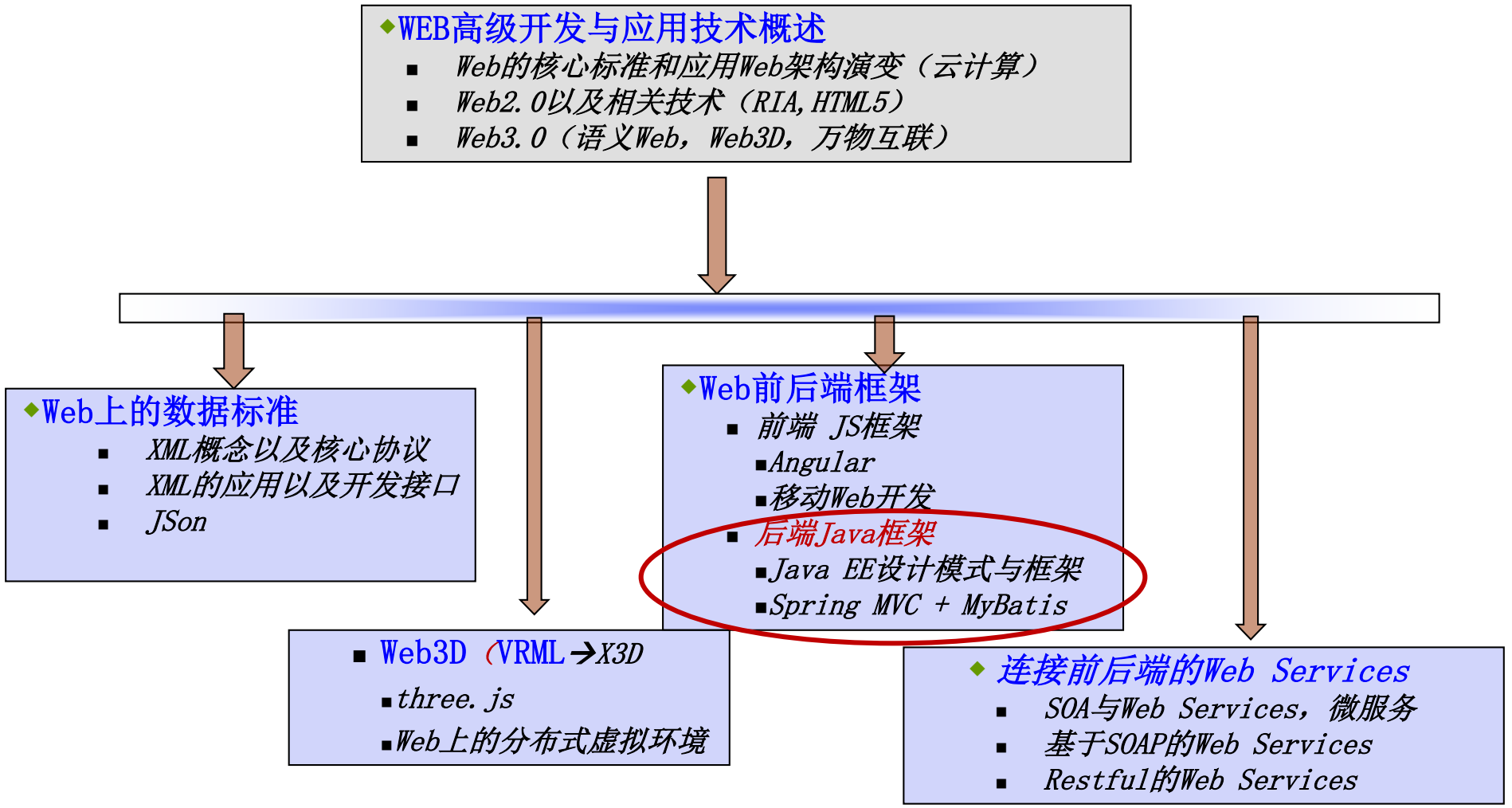
Advanced Web Technology

高级 Web 技术

Java 后端框架

课程内容组织


WEB高级开发与应用技术概述
Web的核心标准和应用Web架构演变（云计算）
Web2.0以及相关技术（RIA,HTML5）
Web3.0（语义Web，Web3D，万物互联）
Web上的数据标准
XML概念以及核心协议
XML的应用以及开发接口
JSon
Web3D （VRML→X3D
three.js
Web上的分布式虚拟环境
Web前后端框架
前端 JS框架
Angular
移动Web开发
后端Java框架
Java EE设计模式与框架
Spring MVC + MyBatis
连接前后端的Web Services
SOA与Web Services，微服务
基于SOAP的Web Services
Restful的Web Services

内容提要

- **Java EE设计模式与框架概述**
- Web层框架- **Spring MVC（Spring Boot）**
- 业务逻辑层框架-**Spring**
- 持久层框架-hibernate, **MyBatis**

Advanced Web Technology

Java EE设计模式与框架概述

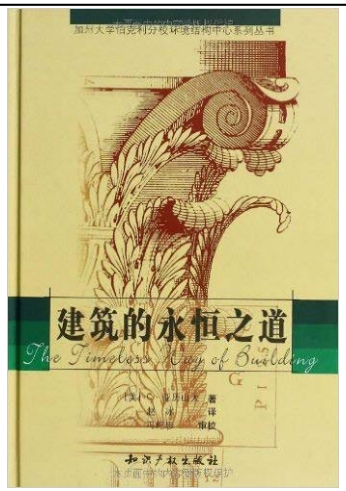
设计模式与框架

- **面向对象的重要设计原则**

设计模式与框架

- **概念**

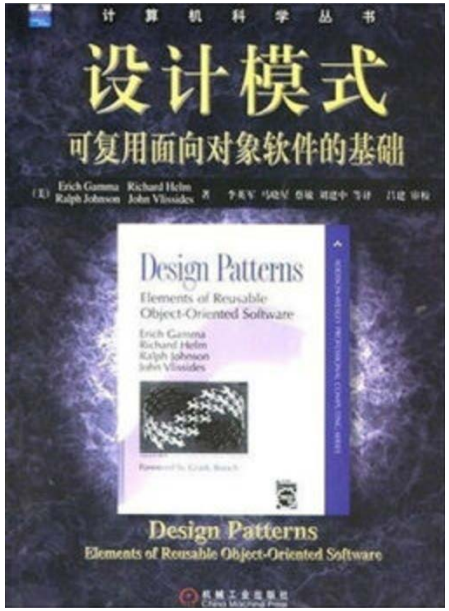
设计模式 (**design pattern**) 是对面向对象设计中反复出现的问题的解决方案。



- 一个设计模式命名、抽象和确定了一个通用设计结构的主要方面，这些设计结构能用来构造可复用的面向对象设计。
- 设计模式确定了所包含的类和实例，他们的角色、协作方式以及职责分配。
- 每一个设计模式都集中在一个面向对象设计问题或者设计要点，描述了什么时候使用他，在另一些设计约束条件下是否还能使用，以及使用的效果和如何取舍。

设计模式与框架

- 运用设计模式的好处
  - 直接提供可供考虑的问题解决方案
  - 使代码具有一致性
  - 帮助确定支持复用的适当粒度
  - 提高设计灵活性，使设计更适于复用，更加健壮，具有可扩充性

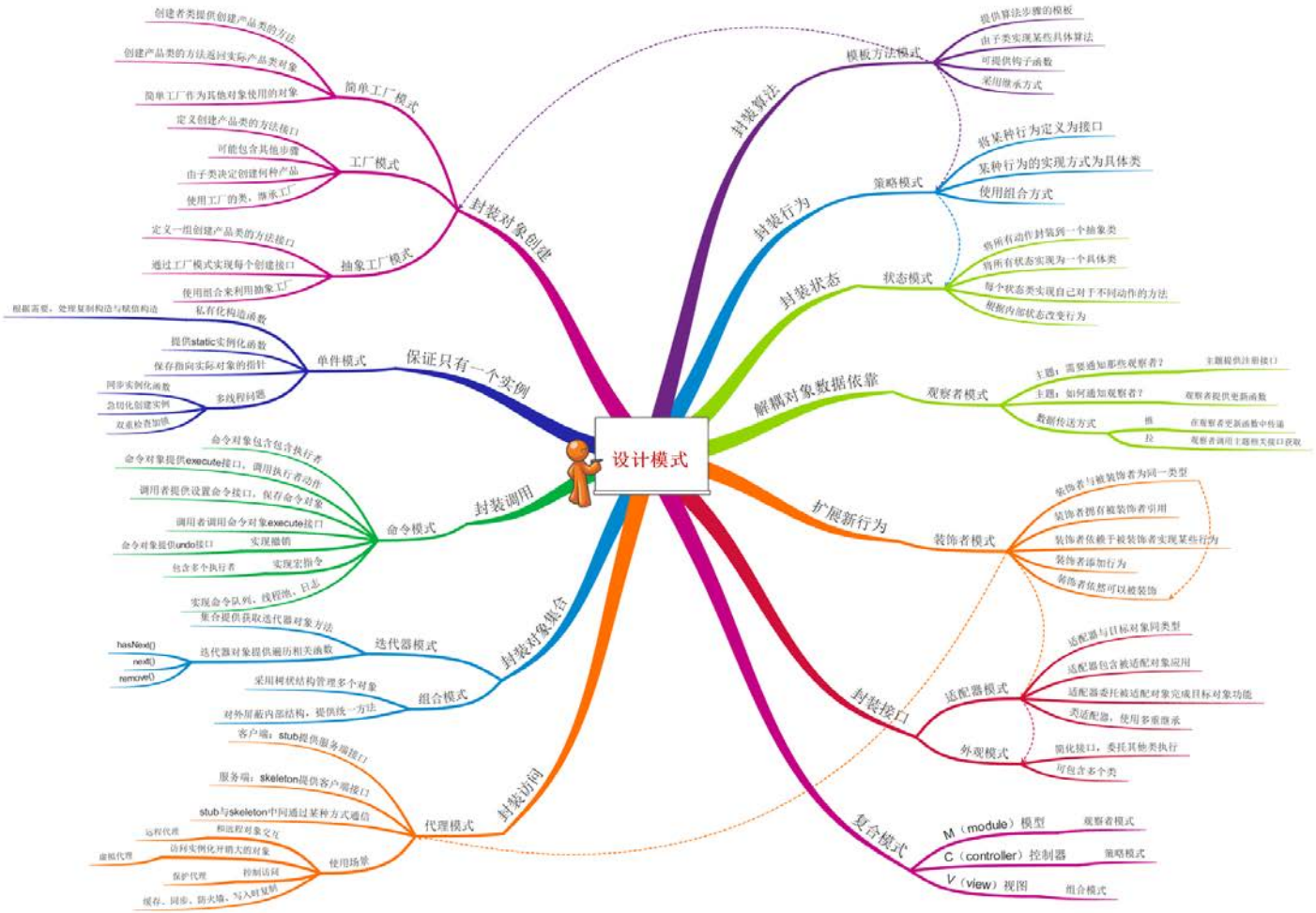


设计模式与框架

- **23种重要的设计模式**

设计模式与框架

- **23种重要的设计模式**

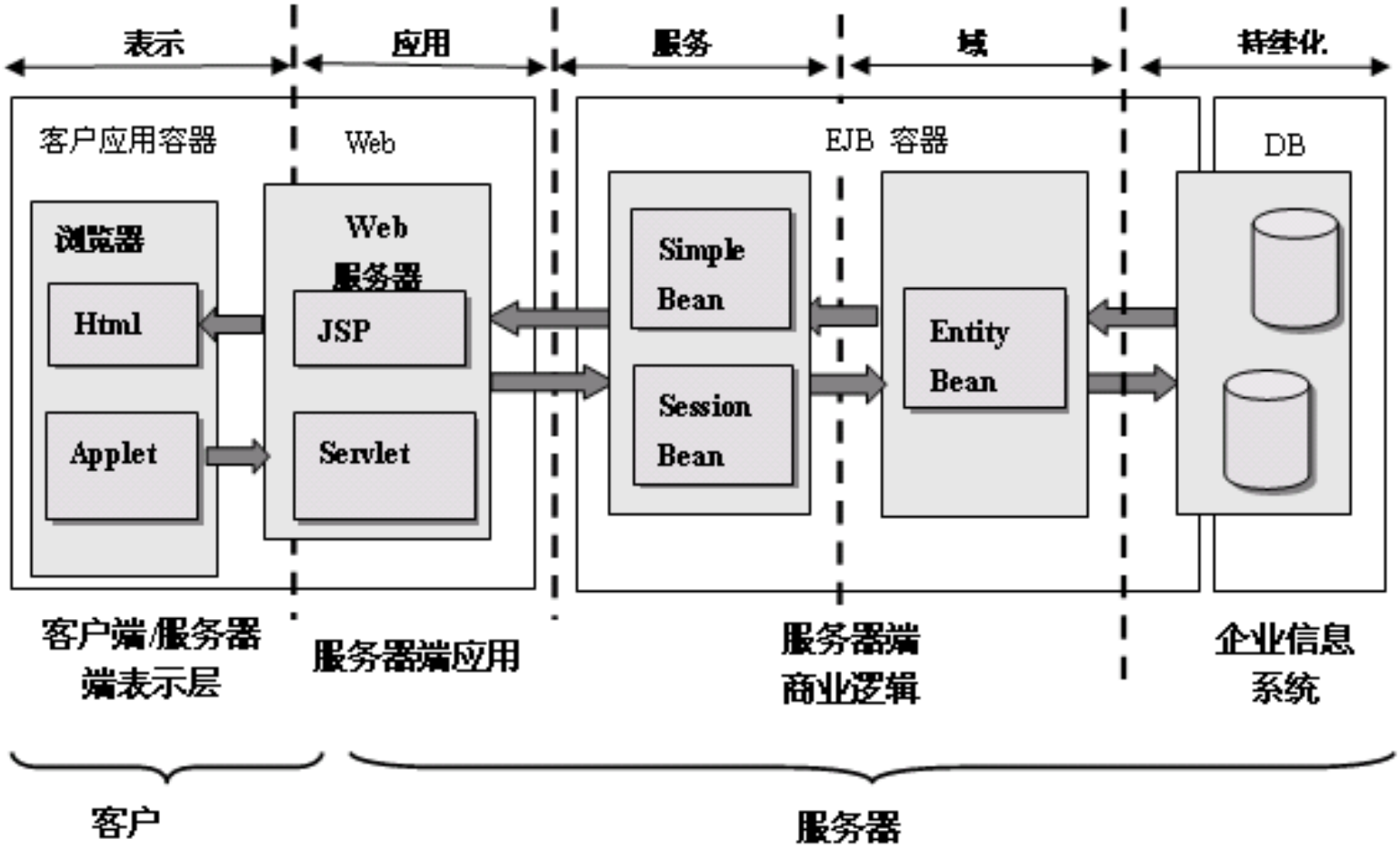


设计模式与框架

- **Java EE多层系统主要由架构设计、框架以及多个设计模式组成**
- **框架（Framework）与模式（Pattern）的关系：**
  - 设计模式比框架更抽象；
  - 设计模式是比框架更小的体系元素；
  - 框架比设计模式更加特例化；

设计模式与框架

- **多层架构设计：表示、应用、服务、域、连通性和持续化。**



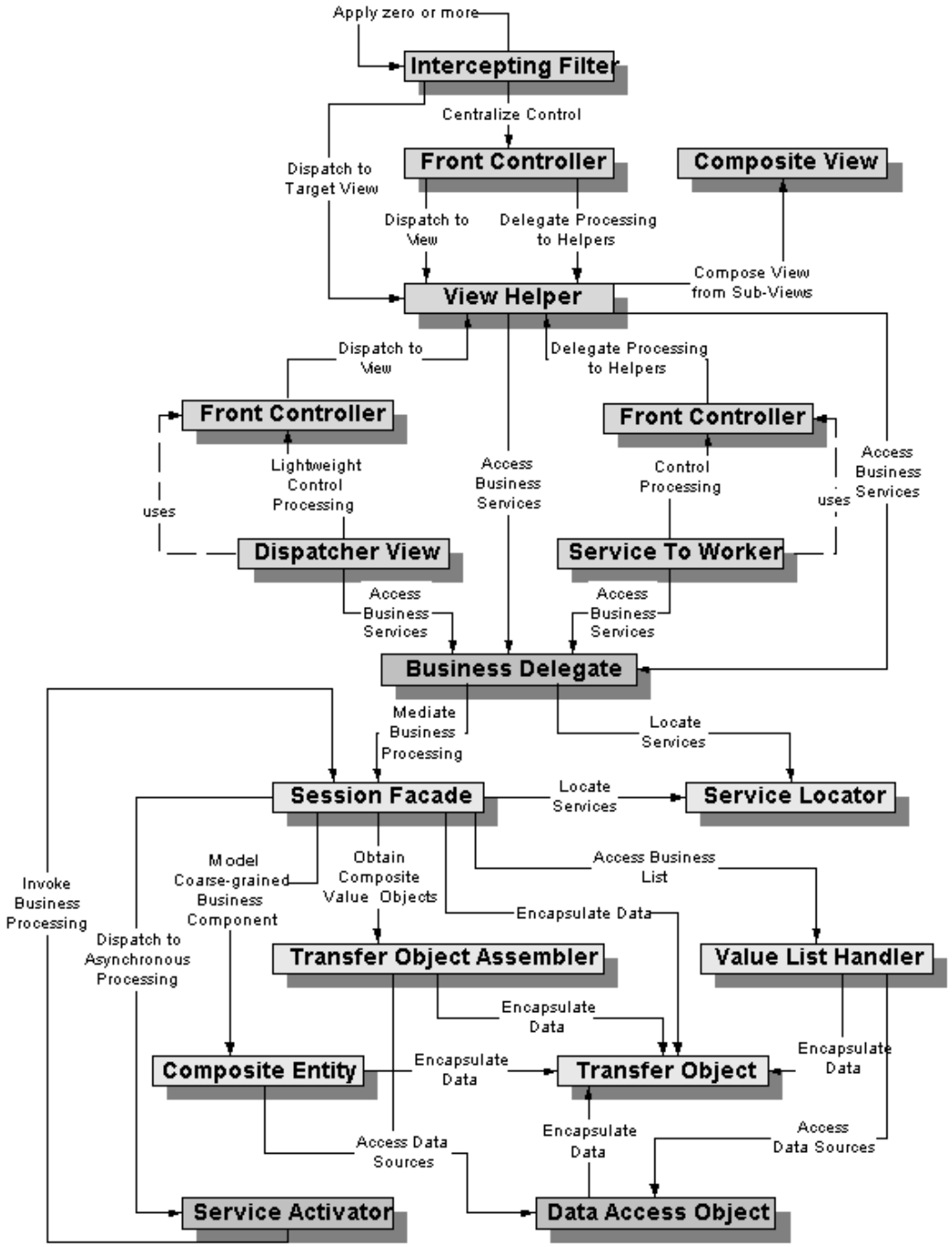
Java EE中的设计模式

- **经典的GOF模式**
  - **工厂模式**
    - **工厂模式的作用在于将创建具体类实例的方法由工厂类控制，客户程序只需要知道对象的类型。**
    - **如Spring中根据工厂配置生成Bean对象。**

```
ApplicationContext context = new
FileSystemXmlApplicationContext("applicationContext.xml");
Computer computer=(Computer)context.getBean("myComputer");
computer.doWork();
```

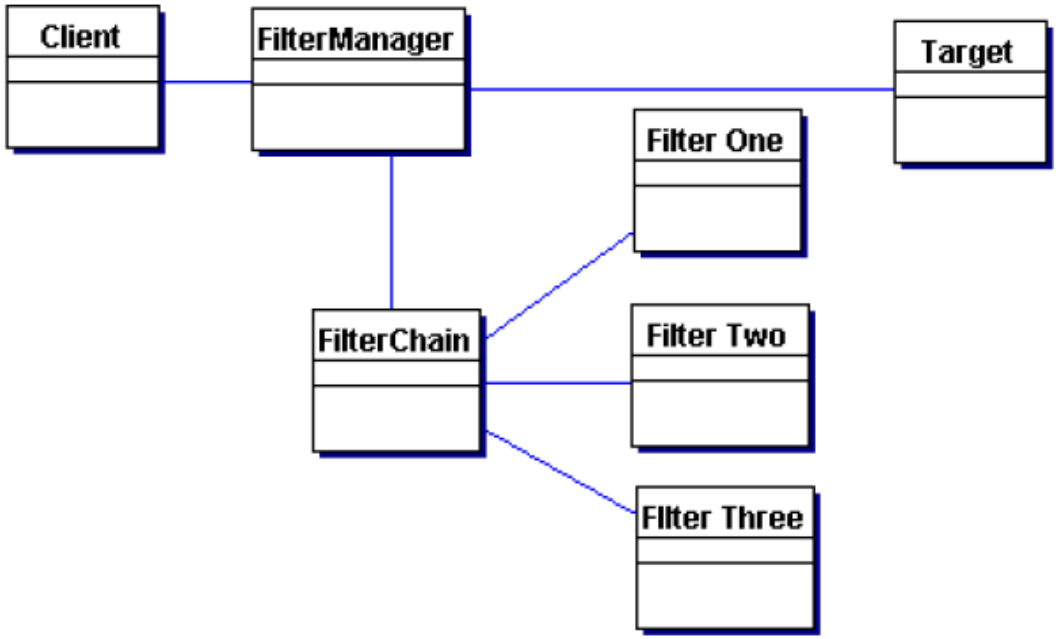
Java EE设计模式

- Java EE模式



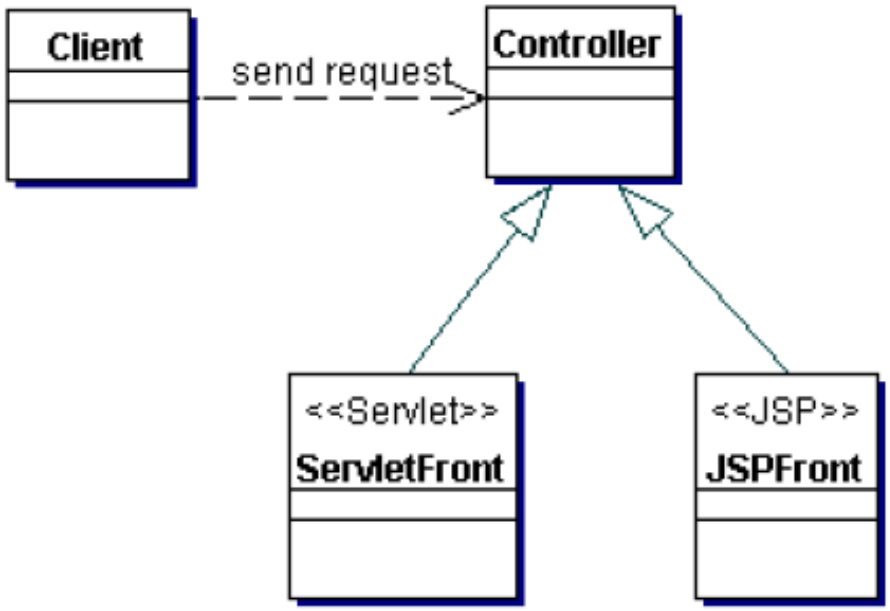
Java EE中的设计模式

- **Intercepting Filter—拦截过滤器模式**
  - 主要用于表现层
  - 采用灵活，非硬编码的方式，在请求到达处理器前，进行检查和处理
  - **用于验证，编码转换，客户端识别，日志处理，session识别等**
  - **如Servlet Filter**



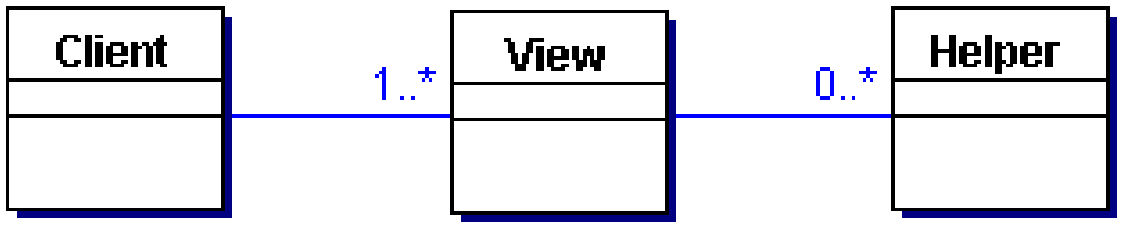
Java EE中的设计模式

- **Front Controller—前端控制器模式**
  - 主要用于表现层
  - 一个中心点，用来协调系统服务，内容检索，视图管理和浏览等
  - **如Spring MVC中的DispatcherServlet**



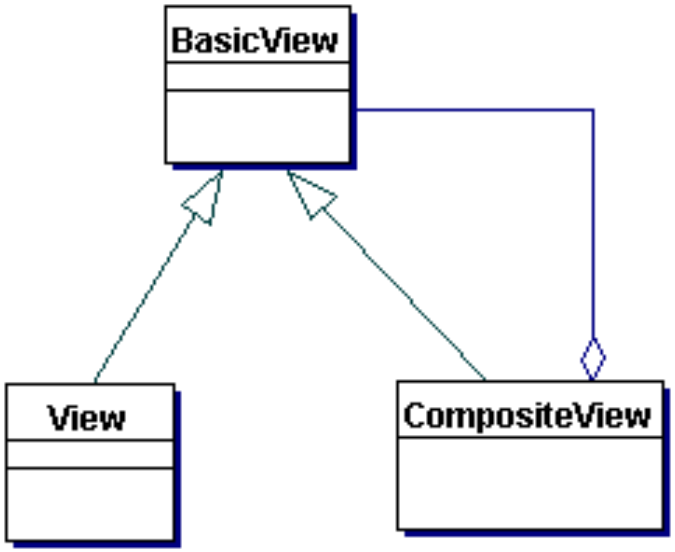
Java EE中的设计模式

- **View Helper—视图助手模式**
  - 主要用于表现层
  - 视图包含格式化的代码，将处理功能让它的helper类来实现，比如JavaBean或者定制化标签
  - **如JSP中的Scriptlet被组件化成JavaBean或者定制化标签**
  - **框架大量采用自定义标签，模块化功能，得到重用，避免错误**



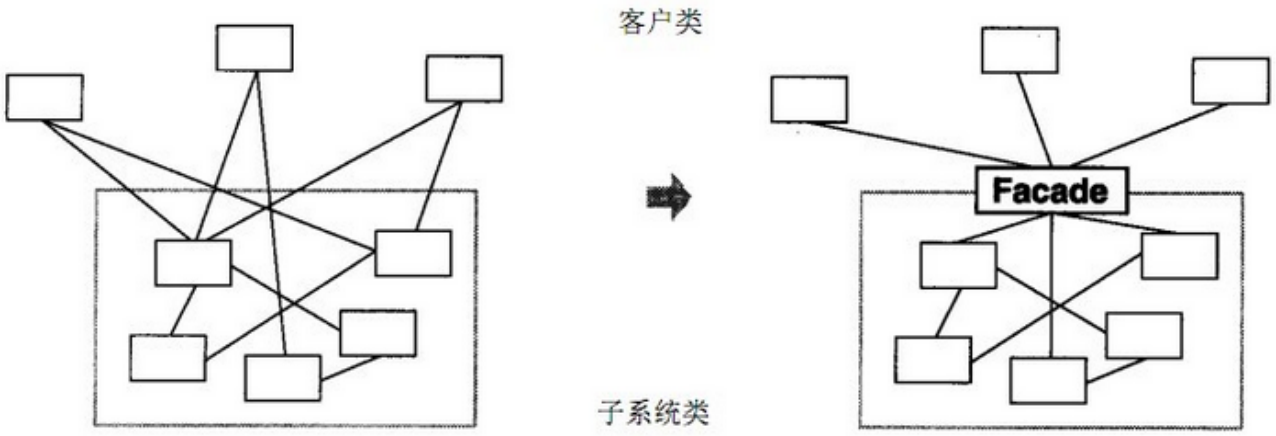
Java EE中的设计模式

- **Composite View—复合视图模式**
  - 主要用于表现层
  - 复杂视图由多个原子视图组成；每个模版组件可以动态加入，页面布局可以独立于内容的进行管理
  - **如portal和portlet;**



Java EE中的设计模式

- **Session Facade—会话门面模式**
  - 如会话Bean和实体Bean的调用关系
  - 一个会话Bean中调用多个实体Bean
    - 该会话Bean是一个Façade类/Manager类
  - 使用Façade 会话Bean优点：
    - 提高性能，节省客户端直接调用实体Bean的网络开销
    - 解耦分层，利于扩展变化

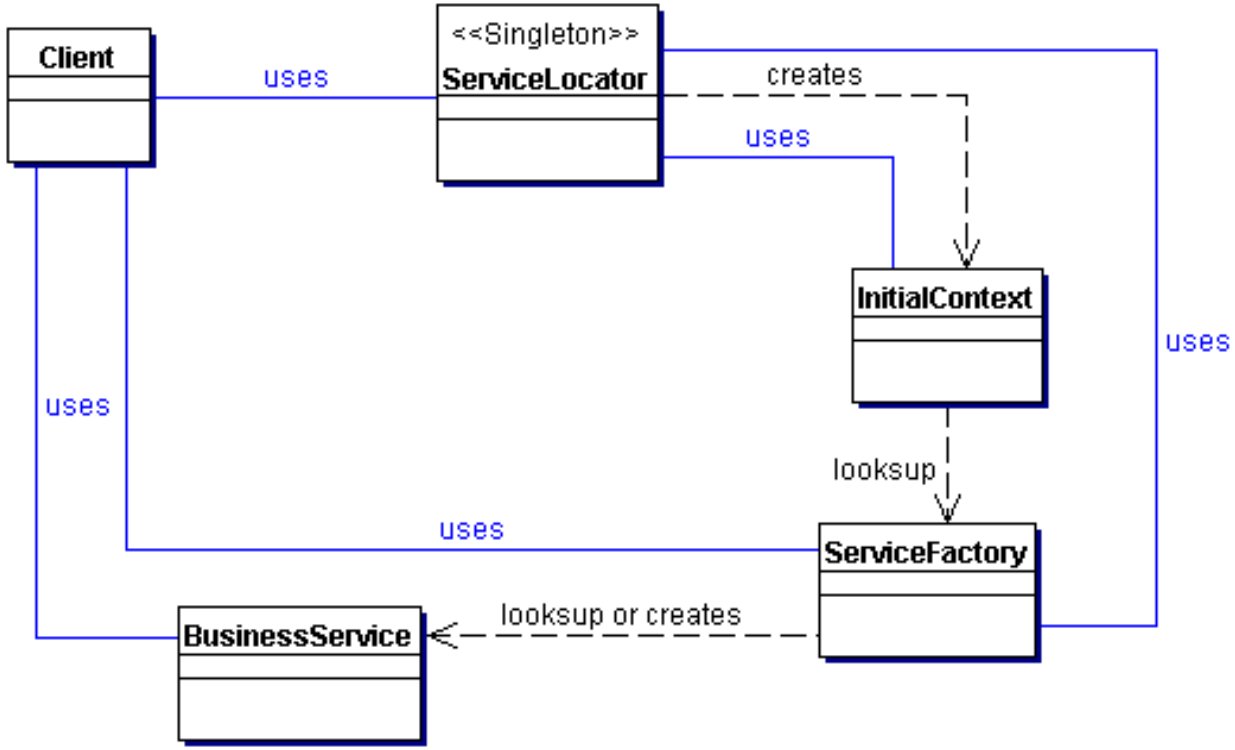


Java EE中的设计模式

- **Transfer Object—传输对象模式**
  - **TO: 在应用程序不同层之间传输的对象。**
    - DTO（Data Transfer Object）模式：一般用于表现（Web）层和应用（Service）层间的数据传输。当和UI密切相关时，也可以认为是个VO。数据传输对象是根据UI的需求进行设计的，而不是根据领域对象进行设计的。
  - 区分下面的概念：
    - VO值对象(Value Object) 是指将数据封装成普通的POJO；如struts1 里的ActionForm 。
    - PO（persistence object）在O/R映射的时候出现的概念，对应数据模型(数据库)，简化对象的数据转换为物理数据的编程，一般用于Service层--Dao层间的数据传输。实体bean就是个PO。
    - BO（business object:业务对象）把业务逻辑封装为一个对象，可能包含多个PO

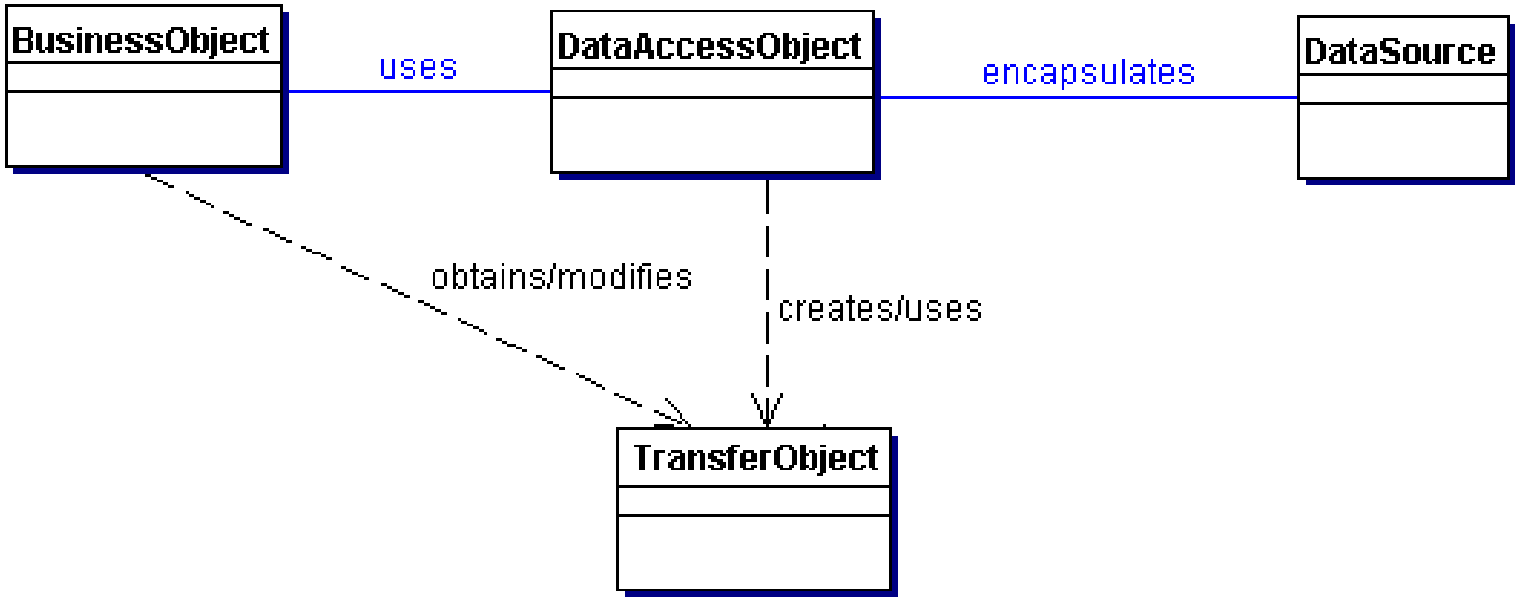
Java EE中的设计模式

- **Service Locator—服务定位器模式**
  - 使用一个服务定位器对象来抽象具体服务对象的查找和生成方法；多个客户端可重用改服务器对象。提供一个单点控制；使用Cache提高性能
  - 比如，通过JNDI获取JMS的受管对象JMS Connection。



Java EE中的设计模式

- **DAO（Data Access Object）—数据访问对象模式**
  - DAO中包含了各种数据库的操作方法，负责持久层的操作，用于访问数据库。
  - 通常和PO结合使用，为业务层提供接口。通过它的方法，结合PO对数据库进行相关的操作。



Web应用程序框架

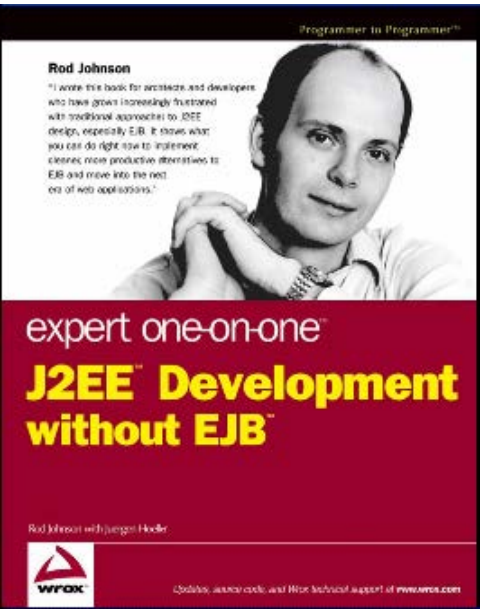
- Spring MVC（Spring Boot）
- Spring
- MyBatis
- Hibernate
- Echo
- Tapestry
- WebWorks
- Apache Struts (struts2)
- JavaServer Faces (JSR-127)

Spring概述

Spring Overview

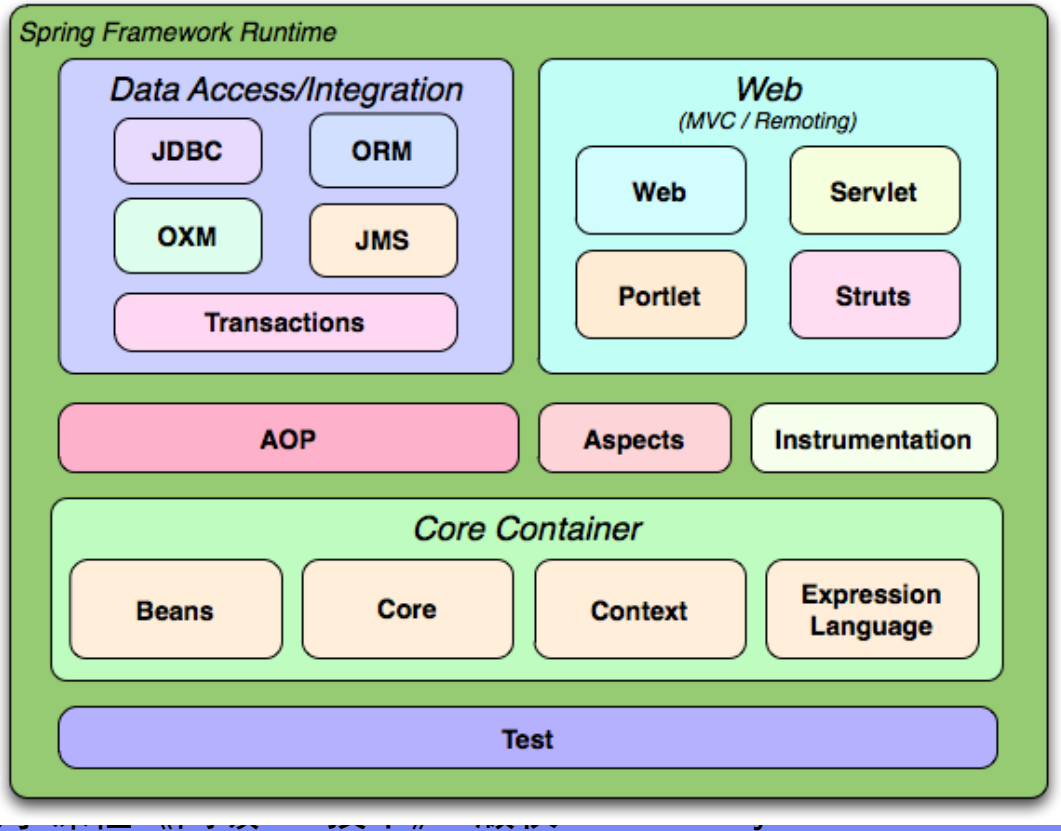
- **Spring是Rod主创的一个应用于Java EE领域的轻量应用程序框架，其核心是一个IOC容器以及AOP实现**

Rod Johnson **《Expert One-on-One Java EE Design and Development》**



Spring框架模块

- 核心容器：Core，Beans，Context，Expression Language
- 数据访问和整合模块：JDBC，ORM，OXM，JMS，Transaction
- Web模块：Web，Web-Servlet，Web-Struts，Web-Portlet
- AOP模块
- 测试模块

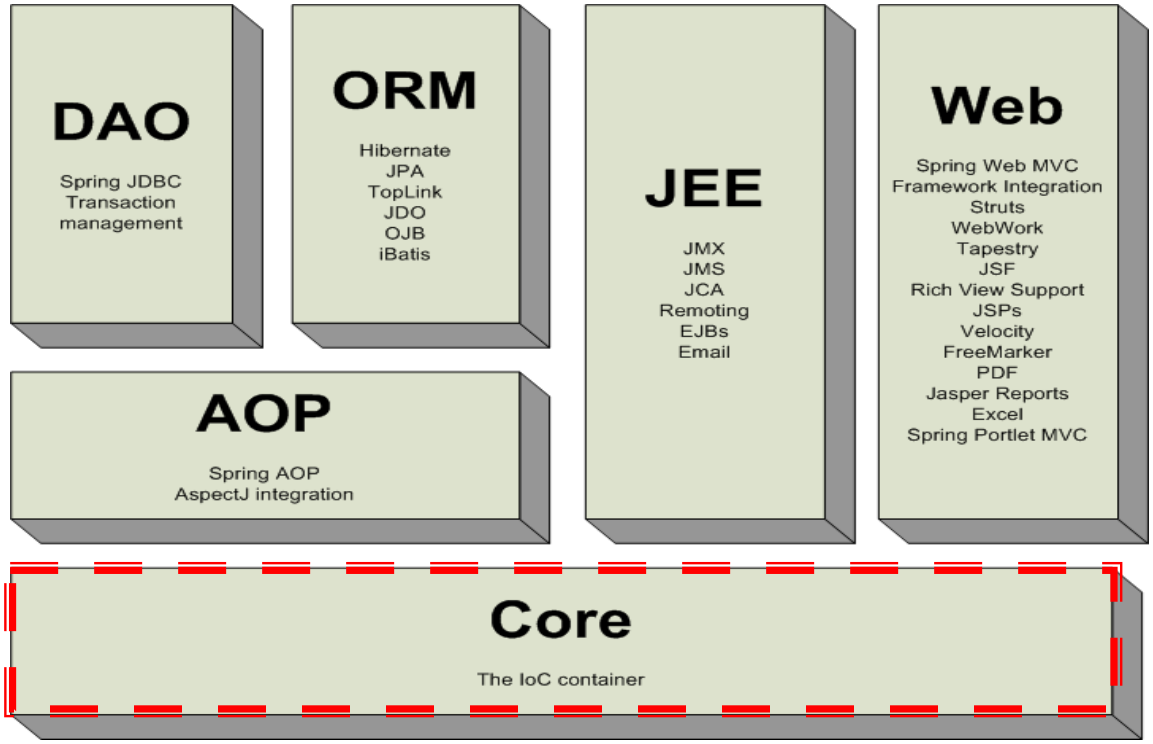


Spring框架的特点

- 轻量级框架
- IOC（控制反转）容器
- AOP（面向切面编程）的支持
- 声明式的事务支持
- 封装和简化了Java EE的很多API
- 对各种框架的整合

Spring Core

- **BeanFactory是spring IOC的核心组件，是工厂模式的一种实现，将实际代码和依赖配置进行了分离。**



控制反转IoC

- **IoC (Inversion of control)**
  - 也称为依赖注入DI (Dependency Injection)
  - 好莱坞原则: “ Don‘t call me, I’ll call you. ”
  - **控制: 对象的生命周期和对象间的关系**

- **如何实现目标功能的松耦合：替换实现功能模块？**
  - 将所有可能出现的实现模块都准备好，使用不同的调用方法。
  - 不改变调用方的调用方式，面向接口编程，并且用到工厂模式；分解依赖，按照需要自动“注入”相关对象
  - **一般通过XML配置文件来决定注入方式。**

控制反转IoC

- 传统做法

  不使用容器

```
public class Greeting{

        public void greet( )
        {
                Speaker s = new Speaker();
                s.sayHello();
        }
}
```

控制反转IoC

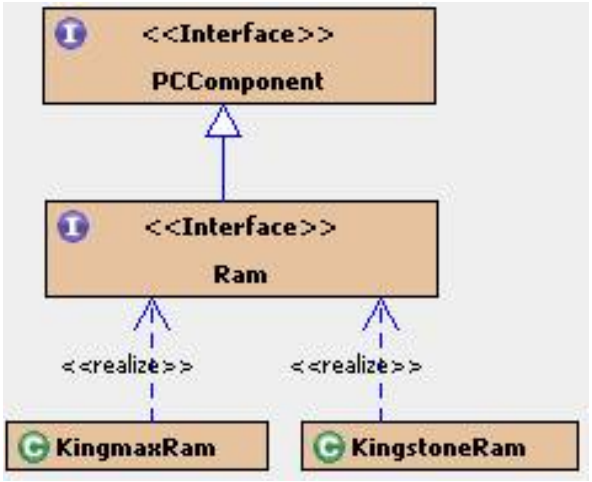
- 传统做法

  注册容器中的受管对象，通过JNDI访问

```
class XxxDAO {
    private DataSource ds;
    private static final String dataSourceJNDI="jdbcDataSource";
    private Connection getConnection() throws Exception {
        if (ds == null) {
            InitialContext context = new InitialContext();
            ds = (DataSource)
            PortableRemoteObject.narrow(context.lookup(dataSourceJ
            NDI), DataSource.class);
        }
        return ds.getConnection();
    }
    public void add(Object xxx){
        // ...
    }
}
```

控制反转IoC

- IoC (Inversion of control)
  - 面向接口编程

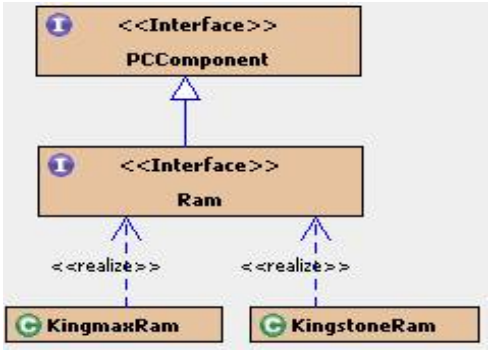



```
public interface PCComponent {
        String getName();// 名称
        double getPrice();// 价格
        String getCompany();// 厂家
}
```

```
public interface Ram extends PCComponent {

     int getSize();// 内存大小
     void inData();// 读数据
     void outData();// 取数据

}
```

控制反转IoC

- IoC (Inversion of control)
  - 面向接口编程

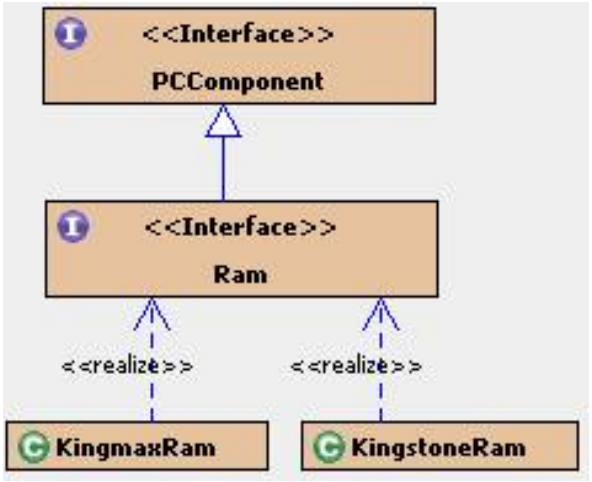



```
public class KingmaxRam implements Ram {

      public int getSize()  {   return 512; }
      public void inData() {    // 读入数据
       }
      public void outData() {  // 输出数据
       }
       public String getName() {return "Kingmax 内存 "; }
       public double getPrice() {return 300; }
        public String getCompany() {return "Kingmax 公司 "; }
 }
```

```
public class KingstoneRam  implements Ram {

      public int getSize()  {   return 512; }
      public void inData() {    // 读入数据
       }
      public void outData() {  // 输出数据
       }
       public String getName() {return "Kingstone  内存 "; }
       public double getPrice() {return 200; }
        public String getCompany() {return "Kingstone公司 "; }
 }
```

控制反转IoC

- IoC (Inversion of control)
  - 工厂模式



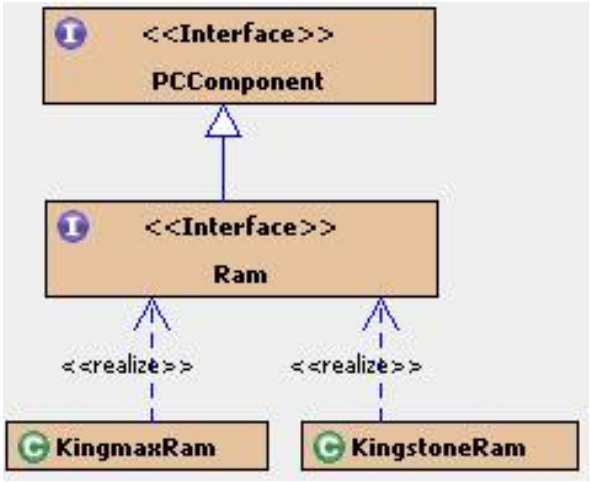
从外部文件中获取

```
public classBeanFactory{ //框架化
  public static Object getBean(StringaName){
     String className= "**";//根据读取的类名；
     return Class.forName(className).newInstance();
     }
 } //设置配置文件，定义别名和类名的映射
```

```
…
Ram ram= (KingstoneRam )BeanFactory.getBean("别名" );
ram. getCompany();
..
```

控制反转IoC

- **IoC (Inversion of control)**
  - **结合前两者**




```
<beans>
  <bean id = "myMainboard" class = "ioc.IntelBoard">
     <property name = "ram" >
          <bean class = "ioc.KingmaxRam"/>
     </property>
     <property name = "cpu" >
          <bean class = "ioc.AMDCpu"/>
    </property>
  </bean>
  <bean id = "myComputer" class = "ioc.Computer" >
     <property name = "mainboard" ref = "myMainboard"/>
  </bean>
</beans>
```

```
ApplicationContext context = new
FileSystemXmlApplicationContext("applicationContext.xml");
Computer computer=(Computer)context.getBean("myComputer");
computer.doWork();
```

控制反转IoC

- **IoC 的优势**
  - **将寻找和产生相关对象的责任交给了配置文件和读取配置文件的工厂类对象。**
  - **减少了实现之间的紧耦合，鼓励面向接口的设计。**
  - **允许应用在代码之外进行重新设置关系。**
  - **支持编写方便测试的独立组件。**

Bean 工厂

- 采用配置的机制管理Bean,如XmlBeanFactory
- 理论上可以采用各种配置文件格式
  - – XML
  - – 属性文件
  - – 数据库
  - – 目录LDAP
-

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<bean id="..." class="...">
...
</bean>
<bean id="..." class="...">
...
</bean>
...
</beans>
```

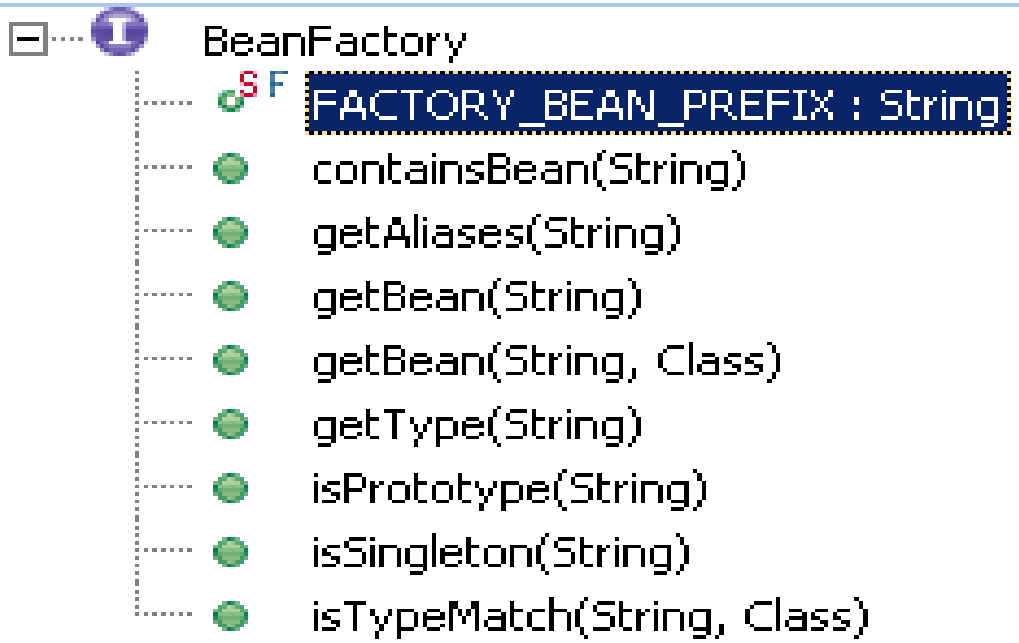
Application Context

- ApplicationContext 是BeanFactory的子接口，添加了许多有用特征：
  - – 支持文本信息的解析
  - – 装在文件资源的通用方法
  - – 发布事件

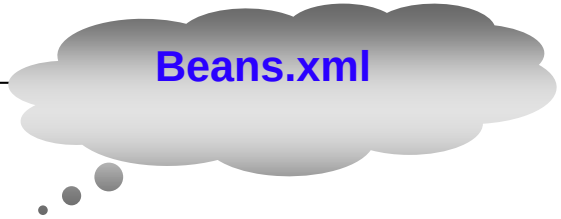
- WebApplicationContext是ApplicationContext的子接口
  - XmlWebApplicationContext是WebApplicationContext的实现，用于 web应用中

Bean Factory

- 位于org.springframework.beans 包下
  org.springframework.beans.factory 接口



XML Configuration File




```
<?xml version="1.0" ?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   <bean id="messageDisplay"
     class="test.SystemOutMessageDisplay"></bean>
   <bean id="messageProducer" class="test.HelloMessageProducer"></bean>
   <bean id="helloApp" class="test.HelloApp" >
      <property name="display">
        <ref local="messageDisplay" />
      </property>
      <property name="producer">
        <ref local="messageProducer" />
      </property>
   </bean>
</beans>
```

Bean Factory Example

```
public class Test {
  public static void main(String[] args) {
    BeanFactory factory = new XmlBeanFactory(
              new ClassPathResource("/Beans.xml"));
    HelloApp helloApp = (HelloApp)
     factory.getBean("helloApp");
    helloApp.displayProducedMessage();
  }
}
```


```
ApplicationContext context = new
FileSystemXmlApplicationContext("applicationContext.xml");
Greeting greeting = (Greeting) context.getBean("Greeting");
```

Spring

- **IOC容器**
  - **非入侵式框架**
    - **侵入式框架: 一组类与对象按照既定的规则交互。如果需要使用框架的功能，用户必须实现框架的接口。于是代码与框架耦合，可移植性降低。**

```
public class xxx implements EJBHome{
        create(){}
        .....
 }
```

Spring

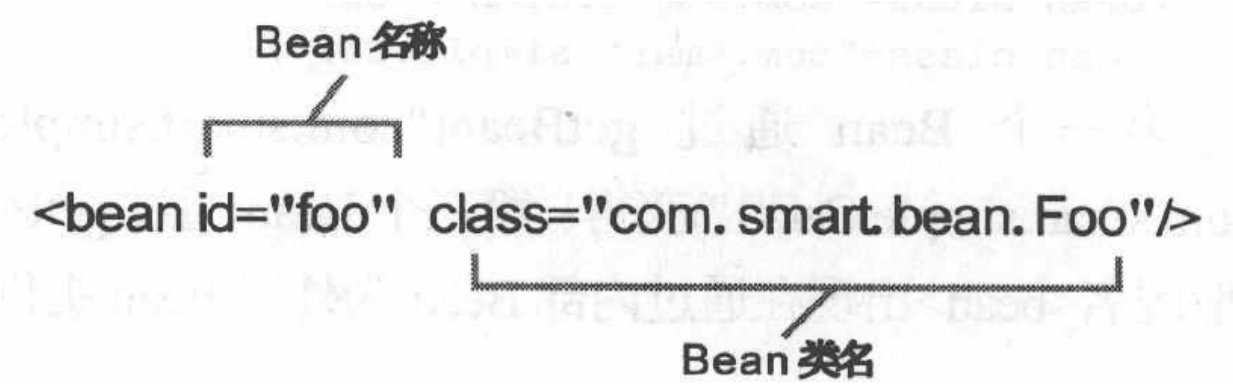
- **Dependent Injection: 依赖注入**
  - **接口注入：实现容器接口，由容器完成对象关系的建立。侵入式代码。**
  - **setter注入：setXXX()，spring推荐。**
  - **constructor注入。在实例化时完成注入。运行时难以改变。**

Demo SpringIOCDemo\Computer in DOS env

HelloSpring in Myeclipse env

Spring

- IOC容器
  - 容器：容纳对象，并维护各个对象之间的关系
    - 通过配置文件管理对象间关系
    - 创建由容器完成
    - “注入”（对象关系的建立）也由容器完成



Spring

- **Dependent Injection: 依赖注入**
  - **注入方式——setter注入：setXXX()，spring推荐。**

```
public class Car {
        private int maxSpeed;
        public String brand;
        private double price;
        public void setBrand(String brand) {
            this.brand = brand;
        }
        public void setMaxSpeed(int maxSpeed)
            this.maxSpeed = maxSpeed;
        }
        public void setPrice(double price) {
            this.price = price;
        }
       ...
}
```

```
<bean id="car" class="com.smart.ditype.Car">
    <property name="maxSpeed"><value>200</value></property>
    <property name="brand"><value>红旗 CA72</value></property>
    <property name="price"><value>20000.00</value></property>
</bean>
```

Spring

- **Dependent Injection: 依赖注入**
  - **注入方式—— constructor注入。在实例化时完成注入。运行时难以改变。**

```
public Car(String brand, double price) {
    this.brand = brand;
    this.price = price;
}
```

```
<bean id="car1" class="com.smart.ditype.Car">
  <constructor-arg type="java.lang.String"> ❶
    <value>红旗 CA72</value>
  </constructor-arg>
  <constructor-arg type="double"> ❷
    <value>20000</value>
  </constructor-arg>
</bean>
```

Spring

- **Dependent Injection: 依赖注入**
  - **对其他bean的引用。**

```
public class Boss {
    private Car car;
    //设置car属性
    public void setCar(Car car) {
        this.car = car;
    }
    ...
}
```

```
<bean id="car" class="com.smart.attr.Car"/>
<bean id="boss" class="com.smart.attr.Boss">
    <property name="car">
        <!--❷引用❶处定义的car Bean -->
        <ref bean="car"></ref>
    </property>
</bean>
```

Spring

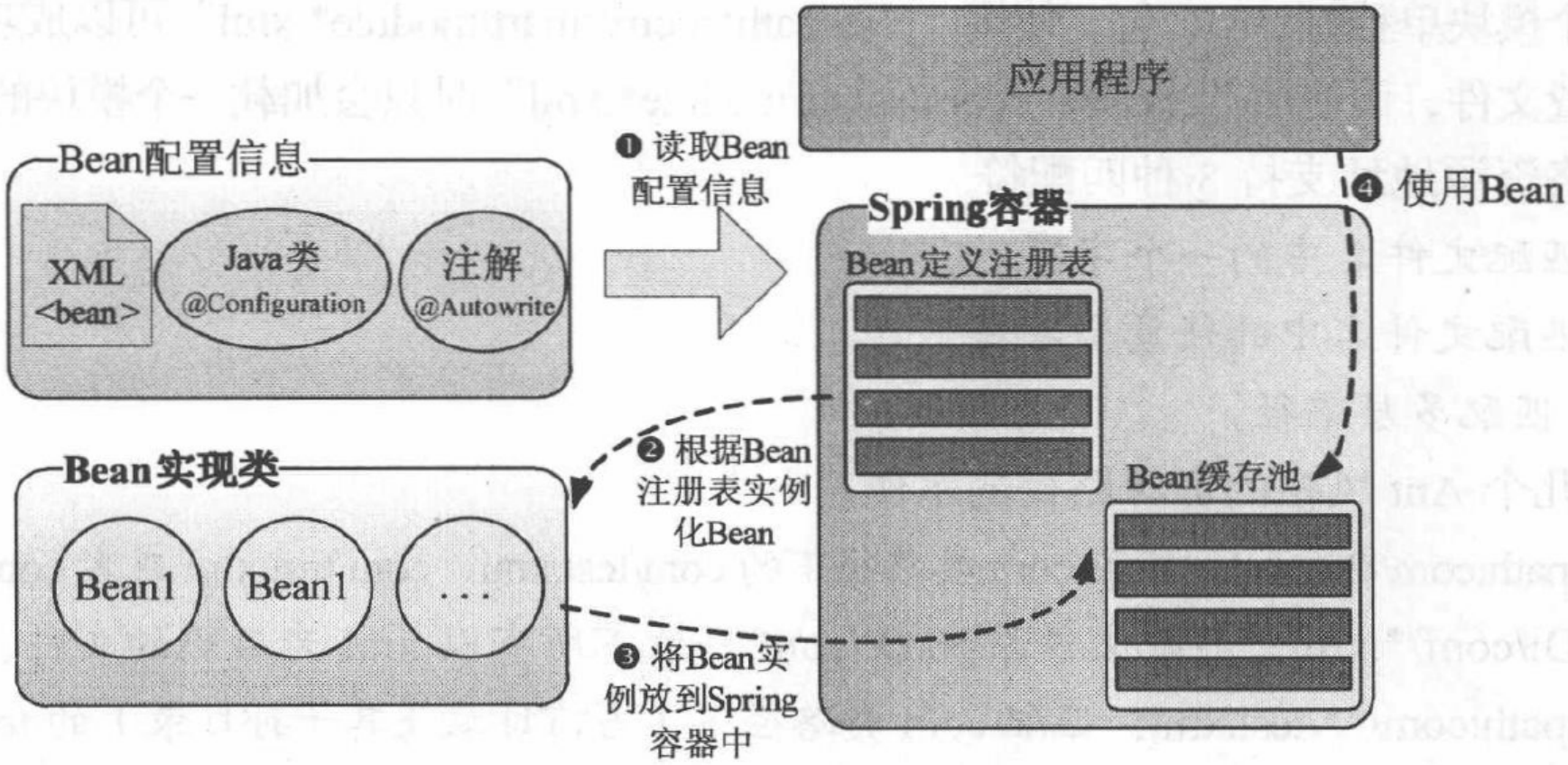
- 依赖注入
  - 采用注释的配置
    - @Component :定义一个通用组件，容器将POJO转为容器管理的bean
    - @Repository：Dao
    - @Service ： Service实现类
    - @Controller ： Controller实现类
    - @Autowired ： 自动装配

```
@Service                                    ❶定义一个 Service
public class LogonService {
    @Autowired                              ❷分别注入 LogDao 及
    private LogDao logDao;                    UserDao 的 Bean
    @Autowired
    private UserDao userDao;
    ...
}
```

```
<!--❷扫描类包以应用注解定义的 Bean-->
<context:component-scan base-package="com.smart.anno"/>
```

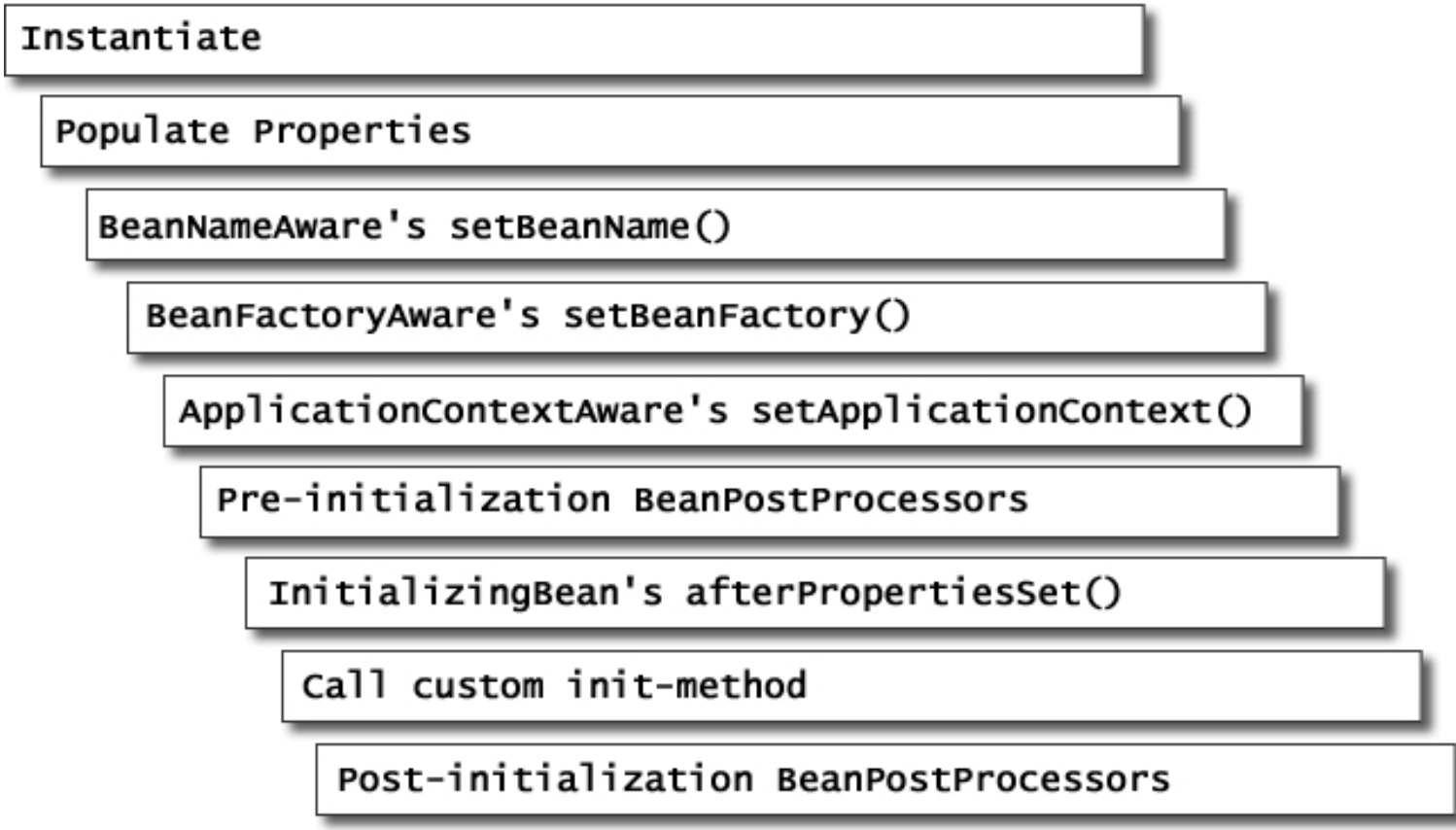
Spring

- **Spring容器高层视图**

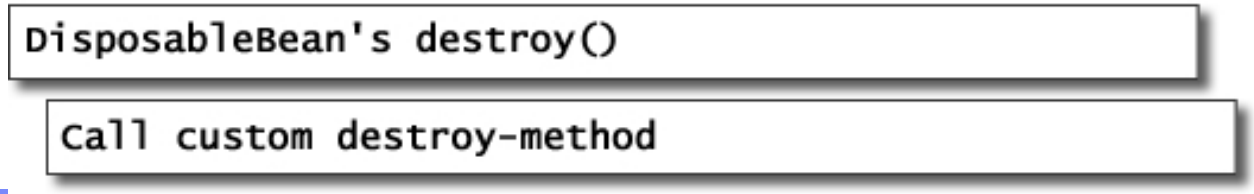


Bean 生命周期

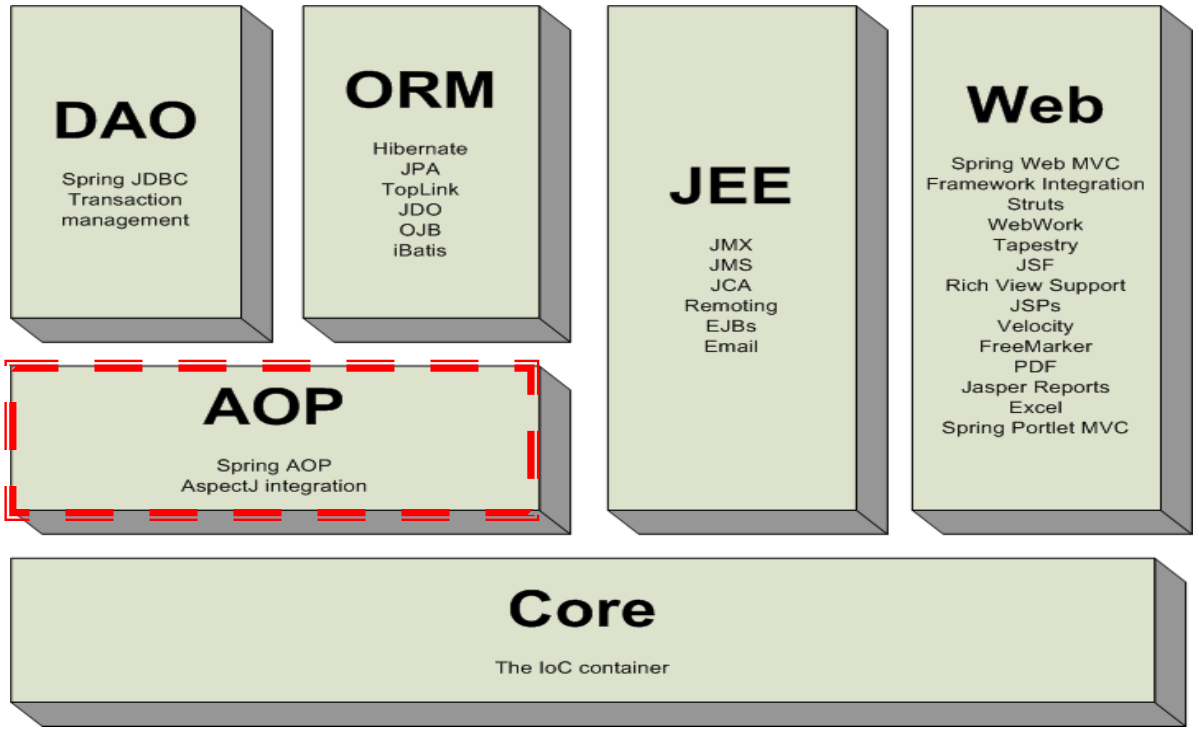
- 启动



- 关闭



Spring AOP



AOP 概念

跨越多个模块的关注点称为横切关注点或交叉关注点(Crosscutting Concerns)

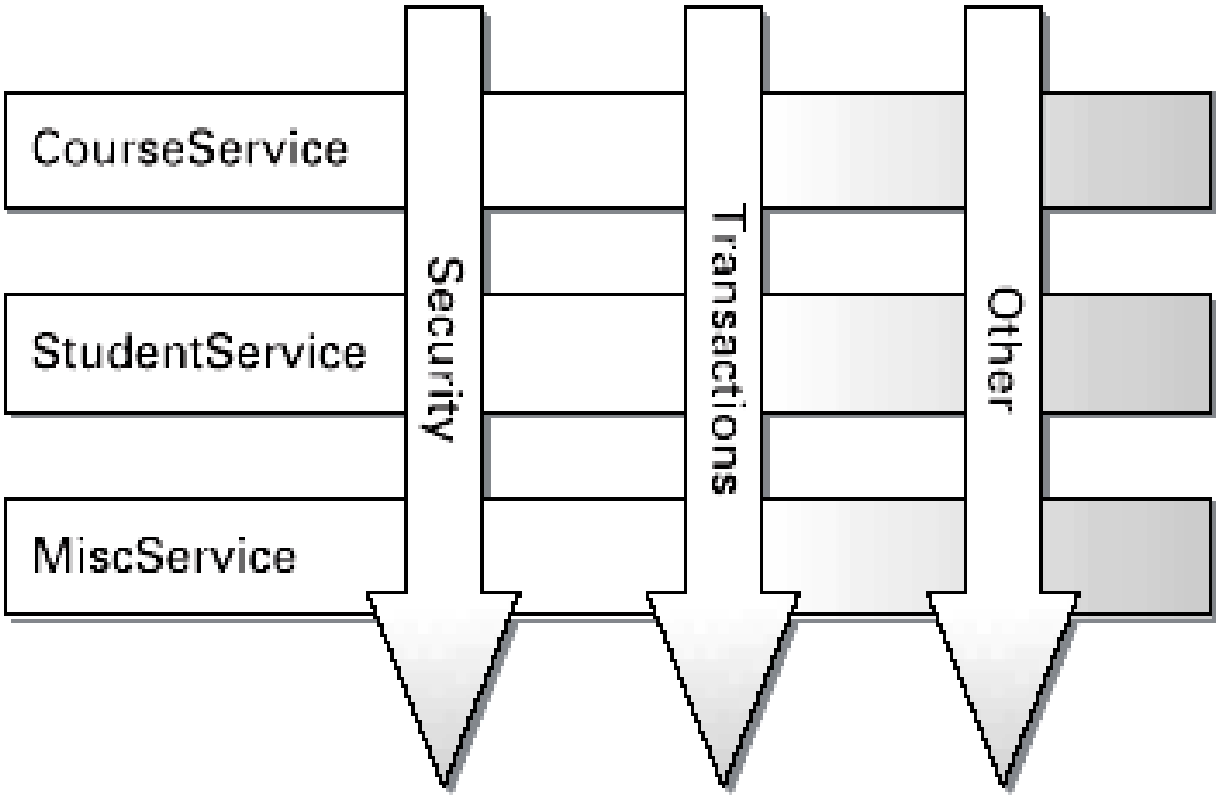


Figure 3.1
Cross-cutting concerns

Spring AOP

- **Aspect(切面)**
  - **解决跨越多个模块的交叉关注点问题(大多数是一些系统级的或者核心关注点外围的问题)的模块。**

    Log, Exception, Security, Transaction

- **Joinpoint （连接点,Spring AOP中当前只有方法调用)**

  Before Method, After Method Return, Exception Thrown

- **Advice (通知,定义了切面中的实际逻辑（即实现))**

  A piece of code

- **Pointcut (切入点,一个或多个joinpoint, 切入点的描述比较具体，而且一般会跟连接点上下文环境结合)**

  Joinpoint where you insert Advice

Spring AOP

- **Introduction(引入)**
  Add attributes or methods to class
- **Target: (目标对象)**
  The object being advices
- **Proxy (代理)**
  - **代理是将通知（Advice）应用到目标对象后创建的对象。**
- **Weaving(织入)**
  The process of create Proxy
  - **运行时织入：大多数AOP采用，代理方式**
  - **编译器织入:使用专门的编译器来编译基于语言的扩展，AspectJ**
- **interceptor(拦截器)**
  - 实现对连接点进行拦截，从而在连接点前或后加入自定义的切面模块功能，**Spring AOP采用**

Spring AOP 示例 – 业务接口

```
package aop;


public interface SayHello {
 public void sayHello(String name);
}
```

Spring AOP 示例– 实现

```
package aop;

  public class SayHelloImpl implements SayHello {

  public void sayHello(String name) {
   System.out.println("Hello, " + name);
  }

}
```

Spring AOP 示例 - Advice

```
package aop;
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class MyAdvice implements MethodBeforeAdvice
 {
  public void before(Method method, Object[] args, Object
   target)
   throws Throwable {
   System.out.println(method.getName() + ":");
  }
}
```

Spring AOP 示例 – beans.xml

```
<?xml version="1.0" ?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
   <beans>
   <bean id="sayHello" class="aop.SayHelloImpl"></bean>
   <bean id="myAdvice" class="aop.MyAdvice"></bean>
   <bean id="mySayHello"
     class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="proxyInterfaces">
        <list><value>aop.SayHello</value></list>
      </property>
      <property name="target"><ref local="sayHello" /></property>
      <property name="interceptorNames">
        <list><value>myAdvice</value></list>
      </property>
   </bean>
</beans>
```

Spring AOP 示例– 调用 Adviced 方法

```
package aop;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class Test {
  public static void main(String[] args) {
   BeanFactory factory = new XmlBeanFactory(new
     ClassPathResource(
   "aop/beans.xml"));
   SayHello hello = (SayHello) factory.getBean("mySayHello");
   hello.sayHello("Leo");
  }
}
```

Spring AOP 示例- 输出结果

```
sayHello:
Hello, Leo
```

Demo SpringAOPDemo inDOS env

Advice类型

- Before advice：
  - org.springframework.aop.MethodBeforeAdvice
- After advice
  - org.springframework.aop.AfterReturningAdvice

returning, throwing, finally

- Around advice
  - org.aopalliance.intercept.MethodInterceptor
  - 有返回值
- Introduction
  - 动态加入方法
  - Spring AOP 提供 IntroductionInterceptor接口
  - 改变类的定义

Advice类型-- Spring

- **Before Advice**

```
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

publicclass BeforeAdvice implements MethodBeforeAdvice
{
        //实现MethodBeforeAdvice的before方法
        public void before(Method method, Object[] args, Object target)
        {
                System.out.println("befor advice");
        }//end before

}//end class BeforeAdvice
```

Advice类型-- Spring

- **After returning advice**

```
import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;

publicclass AfterReturningAdviceImp implements AfterReturningAdvice
{
        public void afterReturning(Object returnValue, Method method,
                        Object[] arg2, Object target)  throws Throwable
        {
                System.out.println("afterReturning“);
        }
}
```

Advice类型-- Spring

- **Around advice**

```
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

publicclass AroundAdvice implements MethodInterceptor
{
        public Object invoke(MethodInvocation mi)
        {
                Object obj = null;
                //do something....
                return obj;
        }//end invoke(...)

}//end class AroundAdvice()
```

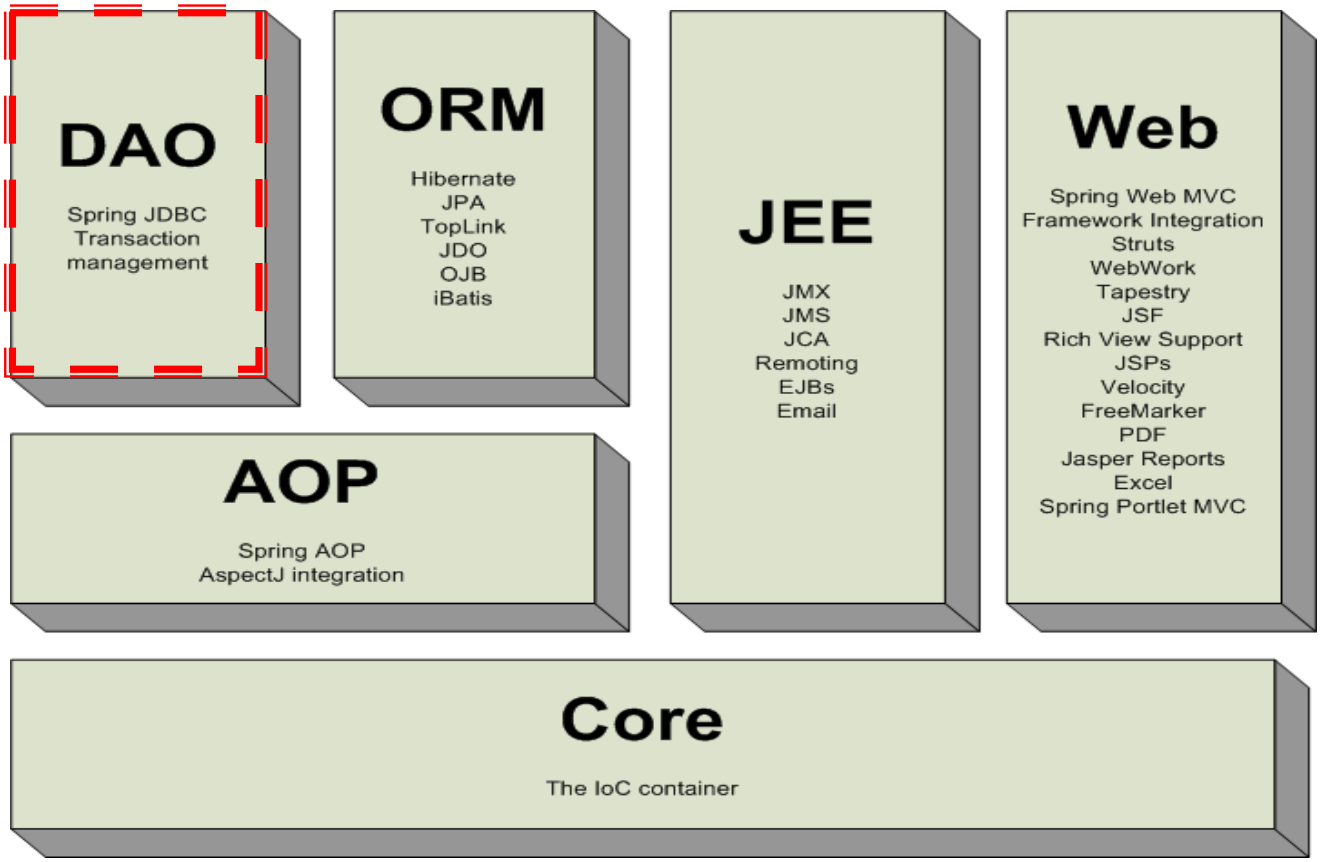
Demo Spring2AOPAround in Myeclipse env

Advice类型-- Spring

- **After throwing advice**

```
import java.lang.reflect.Method;
import org.springframework.aop.ThrowsAdvice;

publicclass AfterThorwsAdvice implements ThrowsAdvice
{
	public void afterThorwing(Throwable throwable)
	{
		//do something....
	}

	public void afterThrowing(Method method, Object[] args, Object target)
	{
		//do something....
	}
}
```

Spring DAO


DAO
Spring JDBC
Transaction
management
ORM
Hibernate
JPA
TopLink
JDO
OJB
iBatis
JEE
JMX
JMS
JCA
Remoting
EJBs
Email
Web
Spring Web MVC
Framework Integration
Struts
WebWork
Tapestry
JSF
Rich View Support
JSPs
Velocity
FreeMarker
PDF
Jasper Reports
Excel
Spring Portlet MVC
AOP
Spring AOP
AspectJ integration
Core
The IoC container

Sping DAO

- **JDBC模板**

  将繁杂和易于出错的异常处理置于框架功能内

- 异常层次

  易于理解的异常定义，不用考虑特定的 SQL表达以及错误处理代码

- 事务管理

  可编程的

  声明式的

JdbcTemplate

- 在一个service 实现类中通过传递一个DataSource 引用来完成JdbcTemplate 的实例化

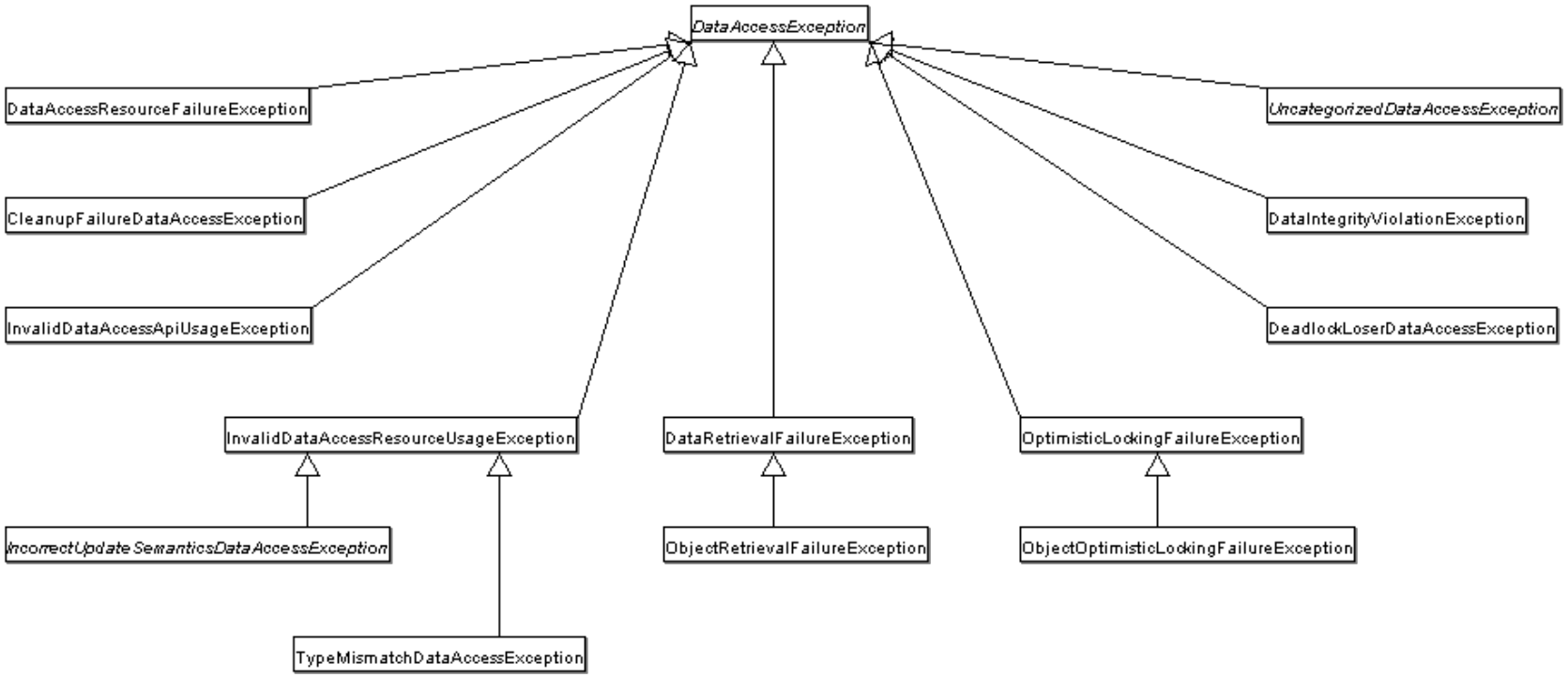
```
JdbcTemplate template = new JdbcTemplate(dataSource);
List names =template.query("SELECT USER.NAME FROM USER",
  new RowMapper() {
     public Object mapRow(ResultSet
     throws SQLException {
        return rs.getString(1);
     }
  }
);
```

- 在application context 中配置一个JdbcTemplate bean

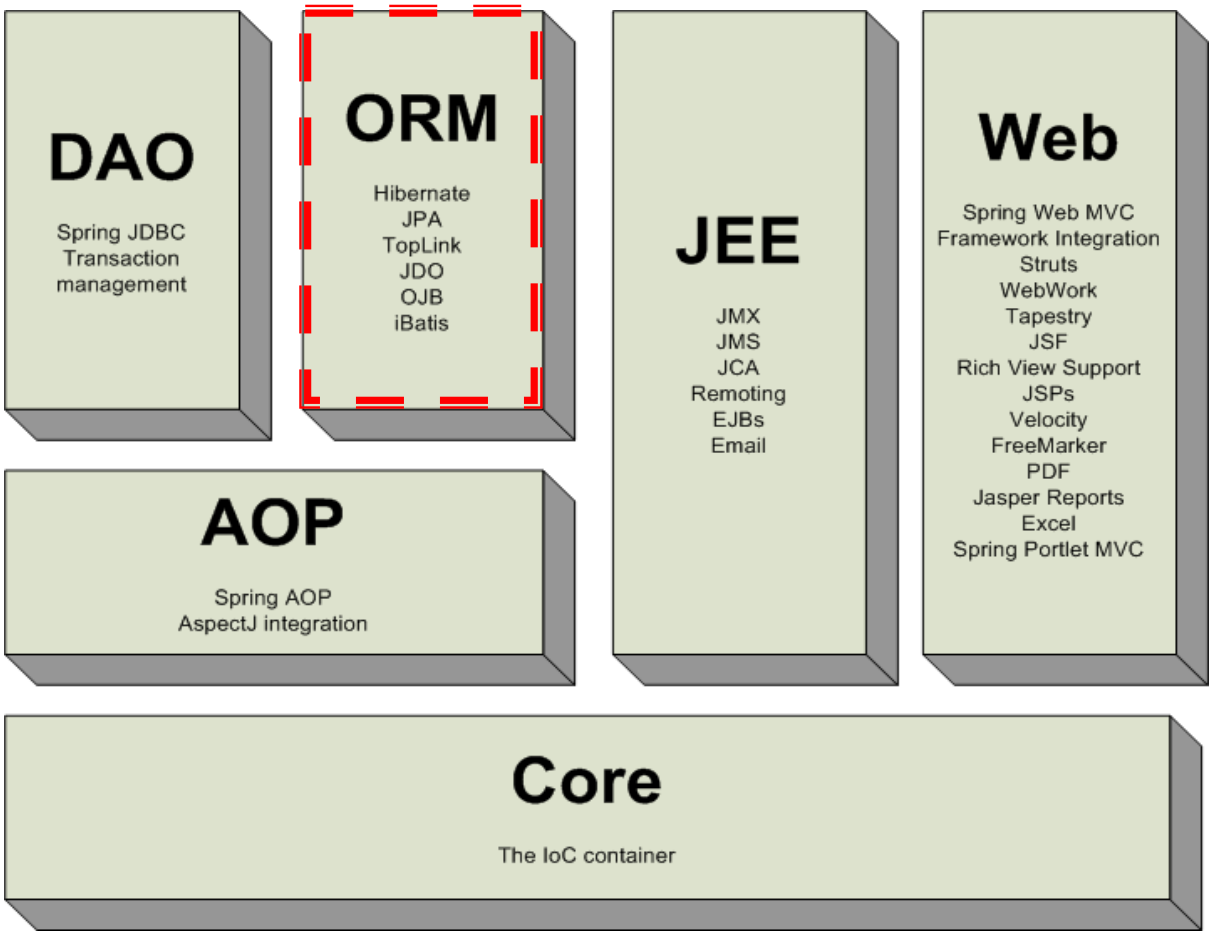
Demo SpringJDBC in Myeclipse

DataAccessException

- 与特定技术无关
- 是无需检测的异常
- **Spring的Dao框架具有一套丰富的分级异常体系**



Spring ORM



Spring ORM 集成

ORM 提供:

- 延迟加载
- 预先获取
- 缓存
- 层叠

Spring 集成了以下ORM:

- **Hibernate**
- **JDO**
- **TopLink**
- **iBATIS,MyBatis等**

Demo springhibernate in Myeclipse

Spring 与 Hibernate集成

- 管理Hibernate资源
  - 配置SessionFactory

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml">
        </property>
</bean>
```

Spring 与Hibernate集成

- 用HibernateTemplate访问Hibernate
  - HibernateTemplate包含大量简便的方法操作数据
  - Dao类继承HibernateDaoSupport（提供了getHibernateTemplate()获得HibernateTemplate）

```
public class StudentDAO extends HibernateDaoSupport {…
    public void save(Student transientInstance) {
            try {   getHibernateTemplate().save(transientInstance); }
          catch RuntimeException re) {throw re;}  }…
}
```

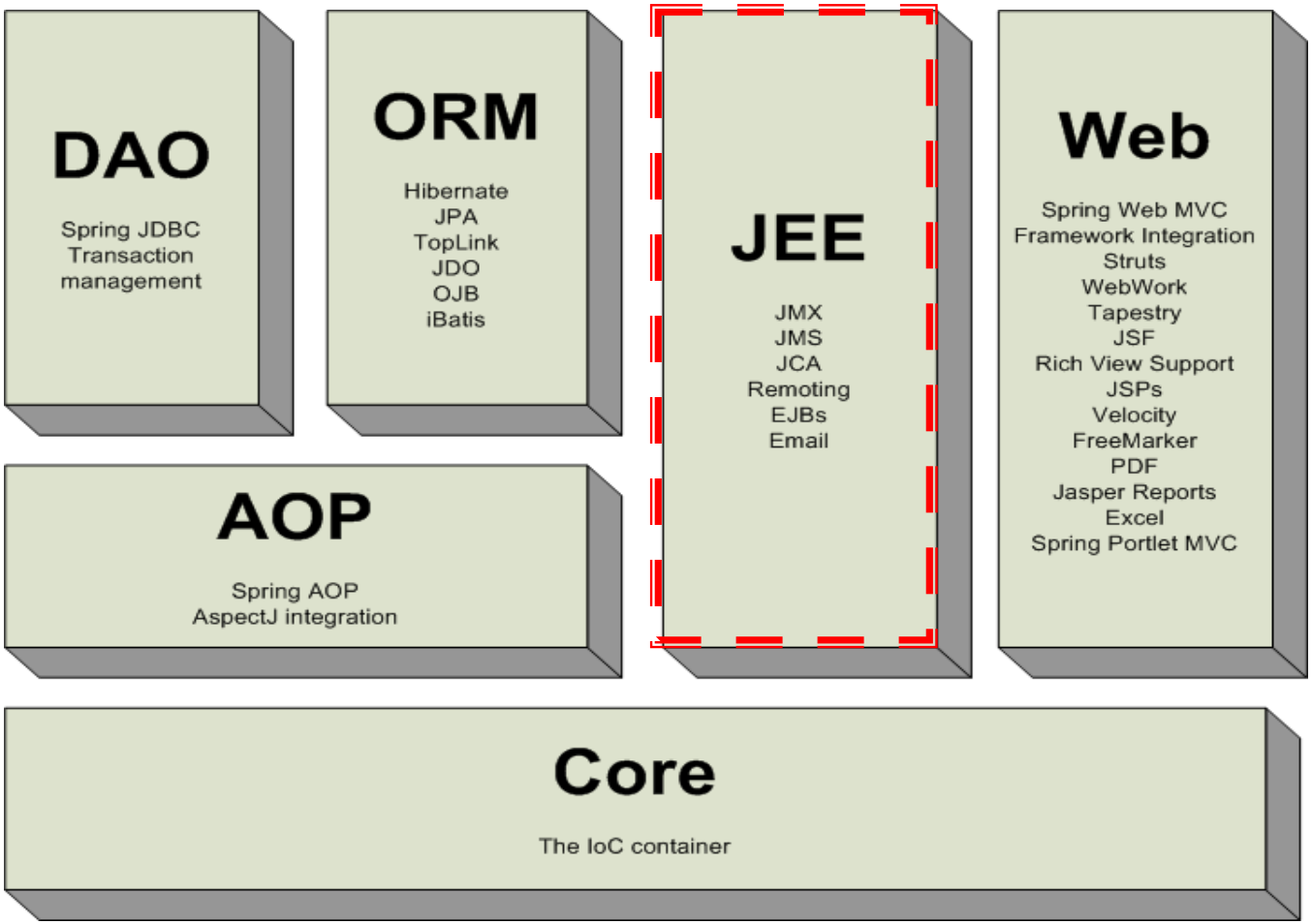
  - 织入SessionFactory(HibernateDaoSupport包含该属性)

```
<bean id="StudentDAO" class="dao.StudentDAO">
      <property name="sessionFactory">
            <ref bean="sessionFactory" />
      </property>
</bean>
```

Spring的ORM的优势

- Session management：线程安全，轻量级模板类
- Resource management：资源管理
- Integrated transaction management：整合事务管理
- Exception wrapping：异常处理
- 便利的支持类

JEE


DAO
Spring JDBC
Transaction
management
ORM
Hibernate
JPA
TopLink
JDO
OJB
iBatis
JEE
JMX
JMS
JCA
Remoting
EJBs
Email
Web
Spring Web MVC
Framework Integration
Struts
WebWork
Tapestry
JSF
Rich View Support
JSPs
Velocity
FreeMarker
PDF
Jasper Reports
Excel
Spring Portlet MVC
AOP
Spring AOP
AspectJ integration
Core
The IoC container

JEE

- **JMX**

  将自定义Bean输出到JMX

- **JMS**

  提供JMS模板

- **JCA CCI**

  以Spring的风格访问CCI 连接器

- 远程调用

  RMI, Spring HTTP调用程序, Hessian, Burlap, JAX RPC, JMS

- **EJBs**

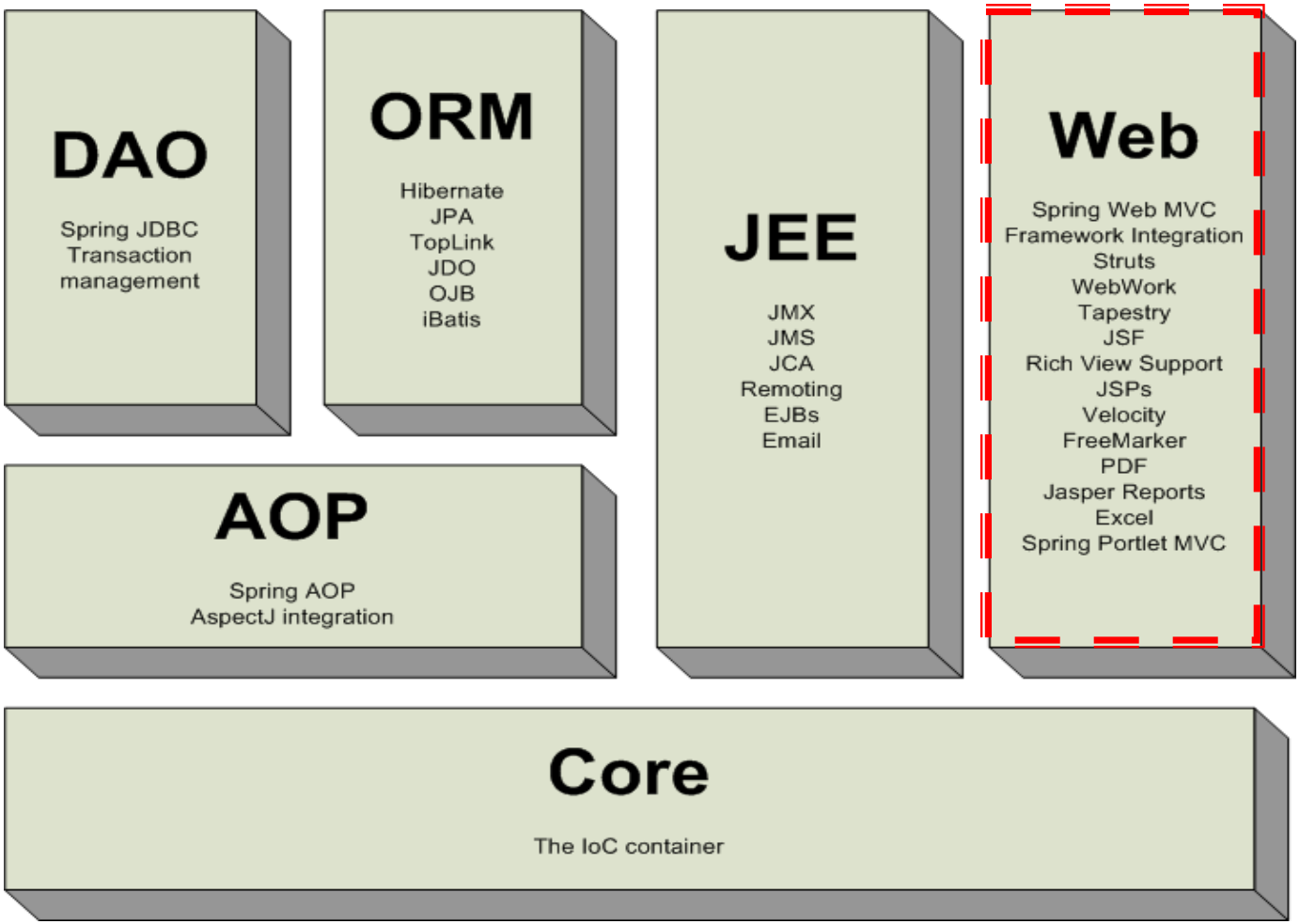
  访问EJBs, 方便EJB 实现

- **Email**

  提供方便的类库

Spring MVC

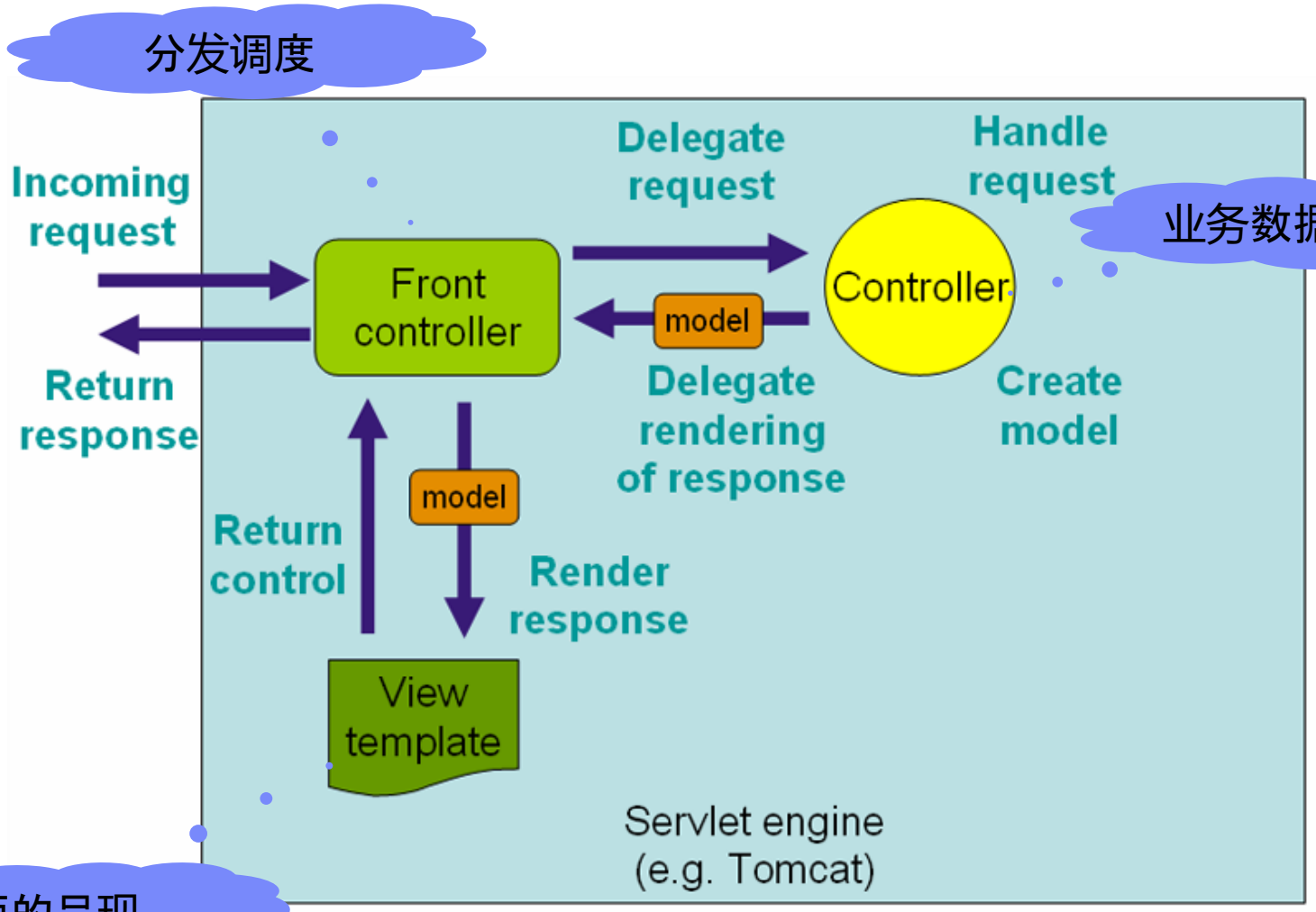
Spring Web


DAO
Spring JDBC
Transaction
management
ORM
Hibernate
JPA
TopLink
JDO
OJB
iBatis
JEE
JMX
JMS
JCA
Remoting
EJBs
Email
Web
Spring Web MVC
Framework Integration
Struts
WebWork
Tapestry
JSF
Rich View Support
JSPs
Velocity
FreeMarker
PDF
Jasper Reports
Excel
Spring Portlet MVC
AOP
Spring AOP
AspectJ integration
Core
The IoC container

Spring MVC

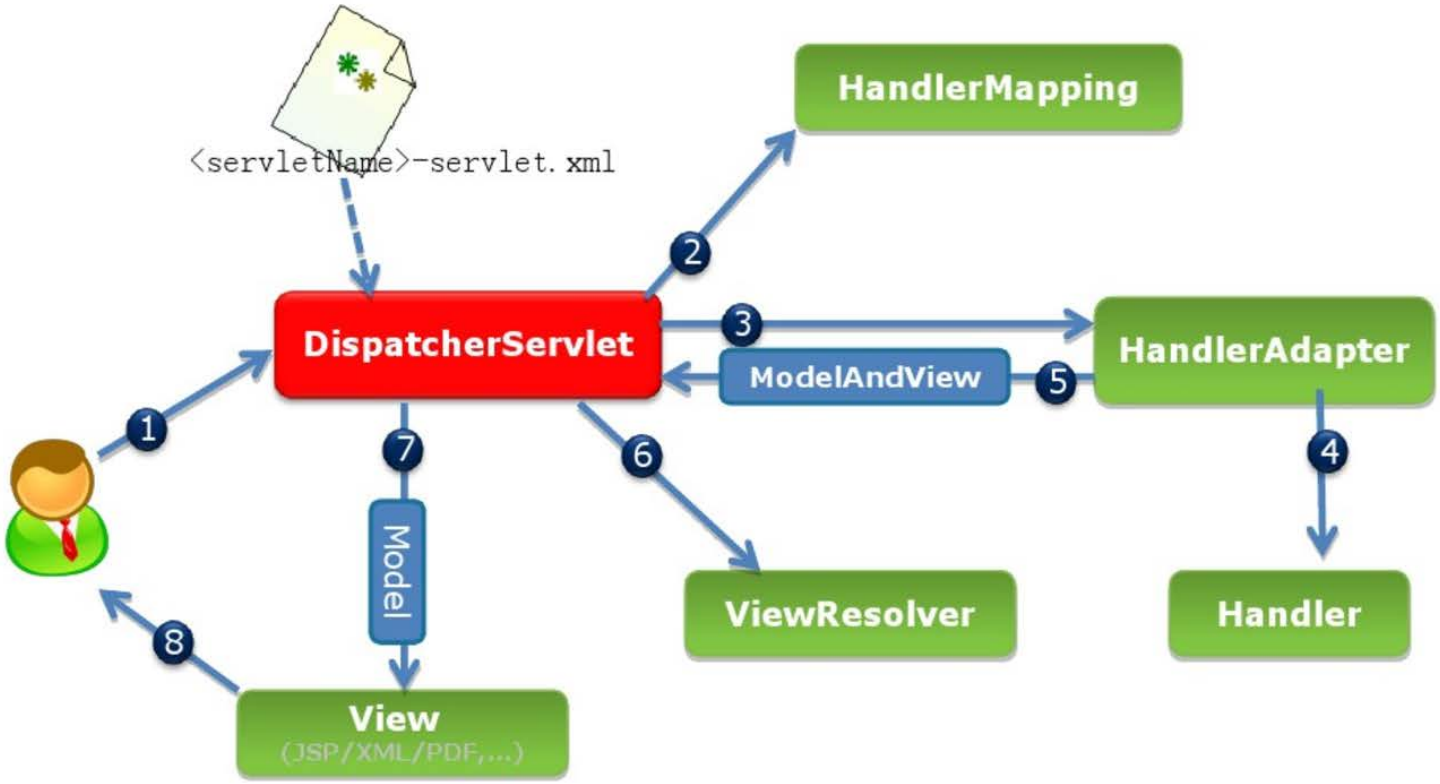
- 概述
  - **Spring MVC 是Spring 技术栈**为展现层提供的基于 **MVC** 设计理念的优秀的 Web 框架。
  - **Spring MVC** 通过一套 **MVC** 注解，让 POJO 成为处为处理请求的控制器，而无须实现任何接口。
  - 支持 **REST** 风格的 **URL**请求。
  - 采用了松散耦合可插拔组件结构，比其他 MVC 框架更具扩展性和灵活性。

Spring MVC


分发调度
Incoming request
Front controller
Delegate request
Handle request
Controller
model
Return response
Delegate rendering of response
Create model
model
Return control
Render response
View template
Servlet engine (e.g. Tomcat)
业务数据抽取
页面的呈现

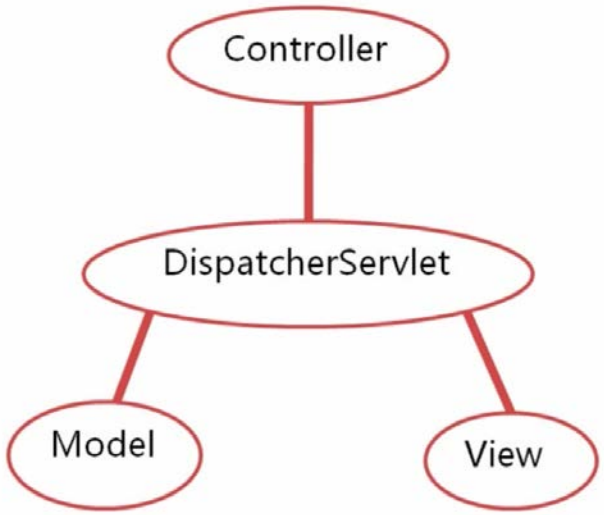
Spring MVC

- 处理流程



Spring MVC

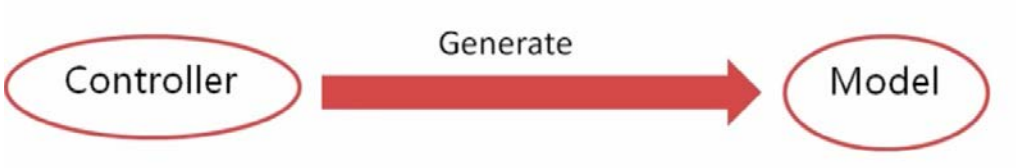
- **DispatcherServlet**
  - 前端控制器，进行调度




```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

Spring MVC

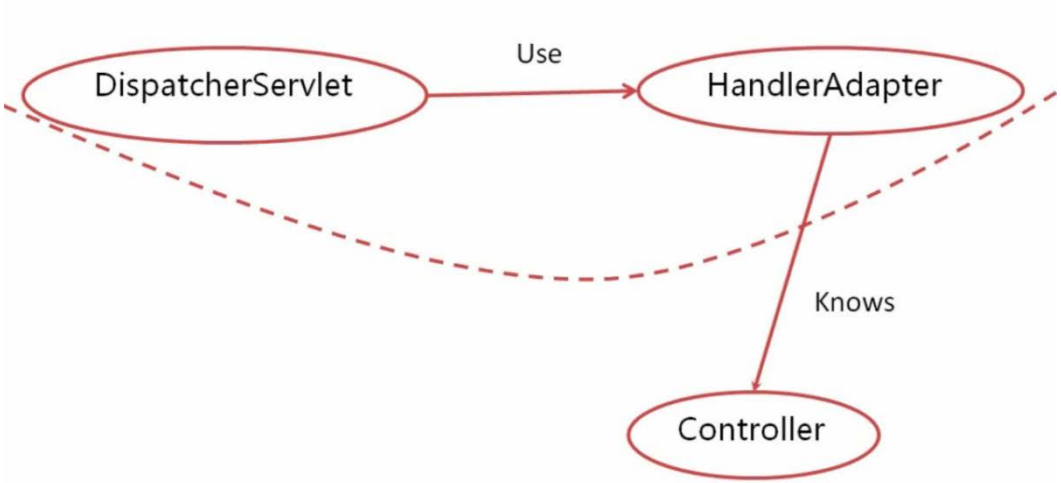
- 控制器类
  - **HandlerMapping**




```
@Controller                          ①← 将UserController变成一个Handler
@RequestMapping("/user")             ②←指定控制器映射的URL
public class UserController {
  @RequestMapping(value = "/register") ③←处理方法对应的URL，相对于②处的URL
   public String register() {
         return "user/register"; ④←返回逻辑视图名
    }
}
```

Spring MVC

- **HandlerAdapter**
  - 框架内部使用，使得**DispatcherServlet**知道调用哪个**Controller**



Spring MVC

- 简单示例：**HelloWorld**
  - 步骤：
    - **加入 SpringMVC 所需要的jar 包**
    - 在 web.xml 中配置 **DispatcherServlet**
    - 加入 Spring MVC 的配置文件
    - 编写处理请求的处理器，并标识为处理器
    - 编写视图

- **commons-logging-1.1.3.jar**
- spring-aop-4.0.0.RELEASE.jar
- spring-beans-4.0.0.RELEASE.jar
- spring-context-4.0.0.RELEASE.jar
- spring-core-4.0.0.RELEASE.jar
- spring-expression-4.0.0.RELEASE.jar
- **spring-web-4.0.0.RELEASE.jar**
- **spring-webmvc-4.0.0.RELEASE.jar**

Spring MVC

- 简单示例：**HelloWorld**
  - 步骤：
    - 加入 SpringMVC 所需要的jar 包
    - **在 web.xml 中配置 DispatcherServlet**
    - 加入 Spring MVC 的配置文件
    - 编写处理请求的处理器，并标识为处理器
    - 编写视图

```
<servlet>
<servlet-name>dispatcherServlet</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>
<param-name>contextConfigLocation</param-name>
<param-value>classpath:applicationContext-mvc.xml</param-value>
</init-param>

<load-on-startup>1</load-on-startup>
</servlet>

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

Spring MVC

- 简单示例：**HelloWorld**
  - 步骤：
    - 加入 SpringMVC 所需要的jar 包
    - 在 web.xml 中配置 DispatcherServlet
    - **加入 Spring MVC 的配置文件**
    - 编写处理请求的处理器，并标识为处理器
    - 编写视图

```
<!-- 配置自定扫描的包 -->
<context:component-scan base-package="fudan.adweb.springmvc">
</context:component-scan>
```

```
<!-- 配置视图解析器: 如何把 handler 方法返回值解析为实际的物理视图 -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
     <property name="prefix" value="/WEB-INF/views/"></property>
     <property name="suffix" value=".jsp"></property>
</bean>
```

Spring MVC

- 简单示例：**HelloWorld**
  - 步骤：
    - 加入 SpringMVC 所需要的jar 包
    - 在 web.xml 中配置 DispatcherServlet
    - 加入 Spring MVC 的配置文件
    - **编写处理请求的处理器，并标识为处理器**
    - 编写视图

```
@Controller
public class HelloWorld {

    @RequestMapping("/helloworld")
    public String hello(){
        System.out.println("hello world");
        return "success";
    }

}
```

Spring MVC

- 简单示例：**HelloWorld**

```
<servlet-mapping>
   <servlet-name>dispatcherServlet</servlet-name>
   <url-pattern>*. action </url-pattern>
</servlet-mapping>
```

web.xml

```
springmvc-1/helloWorld.action
```

url

```
@Controller
public class HelloWorldController {
    @RequestMapping("/helloWorld")
    public String helloWolrd(){
      System.out.println("HelloWorld SpringMVC");
      return "success";
    }
}
```

/WEB-INF/view/success.jsp

实际的物理视图

Handler

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/view/"></property>
  <property name="suffix" value=".jsp"></property>
</bean>
```

SpringMVC 配置文件

Spring MVC

- 映射请求参数、请求方法或请求头
  - @RequestMapping 的 value、method、params 及 heads属性值 分别表示请求 **URL**、请求方法、请求参数及请求头。它们之间是与的关系。

```
@RequestMapping(value="/delete", method=RequestMethod.POST, params="userId")
public String test1(){
    //...
    return "user/test1";
}

@RequestMapping(value="/show", headers="contentType=text/*")
public String test2(){
    //...
    return "user/test2";
}
```

Spring MVC

- 映射请求参数、请求方法或请求头
  - @RequestMapping 的 value、method、params 及 heads 分别表示请求 **URL**、请求方法、请求参数及请求头

示例1：

```
@RequestMapping(value="/delete")
public String test1(@RequestParam("userId") String userId){
    return "user/test1";
}
```

→所有URL为*<controllerURI>*/delete的请求由test1处理(任何请求方法)

示例2：

```
@RequestMapping(value="/delete",method=RequestMethod.POST)
public String test1(@RequestParam("userId") String userId){
    return "user/test1";
}
```

→所有URL为*<controllerURI>*/delete **且请求方法为POST** 的请求由test1处理

Spring MVC

- 映射请求参数、请求方法或请求头：注解
  - 通过 **@PathVariable** 可以将 URL 中占位符参数绑定到控 制器处理方法的入参中。即 URL 中的 {xxx} 占位符可以通过@PathVariable(“xxx”) 绑定到操作方法的入参中。这对REST化提供了很好的支持。

```
@RequestMapping("/delete/{id}")
public String delete(@PathVariable("id") Integer id){
    UserDao.delete(id);
    return "redirect:/user/list.action";
}
```

  - 在处理方法入参处使用 @RequestParam 可以把请求参 数传递给处理方法。

```
@RequestMapping(value = "/testRequestParam")
public String testRequestParam(
    @RequestParam(value = "username") String un,
    @RequestParam(value = "age", required = false, defaultValue = "0") int age) {
    System.out.println("testRequestParam, username: " + un + ", age: "
    + age);
    return SUCCESS;
}
```

Spring MVC

- 对REST的支持
  - HTTP 协议里面，四个表示操作方式的动词：
    - GET：/order/1 HTTP **GET** ：得到 id = 1 的 order
    - POST：/order HTTP **POST**：新增 order
    - PUT：/order/1 HTTP **PUT**：更新id = 1的 order
    - DELETE：/order/1 HTTP **DELETE**：删除 id = 1的 order
  - HiddenHttpMethodFilter
    - 浏览器 form 表单只支持 GET 与 POST 请求
    - Spring3.0 添加了一个过滤器，可以将这些请求转换为标准的 http 方法，使得支持 GET、POST、PUT 与DELETE 请求。

Spring MVC

- 对REST的支持
  - 在Web.xml中注册HiddenHttpMethodFilter

```
<filter>
<filter-name>HiddenHttpMethodFilter</filter-name>
<filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>

<filter-mapping>
<filter-name>HiddenHttpMethodFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

  - 表单中通过隐藏域通过post传递操作方法put

```
<form action="springmvc/testRest/1" method="post">
    <input type="hidden" name="_method" value="PUT"/>
    <input type="submit" value="TestRest PUT"/>
</form>
```

Spring MVC

- 映射请求参数、请求方法或请求头：注解
  - 通过 **@RequestHeader** 即可将请求头中的属性值绑定到处理方法的入参中。

```
@RequestMapping("/handle7")
public String handle7(@RequestHeader("Accept-Encoding") String encoding,
        @RequestHeader("Keep-Alive") long keppAlieve){
    return "success";
}
```

  - **@CookieValue** 可让处理方法入参绑定某个 Cookie 值。

```
@RequestMapping("/handle6")
public String handle6(@CookieValue(value="sessionId", required=false) String sessionId,
        @RequestParam("age") int age){
    return "success";
}
```

Spring MVC

- 使用 POJO 对象绑定请求参数值
  - Spring MVC 会按请求参数名和 POJO 属性名进行自动匹配，自动为该对象填充属性值。
  - 支持级联属性。

写一个User类，属性和表单中的属性一一对应

```
@RequestMapping("/testPojo")
public String testPojo(User user) {
    System.out.println("testPojo: " + user);
    return SUCCESS;
}
```

```
<form action="springmvc/testPojo" method="post">
    username: <input type="text" name="username"/>
    password: <input type="password" name="password"/>
    email: <input type="text" name="email"/>
    age: <input type="text" name="age"/>
    city: <input type="text" name="address.city"/>
    province: <input type="text" name="address.province"/>
    <input type="submit" value="Submit"/>
</form>
```

Spring MVC

- 使用 **Servlet API** 作为入参
  - 可以直接使用一些Servlet API作为入参。

```
@RequestMapping("/testServletAPI")
public void testServletAPI(HttpServletRequest request,
HttpServletResponse response, Writer out) throws IOException {
    System.out.println("testServletAPI, " + request + ", " + response);
    out.write("hello springmvc");
}
```

- HttpServletRequest
- HttpServletResponse
- HttpSession
- java.security.Principal
- Locale
- InputStream
- OutputStream
- Reader
- Writer

支持的servlet API

Spring MVC

- 处理数据模型
  - Spring MVC 提供了以下几种途径输出模型数据：
    - **ModelAndView：**
      - **处理方法体可通过该对象添加模型数据**
    - **Map 及 Model：**
      - **处理方法返回时，Map 中的数据会自动添加到模型中。**
    - **@SessionAttributes：**
      - **将模型中的某个属性暂存到 HttpSession 中**
    - **@ModelAttribute ：**
      - **方法入参标注该注解后, 入参的–对象就会放到数据模型中**

Spring MVC

- 处理数据模型
  - **ModelAndView**
    - **既包含视图信息，也包含模型数据信息。**
    - 添加模型数据：addObject方法，**SpringMVC 会把 ModelAndView 的 model 中数据放入到 request 域对象中。**
    - 设置视图:setViewName方法。

```
@RequestMapping(method = RequestMethod.POST)
public ModelAndView createUser(User user) {
        userService.createUser(user);
        ModelAndView mav = new ModelAndView();
        mav.setViewName("user/createSuccess");
        mav.addObject("user", user);
        return mav;
}
```

视图中访问通过
`${requestScope.user }`

Spring MVC

- 处理数据模型
  - Map 及 Model
    - Spring MVC 在内部使用了一个org.springframework.ui.Model 接口存储模型数据。或者入参为org.springframework.ui.ModelMap 或java.uti.Map存放信息。

```
@RequestMapping("/testMap")
public String testMap(Map<String, Object> map){
        System.out.println(map.getClass().getName());
        map.put("names", Arrays.asList("Tom", "Jerry", "Mike"));
        return SUCCESS;
}
```

```
names: ${requestScope.names }
<br><br>
```

Spring MVC

- 处理数据模型
  - @SessionAttributes
    - 在模型中对应的属性暂存到 HttpSession 中。
    - 除了可以通过属性名指定需要放到会话中的属性外，还可以通过模型属性的对象类型指定哪些模型属性需要放到会话中。
  - 例如
    - @SessionAttributes(types=User.class) 会将隐含模型中所有类型为 User.class 的属性添加到会话中。
    - @SessionAttributes(value={“user1”, “user2”})
    - @SessionAttributes(types={User.class, Dept.class})
    - @SessionAttributes(value={“user1”, “user2”}, types={Dept.class})

Spring MVC

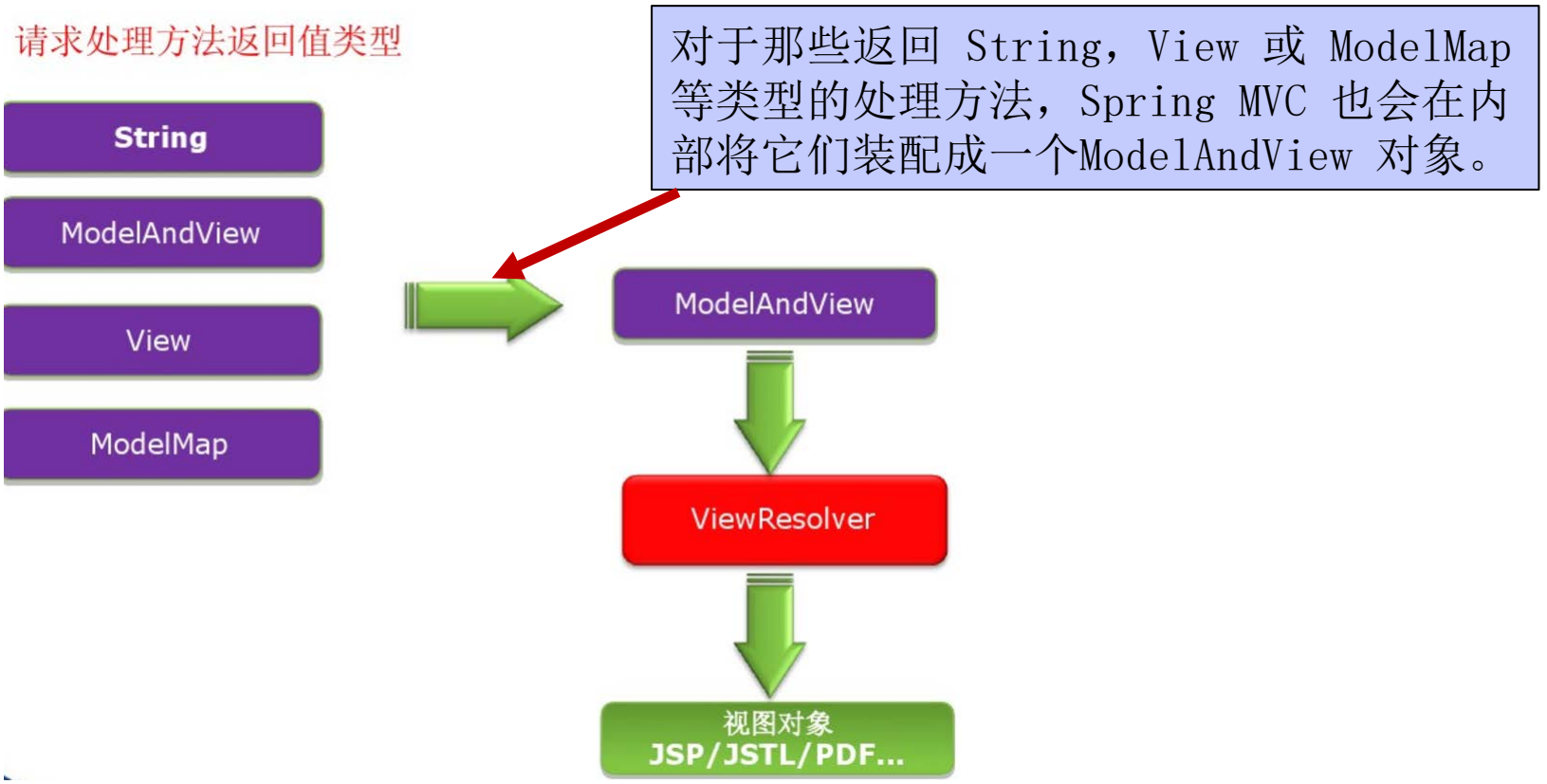
- 处理数据模型
  - **@ModelAttribute：**
    - **方法定义上时：Spring MVC 在调用目标处理方法前，会先逐个调用在方法级上标注了@ModelAttribute 的方法。**在方法的入参前使用 **@ModelAttribute** 注解：将方法入参对象和模型中对应。

```
@SessionAttributes("user")
@Controller
@RequestMapping("/hello")
public class HelloWorld {

   @ModelAttribute("user")
   public User getUser(){
      User user = new User();
      user.setAge(10);

      return user;
   }
```

```
@RequestMapping("/handle21")
public String handle21(@ModelAttribute("user") User user){
   user.setAge(22);
   return "redirect:/hello/handle22.action";
}

@RequestMapping("/handle22")
public String handle22(Map<String, Object> map,
      SessionStatus sessionStatus){
   User user = (User) map.get("user");
   user.setId(200);
   return "success";
}
```

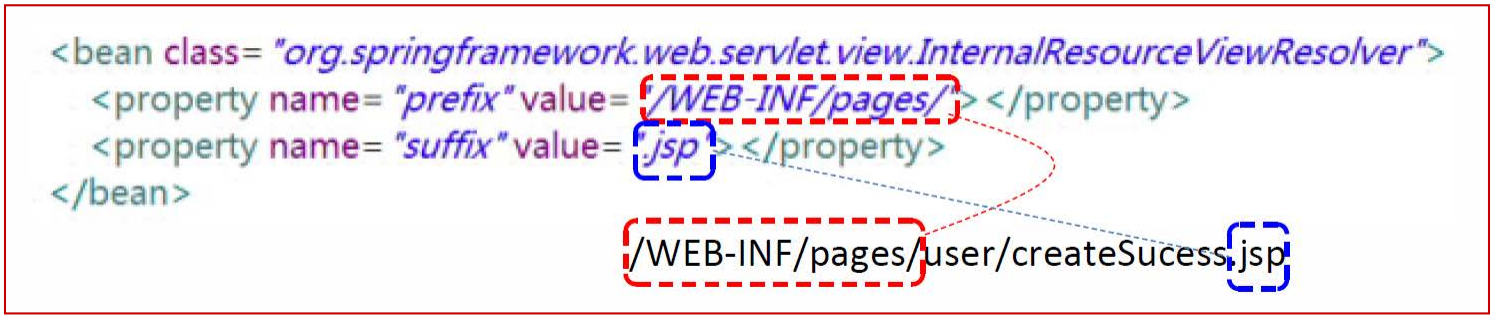
Spring MVC

- **Spring MVC如何解析视图**



Spring MVC

- **Spring MVC如何解析视图**
  - InternalResourceViewResolver
    - JSP 是最常见的视图技术，可以使用InternalResourceViewResolver 作为视图解析器：



Spring MVC

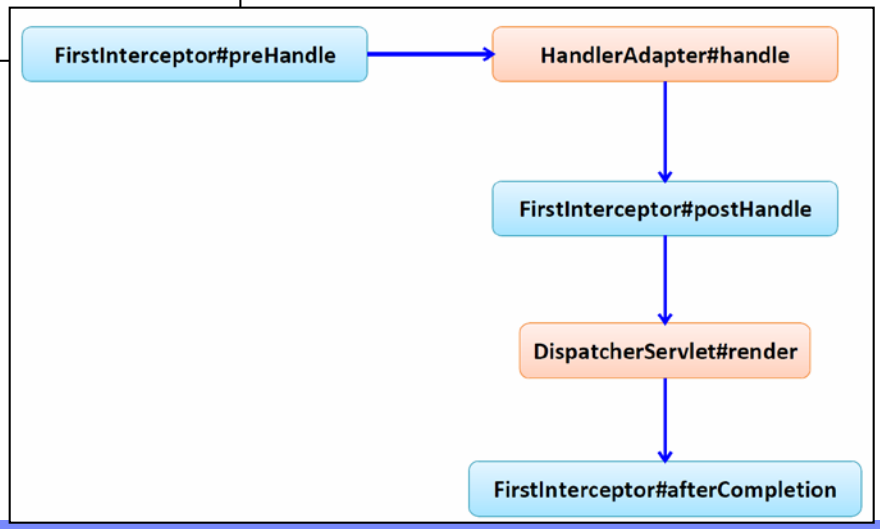
- **Spring MVC如何解析视图**
  - 重定向
    - 一般情况下，控制器方法返回字符串类型的值会被当成逻辑视图名处理。
    - 如果返回的字符串中带 forward: 或 redirect: 前缀时，SpringMVC 会对他们进行特殊处理：将 forward: 和redirect: 当成指示符，其后的字符串作为 URL 来处理

redirect:success.jsp：会完成一个到 success.jsp 的重定向的操作
forward:success.jsp：会完成一个到 success.jsp 的转发操作

Spring MVC

- 拦截器
  - Spring MVC使用拦截器对请求进行拦截处理。
  - 在**Handler**处理前后可以通过拦截实现一些功能

```
<!-- 拦截器设置 -->
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/test"/>
        <mvc:exclude-mapping path="/test/**"/>
        <bean class="com.zaprk.interceptor.MyInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```



Spring MVC

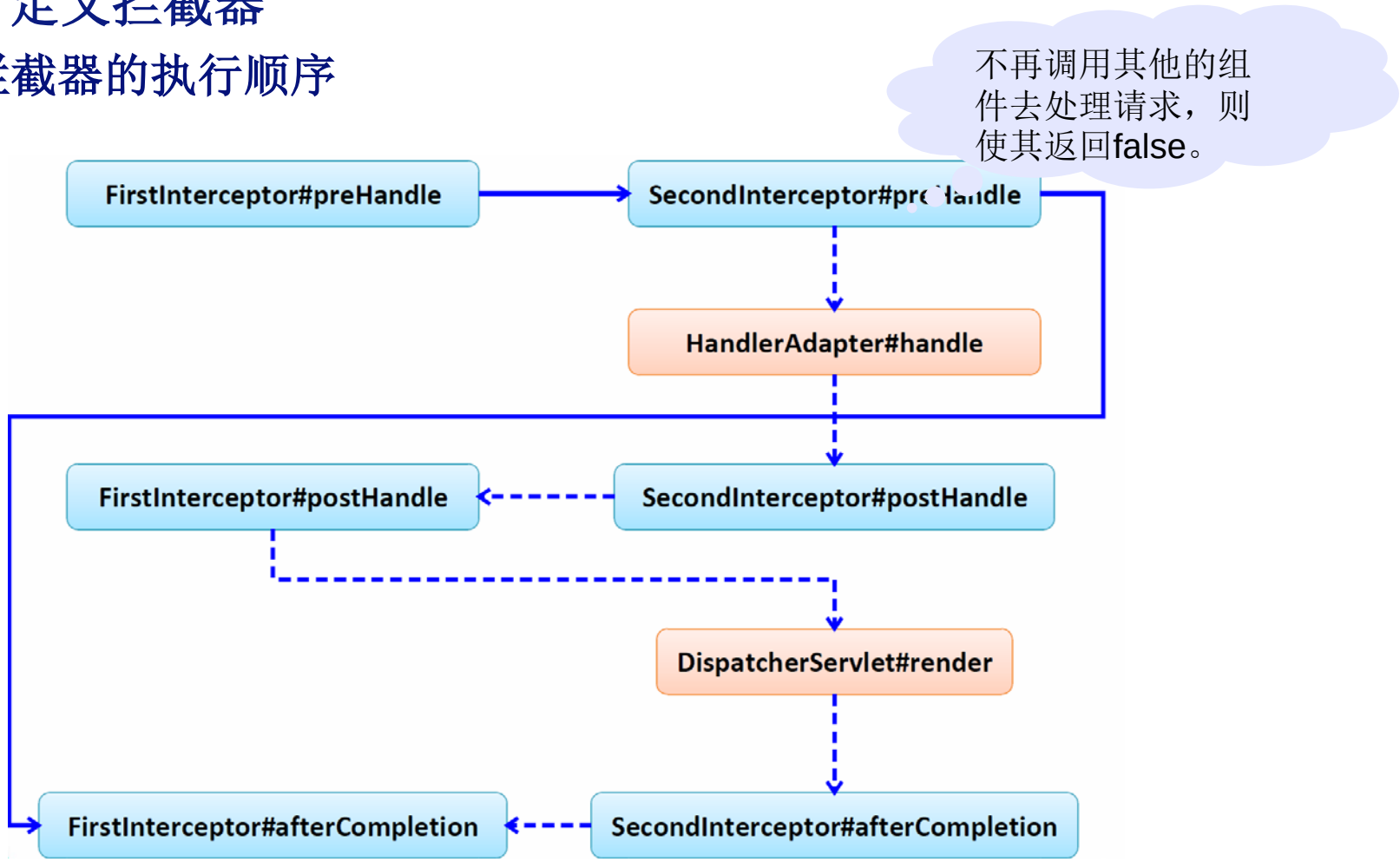
- 用户自定义拦截器
  - 实现**HandlerInterceptor**接口

```
public class MyInterceptor implements HandlerInterceptor {

    public void afterCompletion(HttpServletRequest request,
            HttpServletResponse response, Object target, Exception exception)
            throws Exception {
        // TODO Auto-generated method stub
        System.out.println("afterCompletion");
    }

    public void postHandle(HttpServletRequest request, HttpServletResponse response,
            Object t, ModelAndView modelAndView) throws Exception {
        // TODO Auto-generated method stub
        System.out.println("postHandle");
    }
    public boolean preHandle(HttpServletRequest request, HttpServletResponse responses,
            Object target) throws Exception {
        // TODO Auto-generated method stub
        System.out.println("----preHandle----");
        return true;
    }
}
```

Spring MVC

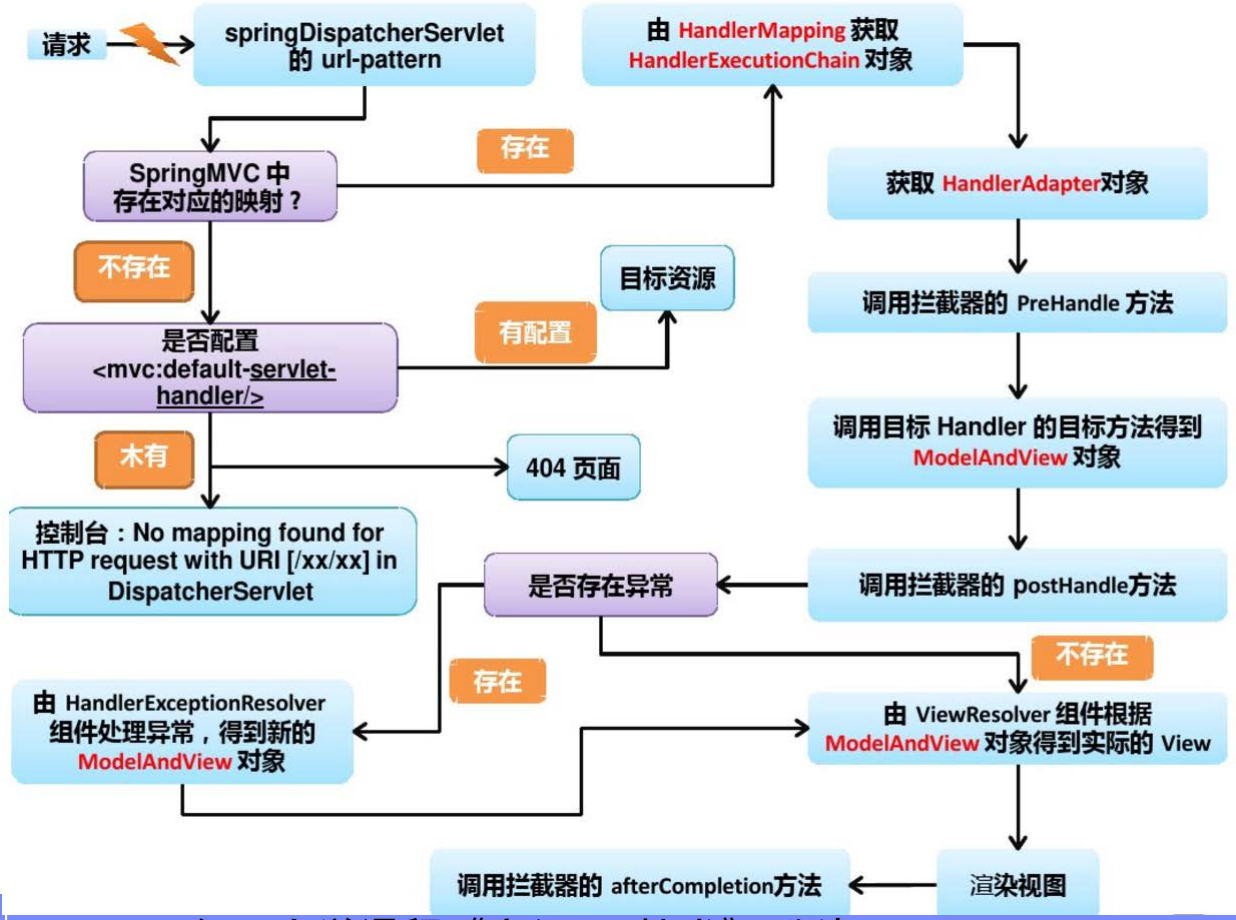
- 用户自定义拦截器

  多个拦截器的执行顺序



Spring MVC

- **Spring MVC详细运行流程**

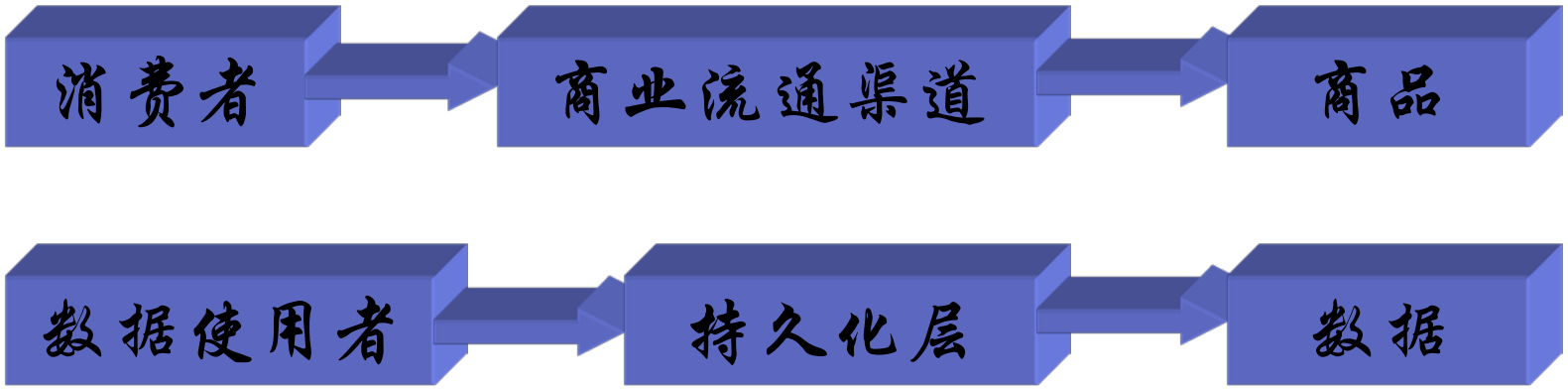


ORM框架： MyBatis

什么是持久层

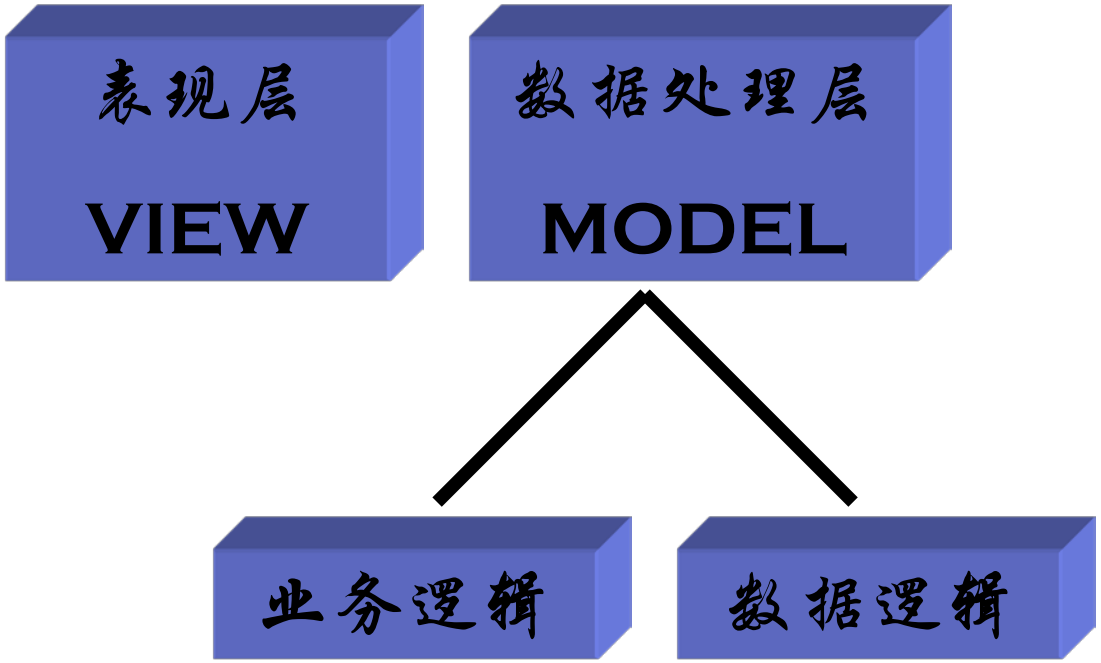
- 在系统逻辑层面上，专注于实现数据持久化的一个相对独立的领域(Domain)，将数据使用者和数据实体相关联。

- 类比：



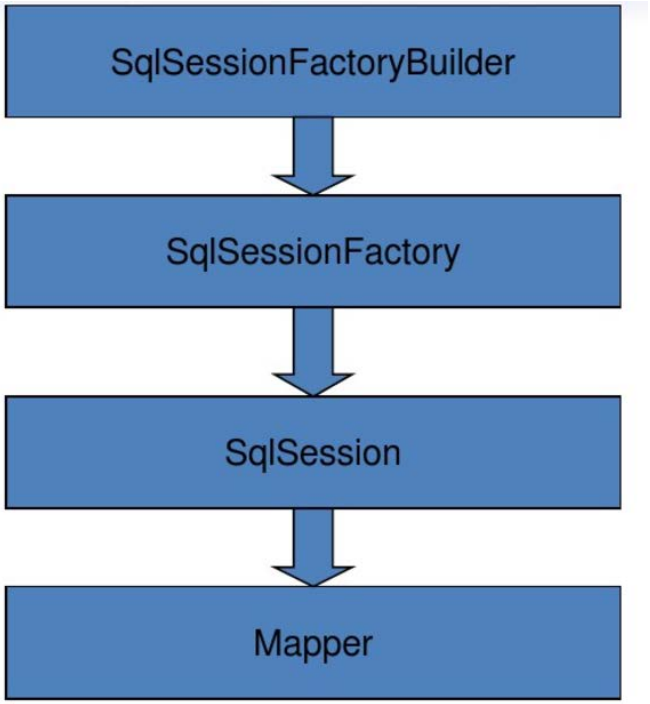
什么是持久层

- 解耦合



MyBatis 概述

- 编程模型



MyBatis 概述

- 数据库环境配置文件

```
<environments default="development">
    <environment id="development">
      <transactionManager type="JDBC">
        <property name="" value=""/>
      </transactionManager>
      <dataSource type="UNPOOLED">
        <property name="driver" value="com.mysql.jdbc.Driver"/>
        <property name="url"
value="jdbc:mysql://127.0.0.1:3306/micro_message"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
      </dataSource>
    </environment>
  </environments>

  <mappers>
    <mapper resource="adweb/fudan/config/sqlxml/Message.xml"/>
  </mappers>
```

MyBatis 概述

- 映射文件

```
<mapper namespace="Message">
        <select id="test" parameterType="adweb.fudan.bean.Message"
      resultMap="MessageResult">
         select ID,COMMAND,DESCRIPTION,CONTENT from MESSAGE where 1=1
        </select>
</mapper>
```

- 动态SQL

```
<select id="findActiveBlogWithTitleLike"
     parameterType="Blog" resultType="Blog">
  SELECT * FROM BLOG
  WHERE state =  'ACTIVE'
  <if test="title != null">
    AND title like #{title}
  </if>
</select>
```

MyBatis 概述

- 整合Spring
  - Jar文件：mybatis-spring-x.xx.jar
  - 配置文件

```
<bean id="sqlSessionFactory"
   class="org.mybatis.spring.SqlSessionFactoryBean">
<property name="dataSource" ref="dataSource" />
</bean>

<bean id="userMapper"
   class="org.mybatis.spring.mapper.MapperFactoryBean">
<property name="mapperInterface"
   value="org.mybatis.spring.sample.mapper.UserMapper" />
<property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>

<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManage
   r">
<property name="dataSource" ref="dataSource" />
</bean>
```

```
<bean
   class="org.mybatis.spring.mapper.MapperScannerConfigurer">
<property name="basePackage"
   value="org.mybatis.spring.sample.mapper" />
</bean>
```