器 c 类不需要特别其他配置，只是一个普通组件而已，如下：

```
@Component("c")
public class C {

    public Object testOne(Object inVal) {
        System.out.println("this is C.testOne " + inVal);
        return " interceptor" + inVal;
    }

    public Object testWo(Object inVal) {
        System.out.println("this is C.testWo ");
        return inVal + " interceptor";
    }

}
```

6.2 版本还提供了一种 Around 方式的拦截，就是在包围原始方法前后都执行的方法，引入的类方式和 Before 和 After 类似：

```
@Introduce("aroundAdvice")
public class D implements DInterface {

    @Around
    public Object myMethod3(Object inVal) {
```

```

        System.out.println("this is D.myMethod3 is active!!!! ");
        return "myMethod3" + inVal;
    }
}

```

但是拦截器 `aroundAdvice` 写法就不是普通类的写法，需要实现接口 `org.aopalliance.intercept.MethodInterceptor`，完成其方法，同时必须有类 Annotation：`@Interceptor`，整个内容如下：

```

@Interceptor("aroundAdvice")
public class AroundAdvice implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.print("\n\n this is AroundAdvice  before \n\n");

        if (isAdviceAround(invocation.getThis().getClass()))
            System.out.print("\n\n this is AroundAdvice  myMethod3 \n\n");

        Object o = invocation.proceed(); //这句关键表示执行原始方法
        System.out.print("\n\n this is AroundAdvice after \n\n");
        return o;
    }
}

```

以上 6.2 版本新增的 **Before After Around** 拦截方式的编写规则，使用 Java 代码替代复杂的 `ponitcut` 规则表达式。简单易用。

另外还有一种和 Spring 类似，在拦截器类中配置被拦截的点 `Pointcut` 的方式，使用这个方式，就无需象上面在被拦截的地方进行注解，解放了被拦截类，而上面这种写法是解放了拦截类，灵活应用。

```

@Interceptor(name = "myInterceptor", pointcut = "a,c")
public class MyInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation methodInvocation) throws
java.lang.Throwable {
        System.out.print("\n\n this is MyInterceptor  before \n\n");
        if (methodInvocation.getMethod().getName().equals("myMethod")) {
            System.out.print("\n\n this is MyInterceptor for a.myMethod  \n\n");
        }
        Object o = methodInvocation.proceed();
        System.out.print("\n\n this is MyInterceptor after \n\n");
        return o;
    }
}

```

拦截器必须实现 `MethodInvoker` 接口，而且必须标注 `@Interceptor(name = "myInterceptor", pointcut = "a,c")`，其中名称 `name` 是拦截器自己的名称，而 `pointcut` 是被

拦截类的@Component(“a”)或@Component(“c”)。

如果拦截所有对象，那么就直接@Interceptor(); 没有任何参数，不过这个拦截所有对象的意思不是在所有对象只要被调用就激活，而是客户端通过 WebApp 进行对 Jdon 框架通过 Annotation 标注的这些 Component 或@Service 时才会激活。

更多见 [AOP 章节](#)。

## 6.2 版本领域事件消息

见前面[领域模型章节](#)。

增加了组件生命周期，组件可以在当前 Web 应用从服务器中 undeploy，或者当前服务器也就是 Web 容器 shutdown 正常关闭时，将自动执行组件类中的 stop()方法，我们可以在 stop 方法中将一些没有保存持久化的数据进行保存持久，防止丢失。

组件类只要实现接口：com.jdon.container.pico.Startable，完成其中 start()和 stop()方法即可。

以 JiveJdon4 中的帖子浏览数为案例，每个帖子每次阅读浏览都要计数一次，如果每次技术都要保存数据库，无疑在大浏览量情况下会加载数据库负载，这时我们将浏览次数保存在内存中，也就是组件类的字段中即可，然后每隔一定时间比如一个小时保存一次数据库，如果遇到应用维护需要正常 redeploy 或服务器关闭，我们就在 stop 方法在那个，把内存中计数值一次性保存到数据库中。

见 com.jdon.jivejdon.manager.listener.ThreadViewCountManager 类代码。

## 6.4 版本引入 Disruptor

引入快速并发框架 Disruptor，该框架在 LMAX 成功应用，每秒处理 6 百万订单，一微秒的延迟获得 100K+的吞吐量。

领域事件是 DCI 架构的一种实现方式。

## 6.5 版本完整 DCI 框架

6.5 版本将原来的 JdonFramework.jar 拆解，分成以业务领域层为核心的新的 JdonFramework.jar 和 JdonAccessory.jar 等配件包，该包主要包含 Struts1.x JdbcTemp 和 Hibernate 和远程访问 remote-hessian 等包。

同时在 6.5 框架的演示软件 SimpleJdonFrameworkTest.rar 中引入道友 oojdon 开发的 JdonMVC，这是一个类似 SpringMVC 3 的 REST 兼具 MVC 框架，简单而优雅。

见：[关于将 Jdon 框架提升为 DCI 框架的设想](#)

6.5 版本引入了 com.jdon.domain.dci.RoleAssigner，它是一个角色分配器，可以向任何模型中注入任何接口 (Mixin)

当使用 RoleAssigner，我们就没有必要从带有元注释 @Introduce(“modelCache”) 和 @Around 的仓储中首先获得一个模型对象。RoleAssigner 可以手工对任何一个模型对象从



外部进行事件注入或角色分配。提高了使用灵活度。

在 6.5 版本上，我们可以有不同风格的 DCI+Domain Events 实现：当我们有了领域模型对象时，可以通过其领域事件实现功能；当我们没有领域模型对象时，我们可以通过 RoleAssigner 对一个新构造的领域模型对象注入事件发送者角色，也可以使用领域事件实现各种功能。

## 技术支持

无论 Jdon 框架本身还是使用 Jdon 框架开发的任何系统出现问题，都可以在论坛

<http://www.jdon.com/jivejdon/forum.jsp?forum=61>

发生错误后，请按下面步骤贴出错误：

1. 需要打开 Jboss/server/default/log/server.log

2. 键入搜索" ERROR "，注意 ERROR 两边各有一个空格，找到第一个错误，那是错误起源，贴到论坛中。

### Weblogic + Oracle 环境下使用注意点

注意两点：

1. Oracle 字符串类型很特殊，它和 Java 的 JDBC 字符串类型有些特别，Oracle 字符串如果是 10 位，而 Java 传给它是两个字母 ab，你必须将剩余 8 个字母用空格填满，否则，出现“查询页面不显示记录，但是可以向数据库里插入记录”现象。

建议 oracle 数据库主键使用 Number，模型 Model 主键使用 Long 即可。

2. log4j 需要正常在 Weblogic 中显示，配置不是非常容易，按本[说明书相应章节](#)耐心配置。

总之，试验一个 Jdon 框架技术应该分开几个环节来分别测试：

1. 使用 Jdon 框架推荐的环境测试。
2. 使用自己特殊的数据库测试。
3. 使用自己特殊的服务器测试。

从这几个环境中，可以分别了解差异产生的原因。

### 开发环境配置

为方便初学者快速入手使用 Jdon 框架开发 J2EE/Java EE 系统，现将开发环境配置和软件下载陈述如下：

下载 Java 运行环境 也就是 JDK1.4 或 JDK5.0

下载 Eclipse IDE 开发软件，因为 Eclipse 插件非常多，安装后不一定能用，可在本站 VIP 下载区下载完整立即可用的 [Eclipse 压缩包](#)。

下载运行环境，使用 Tomcat/JBoss/Weblogic 等等都可以，本站提供 JBoss + MySQL 的[现成压缩包](#)。注意，需要将最新版本的 Jdon 框架拷贝进去，方法见前面[安装章节](#)。

以上开发环境安装完成后，你可打开 Jdon 框架的案例项目进行研究学习，祝你成功。

## 配置启动 Jdon 框架

如果使用 annotation 元注释，没有 xml 配置，则无需配置启动。

(1) 配置 jdonframework.xml 在 struts-config.xml 中。

```
<plug-in className="com.jdon.strutsutil.InitPlugIn">
  <set-property property="modelmapping-config" value="jdonframework.xml 完整路径" />
</plug-in>
```

如果没有使用 Struts，直接在 Web.xml 中配置，见前面配置章节

(2) 将 jdonframework.xml 打包到你的 Web 项目。

否则，后台控制台会出现：

[InitPlugIn] looking up a config: jdonframework.xml

[InitPlugIn] cannot locate the config:: jdonframework.xml

## 打包部署

有两种方式打包成部署文件.war 或.ear。

第一：如果是 Eclipse 工具，使用 Ant 进行打包，在本系统源码中已经编制好 build.xml 文件，更改其中 J2EE 服务器为的路径后，即可直接运行 Ant 打包。

第二：通过 JBuilder 直接打包，点本项目的 Web 项目，选择 make 即可，将自动出现 xxx.war 部署包（如果你是 Tomcat 就不会出现 war 文件，只能安装 Tomcat 方式配置）。

注意，将 Web 项目中去除任何包，也就是说 WEB-INF/lib 下没有任何包，Struts 和 JdonFramework 包都放置在 JBoss 的 server/default/lib 下。

如果出现与 tld 相关的错误，如：

File "/WEB-INF/MultiPages.tld" not found

解决：在 web.xml 中应该有下面两行 tld 配置：

```
<taglib>
  <taglib-uri>/WEB-INF/MultiPages.tld</taglib-uri>
  <taglib-location>/WEB-INF/MultiPages.tld</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/WEB-INF/TreeView.tld</taglib-uri>
  <taglib-location>/WEB-INF/TreeView.tld</taglib-location>
</taglib>
```

并且确保 /WEB-INF/MultiPages.tld 和 /WEB-INF/TreeView.tld 下文件存在，这两个 MultiPages.tld 和 TreeView.tl 在 JdonFramework.jar 中的 META-INF/tlds 中，可手工拷贝过去。

使用 JBuilder2005 直接打包时，缺省是不会将所有的编译好 class 打包进去的，同时也不会将 jdonframework.xml 等资源文件打包进去，我们需要经过特别步骤处理：

选择项目的 Web 模块如 testweb，右键点选属性，选择其中的 Content，点选 include all classes and resources，在 Content 子节点中，选择 Dependencies，将所有包都 exclude all，因为这些包我们事先已经放置在 JBoss 的 server/default/lib 下，不需要在本 Web 模块中携

带。

确定后，重新编译一次项目，检查 Web 模块如 testweb 下的 testweb.war 文件，展开后，检查是否有 jdonframework.xml，如果没有，接着如下步骤：

再次选择项目的 Web 模块如 testweb，右键点选属性，选择 Content，这次选择 include speified filtered file and resource，然后点按 Add Filters，输入 \*\*/\*.\*, 表示全部资源，确定。

再到 Add Filters 下面选择 Add File，将 jdonframework.xml 等资源文件加入。

再次编译整个项目，检查 war 文件，应该所有 class 和资源文件都包括其中了。

更多详细错误，打开 JBoss 日志文件：server/default/log/server.log，找到 server.log 中第一个 ERROR 信息，分析原因，或者贴到 Jdon 框架开发论坛中。

## 运行案例

打开浏览器，键入 <http://localhost:8080/testWeb/index.jsp>

我们将 index.jsp 导向真正执行/userListAction.do

网上实时演示网址：<http://www.jdon.com:8080/testWeb/>

当在 JBoss 或 Tomcat 控制台 或者日志文件中出现下面字样标识 Jdon 框架安装启动成功：

<===== Jdon Framework started successfully! =====>

## 性能压力测试

可见 jivejdon 的压力测试：

<http://code.google.com/p/jivejdon/source/browse/#svn%2Ftrunk%2Fjivejdon%2Fdoc%2FperformanceTest>

用户自己对 Jdon 框架应用案例的测试心得：

<http://www.jdon.com/jivejdon/thread/32894.html>

全文完