



SPRING 3X

2016年06月

(北京邮电大学吴国仕)



Spring3.x 关键点及实例





什么是Spring

- Spring是分层的JavaSE/EE full-stack(一站式) 轻量级开源框架
- 以IoC（Inverse of Control 反转控制）和AOP（Aspect Oriented Programming 面向切面编程为内核）
- 官网: <http://www.springsource.org/>
- Spring的出现是为了取代EJB的臃肿、低效、脱离现实
 - ✓ *Expert One-to-One J2EE Design and Development*
 - ✓ *Expert One-to-One J2EE Development without EJB*



为什么要用 Spring

web tie 表现层

Servlet JSP

struts1 表现层框架 简化JSP和Servlet

Spring MVC

bussiness tie 业务逻辑层

EJB Enterprise JavaBean

IoC、AOP、声明式事务管理

EIS tie 集成层 (数据层、持久层)

hibernate框架 简化JDBC操作

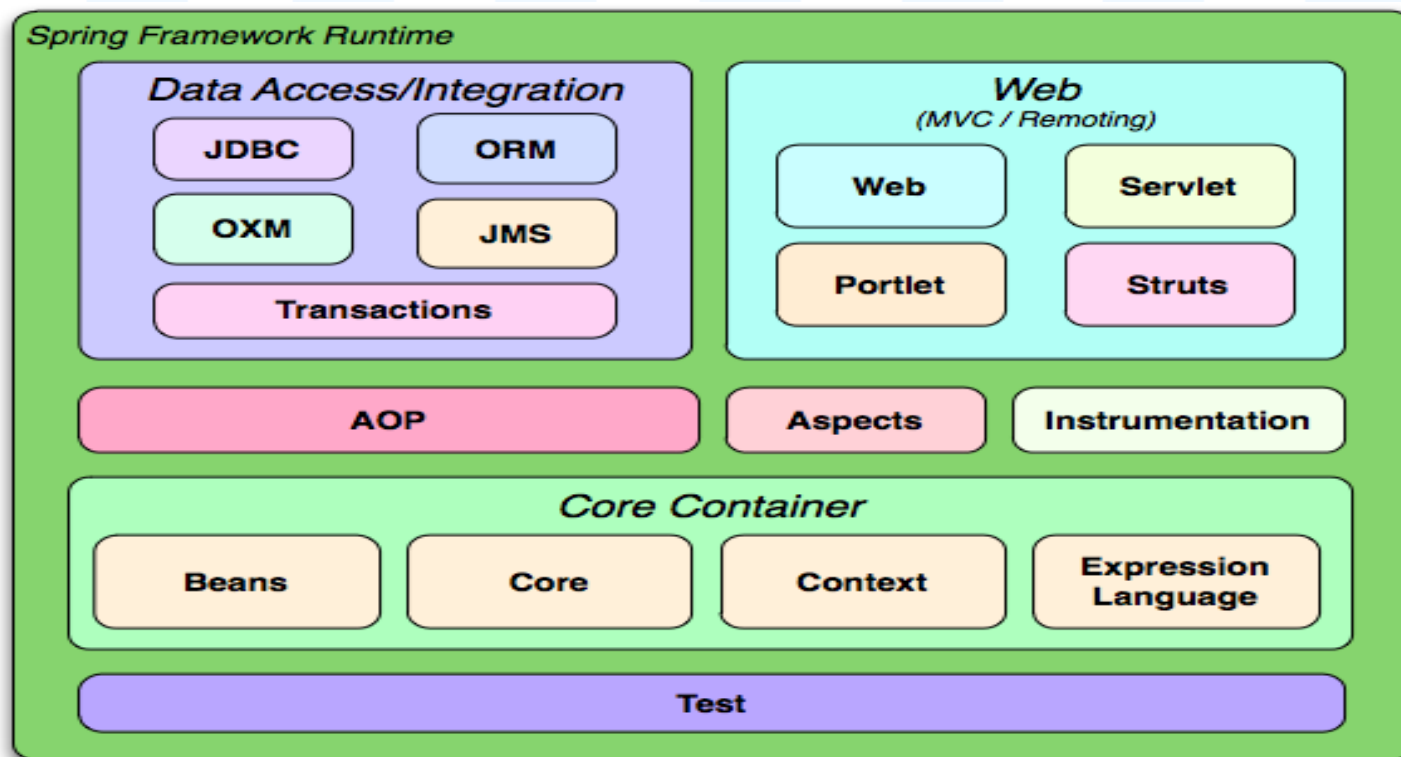
Spring JdbcTemplate
Spring提供对各种ORM框架整合
(MyBatis、Hibernate、JPA)



应用Spring的好处

- 方便解耦，简化开发
 - ✓ **Spring**就是一个大工厂，可以将所有对象创建和依赖关系维护，交给**Spring**管理
- **AOP**编程的支持
 - ✓ **Spring**提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能
- 声明式事务的支持
 - ✓ 只需要通过配置就可以完成对事务的管理，而无需手动编程
- 方便程序的测试
 - ✓ **Spring**对**Junit4**支持，可以通过注解方便的测试**Spring**程序
- 方便集成各种优秀框架
 - ✓ **Spring**不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：**Struts**、**Hibernate**、**MyBatis**、**Quartz**等）的直接支持
- 降低**JavaEE API**的使用难度
 - ✓ **Spring** 对**JavaEE**开发中非常难用的一些**API**（**JDBC**、**JavaMail**、远程调用等），都提供了封装，使这些**API**应用难度大大降低

Spring体系结构



Spring 框架是一个分层架构,它包含一系列的功能要素并被分为大约20个模块。这些模块分为Core Container、Data Access/Integration、Web、AOP (Aspect Oriented Programming)、Instrumentation和测试部分,如图所示。

IOC概念

Spring出现是为了取代EJB，轻量级控制反转

1、项目开发

表现层	业务层	数据层
Servlet	Service	DAO
Action		

表现层 调用 业务层
业务层 调用 数据层

```
UserService userService = new UserService();
```

2、面向接口编程

为什么？多态

```
包子 = new 包子(); 早点 = new 包子();  
new 油条();
```

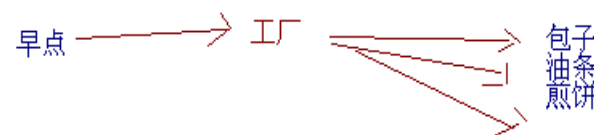
3、接口和实现耦合问题

早点 = new 包子();
针对早点编程

使用包子作为早点实现类，如果更换实现类，也必须修改代码

如何解除耦合??

4、引入工厂



```
早点 = 工厂.提供早点();
```

```
工厂 {  
    提供早点() {  
        return new 包子();  
    }  
}
```

5、引入配置文件

```
工厂 {  
    提供早点() {  
        读取配置文件  
    }  
}
```

properties
xml

```
Class.forName(实现类).newInstance();
```

早点= 包子

Spring 是IoC容器，相当于一个工厂，当需要一个接口实现类，之前自己new 创建，现在可以通过IoC容器获得，需要对象有IoC容器创建 ----- 控制反转

什么被反转？对象的创建权 被反转到IoC容器



DI 概念

问题：到底什么被IoC了？ 依赖的对象被IoC！

Servlet

~~IUserService userService = new UserService();~~ Spring提供工厂，获得UserService对象

userService.login(user);

class UserService implements IUserService {
 login() {
 IUserDAO userDAO = ~~new UserDAO();~~
 userDAO.login(user);
 }
}

Spring工厂获得UserDAO对象

Spring 容器负责对象创建

```
<bean id="userService" class="xxx.UserService" >  
    <property name="userDAO" ref="userDAO" />  
</bean>
```

描述对象之间依赖关系

```
<bean id="userDAO" class="xxx.UserDAO" >  
  
</bean>
```

Spring 在构造对象时，构造 UserService 和 UserDAO 两个对象，将 UserDAO对象，传递到UserService 中，建立引用关系（依赖注入）



Spring IoC控制反转快速入门案例

- 下载Spring最新开发包
- 复制Spring开发 jar包到工程
- 理解IoC控制反转和DI依赖注入
- 编写Spring核心配置文件
- 在程序中读取Spring配置文件，通过Spring框架获得Bean，完成相应操作



Spring IoC控制反转快速入门案例

- 下载Spring最新开发包
- 复制Spring开发 jar包到工程
- 理解IoC控制反转和DI依赖注入
- 编写Spring核心配置文件
- 在程序中读取Spring配置文件，通过Spring框架获得Bean，完成相应操作









Spring loc控制反转快速入门案例

➤ 官方下载Spring 3.x 最新开发版本

✓ <http://www.springsource.org/download/community>

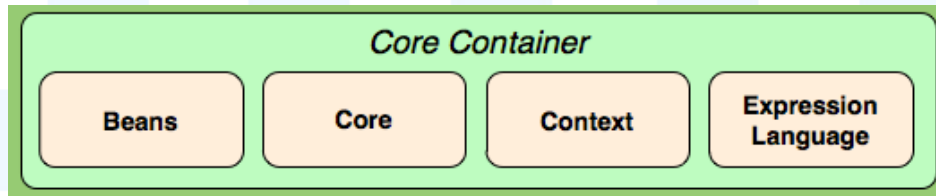
➤ Spring 3.2版本目录结构

名称	
 docs	api文档和开发规范
 libs	开发需要jar包 (源码)
 schema	开发需要schema文件
 license.txt	
 notice.txt	
 readme.txt	



Spring loc控制反转快速入门案例

➤ 导入Spring核心开发包到创建工程



- ✓ **spring-beans-3.2.0.RELEASE.jar**
- ✓ **spring-context-3.2.0.RELEASE.jar**
- ✓ **spring-core-3.2.0.RELEASE.jar**
- ✓ **spring-expression-3.2.0.RELEASE.jar**

➤ 还需要下载commons-logging日志包

- ✓ **commons-logging-1.1.1.jar** 集成log4j 导入log4j jar包

提示：spring3.0.X 版本 asm jar包 已经被合并到 spring core包中

Spring IoC控制反转快速入门案例



HelloTest类中使用 HelloService类对象

传统方式：HelloService helloService = new HelloService();

```
public interface IUserService {  
    /**  
     * 登陆  
     *  
     * @param username  
     * @return  
     */  
    public String login(String username);  
}
```

```
private String info = "登录成功";
```

```
// 提供注入
```

```
public void setInfo(String info) {  
    this.info = info;  
}
```

```
@Override
```

```
public String login(String username) {  
    return username + "," + info;  
}
```

```
public void testLogin() {
```

```
    IUserService userService = new UserServiceImpl();  
    String result = userService.login("bupt");  
    System.out.println(result);  
}
```

见例子Lect1_XML
bupt.spring.a_quicks
tart

IoC Inverse of Control 反转控制的概念，就是将原本在程序中手动创建HelloService对象的控制权，交由Spring框架管理，简单说，就是创建HelloService对象控制权被反转到了Spring框架



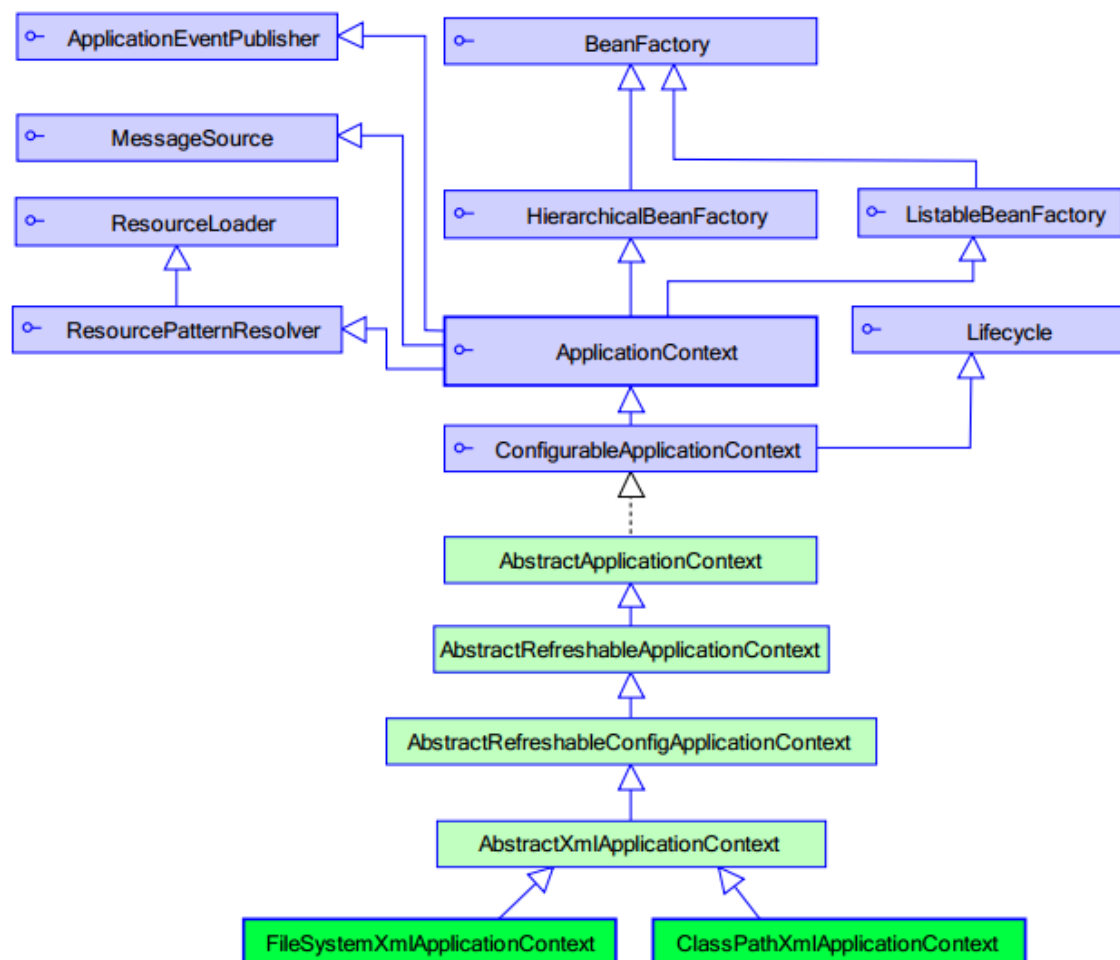
Spring ioc控制反转快速入门案例

- 配置spring核心配置文件，将HelloService的创建交由容器管理
- 在src下创建applicationContext.xml (习惯上)

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:p="http://www.springframework.org/schema/p"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- 配置程序 依赖对象 -->
    <bean id="userService" class="bupt.spring.a_quickstart.UserServiceImpl">
        <!-- userService 需要依赖info对象 -->
        <property name="info" value="登陆成功了" />
    </bean>
```

```
@Test
// 使用spring 管理依赖对象
public void testLogin2() {
    // 使用Spring容器获得Bean对象
    // 1、获得Spring 容器对象 (工厂对象) |
    ApplicationContext applicationContext =
        new ClassPathXmlApplicationContext("applicationContext.xml");
    // 2、通过工厂获得需要Bean对象
    IUserService userService =
        (IUserService) applicationContext.getBean("userService");
    String result = userService.login("buptsse");
    System.out.println(result);
}
```


BeanFactory和ApplicationContext





Spring IOC控制反转快速入门案例

```
@Test
// 使用BeanFactory 管理依赖对象
public void testLogin3() {
    // 使用Spring容器获得Bean对象
    // 1、获得Spring 容器对象 (工厂对象)
    BeanFactory beanFactory =
        new XmlBeanFactory(new ClassPathResource("applicationContext.xml"));
    // 2、通过工厂获得需要Bean对象
    IUserService userService =
        (IUserService) beanFactory.getBean("userService");
    String result = userService.login("BUPTSSE Log3");
    System.out.println(result);
}
```

通过 getBean方法获得Spring容器管理Bean对象



BeanFactory和ApplicationContext区别

- **BeanFactory** 采取延迟加载，第一次getBean时才会初始化Bean
- **ApplicationContext**是对BeanFactory扩展，提供了更多功能
 - ✓ 国际化处理
 - ✓ 事件传递
 - ✓ Bean自动装配
 - ✓ 各种不同应用层的Context实现



IoC容器装配Bean（xml配置方式）





三种实例化Bean的方式

1.使用类构造器实例化(默认无参数)

```
<bean id="bean1" class="bupt.spring.b_constructor.Bean1" />
```

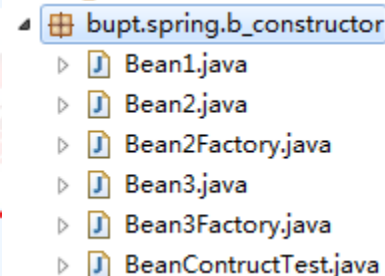
2.使用静态工厂方法实例化(简单工厂模式)

```
<bean id="bean2"  
      class="bupt.spring.b_constructor.Bean2Factory" factory-  
      method="createBean2"/>
```

3.使用实例工厂方法实例化(工厂方法模式):

```
<bean id="bean3factory"  
      class="bupt.spring.b_constructor.Bean3Factory"/>  
<bean id="bean3" factory-bean="bean3factory" factory-  
      method="getBean3"></bean>
```

见例子





Bean的作用域

类别	说明
singleton	在Spring IoC容器中仅存在一个Bean实例，Bean以单例方式存在
prototype	每次从容器中调用Bean时，都返回一个新的实例，即每次调用getBean()时，相当于执行new XxxBean()
request	每次HTTP请求都会创建一个新的Bean，该作用域仅适用于WebApplicationContext环境
session	同一个HTTP Session 共享一个Bean，不同Session使用不同Bean，仅适用于WebApplicationContext 环境
globalSession	一般用于Portlet应用环境，该作用域仅适用于WebApplicationContext 环境

开发时通常使用前两种：

- Singleton 单例，在一个Spring容器上下文中，只存在一个对象
- Prototype 多例，在每次getBean时，都会重新创建一个Bean实例

见例子

```
bupt.spring.c_scope
├─ BeanScopeTest.java
├─ PrototypeBean.java
└─ SingletonBean.java
```



Spring容器中Bean的生命周期

Spring初始化bean或销毁bean时，有时需要作一些处理工作，因此spring可以在创建和拆卸bean的时候调用bean的两个生命周期方法。

```
<bean id="foo" class="...Foo"  
      init-method="setup"  
      destroy-method="teardown"/>
```

当bean被载入到容器的时候调用
setup

当bean从容器中删除的时候调用
teardown(scope= singleton有效)

web容器中会自动调用，但是main函数或
测试用例需要手动调用



Spring容器中Bean的生命周期

见例子：

```
└─ bupt.spring.d_lifecycle
   └─ LifeCycleBean.java
   └─ LifeCycleBeanTest.java
   └─ MyBeanPostProcessor.java
```

```
<bean id="lifeCycleBean"
      class="bupt.spring.d_lifecycle.LifeCycleBean" init-
      method="setup" destroy-method="teardown" />
```

```
public void demo1() {
    ClassPathXmlApplicationContext applicationContext = new
    ClassPathXmlApplicationContext("applicationContext.xml");
    LifeCycleBean lifeCycleBean = (LifeCycleBean)
    applicationContext.getBean("lifeCycleBean");
    System.out.println(lifeCycleBean);
    // 调用容器close方法
    applicationContext.close(); }

```




依赖注入Bean的属性

- 对于类成员变量，注入方式有三种
 - ✓ 构造函数注入
 - ✓ 属性setter方法注入
 - ✓ 接口注入
- **Spring支持前两种**



依赖注入Bean的属性

依赖注入三种方式:

1、构造方法参数注入

```
class UserService {  
    private UserDao userDao ;  
  
    public UserService(UserDao userDao) { this.userDao = userDao ;}  
}
```

2、setter方法注入

```
class UserService {  
    private UserDao userDao ;  
  
    public void setUserDao(UserDao userDao ){  
        this.userDao = userDao ;  
    }  
}
```

setter 方法概念来自JavaBean

属性name ---- 属性名首字母大写 + set前缀 --- setName (name属性setter方法)

3、接口注入

```
Interface UserDaoAware {  
    public void injectUserDao(UserDao userDao );  
}  
  
class UserService implements UserDaoAware {  
  
    private UserDao userDao ;  
  
    public void injectUserDao(UserDao userDao) {  
        this.userDao = userDao ;  
    }  
}
```

Spring框架支持 构造参数注入 和 setter方法注入

Spring支持前两种依赖注入。



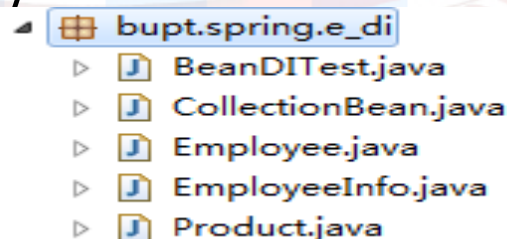
依赖注入Bean属性 -- 构造方法注入

- 使用构造方法注入，在Spring配置文件中，通过<constructor-arg>设置注入的属性 (可以通过index或者type注入)

```
public class Car {  
    private String name;  
    private double price;  
  
    public Car(String name, double price) {  
        this.name = name;  
        this.price = price;  
    }  
  
    public void showInfo() {  
        System.out.println("name:" + name + ",price:" + price);  
    }  
}
```

```
<bean id="product" class="bupt.spring.e_di.Product" >  
  <constructor-arg index="0" type="java.lang.String" value="笔记本" />  
  <constructor-arg index="1" type="double" value="4000" />  
</bean>
```

见例子





依赖注入Bean属性 -- setter方法注入

- 使用setter方法注入，在Spring配置文件中，通过<property>设置注入的属性，使用<property>还可以引入引用其他Bean

```
public class Employee {  
    private String name;  
    private int age;  
  
    private Product product;  
  
    // 为每个属性提供setter 方法  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public void setAge(int age) {  
        this.age = age;  
    }  
  
    public void setProduct(Product product) {  
        this.product = product;  
    }  
}  
  
<!-- setter 方法注入 -->  
<bean id="employee" class="bupt.spring.e_di.Employee">  
    <!--  
        name 对应setter 方法属性名  
        value 用于基本数据类型或者String 数据类型属性注入  
        ref 用于复杂自定义数据类型属性注入  
    -->  
    <property name="name" value="张三" />  
    <property name="age" value="20" />  
    <property name="product" ref="product" />  
</bean>
```



集合类型属性注入

- Spring对象java.util中常用集合对象提供了专门的配置标签
- java.util包中常用集合
 - ✓ List
 - ✓ Set
 - ✓ Map
 - ✓ Properties





集合类型属性注入 -- List（数组）

```
public class CollectionBean {  
    private List<String> list;  
    private String[] args;  
  
    public void setList(List<String> list) {  
        this.list = list;  
    }  
  
    public void setArgs(String[] args) {  
        this.args = args;  
    }  
}
```

```
<property name="list">  
    <list>  
        <value>音乐</value>  
        <value>读书</value>  
    </list>  
</property>
```

```
<property name="args">  
    <list>  
        <value>篮球</value>  
        <value>足球</value>  
    </list>  
</property>
```



集合类型属性注入 -- Set

```
public class CollectionBean {  
    private Set<String> set;  
  
    public Set<String> getSet() {  
        return set;  
    }  
  
    public void setSet(Set<String> set) {  
        this.set = set;  
    }  
}
```

```
<property name="set">  
    <set>  
        <value>金牌会员</value>  
        <value>银牌会员</value>  
    </set>  
</property>
```




集合类型属性注入 -- Map

```
<property name="map">
  <map>
    <entry>
      <key>
        <value>公司</value>
      </key>
      <value>传智播客</value>
    </entry>
    <entry>
      <key>
        <value>城市</value>
      </key>
      <value>北京</value>
    </entry>
  </map>
</property>
```

简化

```
public class CollectionBean {
    private Map<String, String> map;

    public Map<String, String> getMap() {
        return map;
    }

    public void setMap(Map<String, String> map) {
        this.map = map;
    }
}
```

```
<property name="map">
  <map>
    <entry key="公司" value="传智播客"/>
    <entry key="城市" value="北京"/>
  </map>
</property>
```



集合类型属性注入 -- Properties

```
public class CollectionBean {  
    private Properties properties;  
  
    public Properties getProperties() {  
        return properties;  
    }  
  
    public void setProperties(Properties properties) {  
        this.properties = properties;  
    }  
}
```

```
<property name="properties">  
    <props>  
        <prop key="公司">传智播客</prop>  
        <prop key="城市">北京</prop>  
    </props>  
</property>
```



Web开发中应用Spring 框架





Web应用中使用Spring

- 导入Spring开发基本jar包
 - ✓ spring-beans-3.2.0.RELEASE.jar
 - ✓ spring-context-3.2.0.RELEASE.jar
 - ✓ spring-core-3.2.0.RELEASE.jar
 - ✓ spring-expression-3.2.0.RELEASE.jar
- 导入commons-logging的jar包
 - ✓ commons-logging-1.1.1.jar
- 导入Spring web开发jar包
 - ✓ spring-web-3.2.0.RELEASE.jar



web.xml

配置web.xml

- 将Spring容器初始化，交由web容器负责
- 配置核心监听器 ContextLoaderListener
- 配置全局参数contextConfigLocation
 - ✓ 用于指定Spring的框架的配置文件位置

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:applicationContext.xml</param-value>
</context-param>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```



获得WebApplicationContext对象

- 因为Spring容器已经交由web容器初始化和管理的
- 获得WebApplicationContext对象，需要依赖ServletContext对象
 - ✓ 通常在Servlet中完成

```
WebApplicationContext applicationContext =  
    WebApplicationContextUtils.getWebApplicationContext(getServletContext());
```

- 另一种方式

```
WebApplicationContext applicationContext = (WebApplicationContext)  
    getServletContext().getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_C  
    ONTEXT_ATTRIBUTE);
```

见例子：spring3_lec1_web



使用JUnit测试Spring程序



使用junit4测试Spring

➤ 导入Spring test测试jar包

✓ spring-test-3.2.0.RELEASE.jar

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:applicationContext.xml")
public class SpringTest {
    @Autowired
    private HelloService helloService;

    @Test
    public void demo() {
        helloService.sayHello();
    }
}
```



使用junit4测试Spring

➤ 运行测试

在HelloServiceTest.java， 点击右键， 然后运行Runas， 点击JUnit Test， 即可测试功能。

```
16:14:39,839 INFO XmlBeanDefinitionReader:315 - L
16:14:39,908 INFO GenericApplicationContext:510 -
16:14:39,951 INFO DefaultListableBeanFactory:577
hello web, abc
16:14:39,969 INFO GenericApplicationContext:1042
16:14:39,969 INFO DefaultListableBeanFactory:444
```



AOP 面向切面编程





什么是AOP

- **AOP Aspect Oriented Programing** 面向切面编程（是**OOP**的延伸）
- **AOP**采取横向抽取机制，取代了传统纵向继承体系重复性代码（性能监视、事务管理、安全检查、缓存）
- **Spring AOP**使用纯**Java**实现，不需要专门的编译过程和类加载器，在运行期通过代理方式向目标类织入增强代码
- **AspectJ**是一个基于**Java**语言的**AOP**框架，**Spring2.0**开始，**Spring AOP**引入对**Aspect**的支持
 - ✓ 性能监视：通过日志记录方法运行时间，找出运行最慢方法，进行优化
 - ✓ 事务管理：将开启事务，提交事务，回滚事务操作放入**AOP**代理中，目标中只需要执行**SQL**语句
 - ✓ 安全检查：在**AOP**代理中，判断用户是否具有目标方法访问权限，如果没有阻止访问
 - ✓ 缓存：在**AOP**代理中，将第一次访问返回数据结果放入缓存，后面每次访问从缓存返回数据，不需要执行目标方法



什么是AOP

纵向继承体系 代码复用

```
class UserDAO {  
    save() {...}  
    delete() {...}  
}
```

记录日志

```
class ProductDAO {  
    add() {...}  
    update() {...}  
}
```

记录日志

在DAO 进行增删改查时，记录方法运行日志（运行时间）

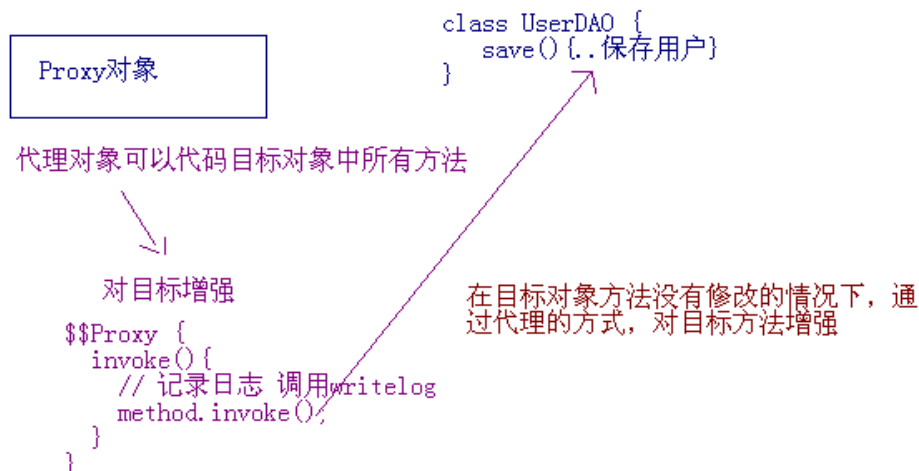
public void writeLog() {...} 抽取记录日志方法

面向对象 通过 继承实现代码复用

```
public abstract class BaseDAO {  
    public void writeLog() {  
        ....  
    }  
}
```

```
class UserDAO extends BaseDAO {}  
class Product extends BaseDAO {}
```

AOP 横向抽取机制 ---- 代理技术



问题：Java语言是单继承，使用继承来完全实现代码复用，不灵活，不便于维护



如何实现AOP 编程

➤ Spring 提供两种实现AOP 编程方式

- ✓ Spring AOP使用纯Java实现，不需要专门的编译过程和类加载器，在运行期通过代理方式向目标类织入增强代码（Spring 1.2时代）
- ✓ AspectJ是一个基于Java语言的AOP框架，Spring2.0开始，Spring AOP引入对Aspect的支持

➤ 很多Spring 内置功能（事务管理），使用Spring1.2 AOP编程，如果需要自己定义AOP切面代理主流使用 Spring2.0之后和AspectJ 整合方式

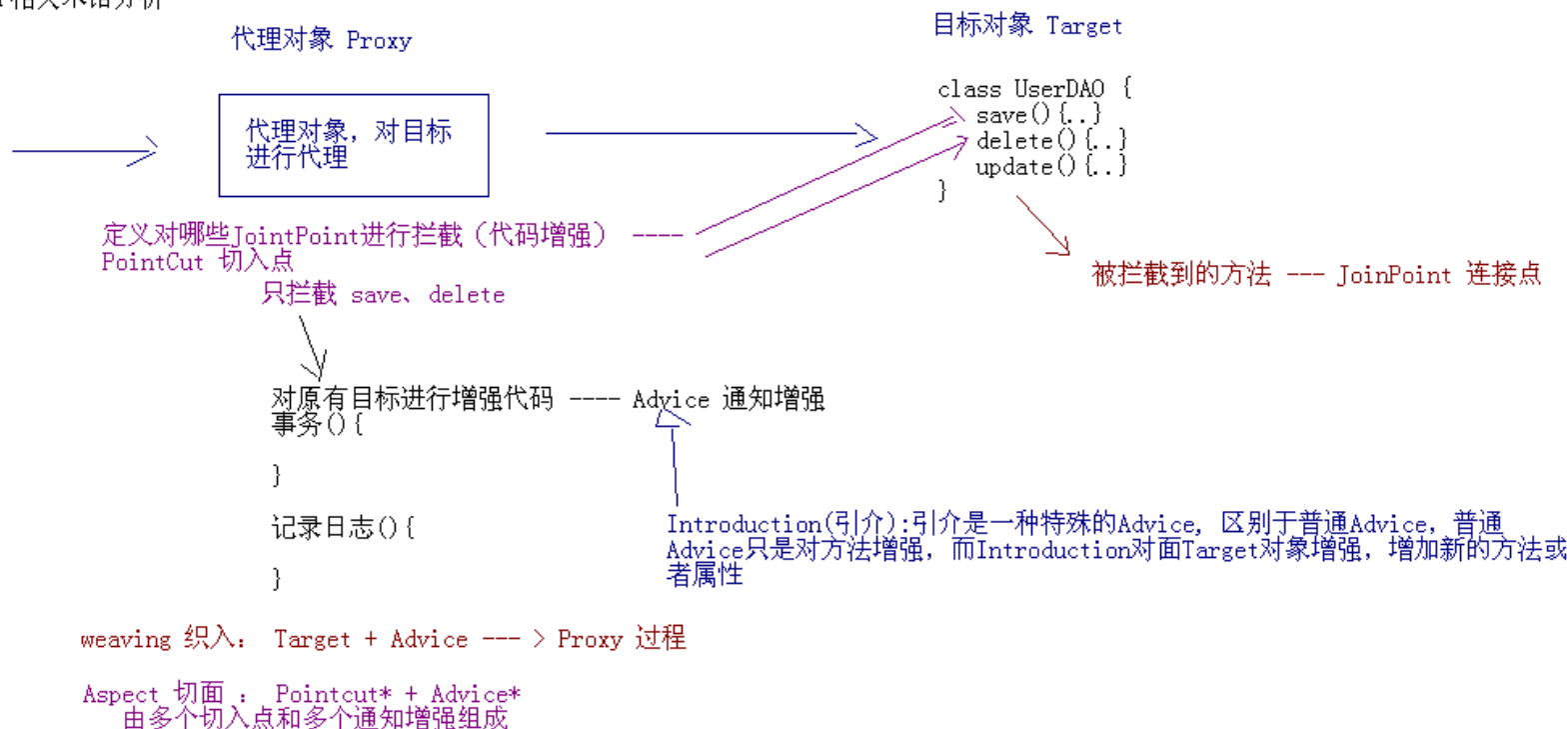


AOP相关术语

- **Joinpoint(连接点)**:所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点.
- **Pointcut(切入点)**:所谓切入点是指我们要对哪些Joinpoint进行拦截的定义.
- **Advice(通知/增强)**:所谓通知是指拦截到Joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知(切面要完成的功能)
- **Introduction(引介)**: 引介是一种特殊的通知在不修改类代码的前提下, Introduction可以在运行期为类动态地添加一些方法或Field.
- **Target(目标对象)**:代理的目标对象
- **Weaving(织入)**: 是指把增强应用到目标对象来创建新的代理对象的过程.
spring采用动态代理织入, 而AspectJ采用编译期织入和类装在线织入
- **Proxy (代理)**: 一个类被AOP织入增强后, 就产生一个结果代理类
- Aspect(切面)**: 是切入点和通知 (引介) 的结合

AOP相关术语

AOP相关术语分析





代理技术

- 代理代码实现，分为动态代理（代理类是在虚拟机内部动态构造）和静态代理（用户手动编写代理类，对目标类代理）
 - ✓ Spring AOP 底层使用动态代理（Struts2 框架内核 StrutsActionProxy 类，基于AOP思想，静态代理）
 - ✓ 动态代理实现技术有哪些？JDK动态代理、Javassist 动态代理、Cglib 动态代理
 - ✓ Spring 底层使用 JDK动态代理 + Cglib 动态代理



JDK动态代理

- JDK1.3引入动态代理技术
- 编写动态代理程序
 - ✓ `java.lang.reflect.Proxy`
 - ✓ `java.lang.reflect.InvocationHandler`

```
public interface UserService {  
    public void regist(User user);  
}
```

```
public class JDKProxy implements InvocationHandler {  
  
    private Object target;  
  
    public Object getProxy(Object target) {  
        this.target = target;  
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),  
            target.getClass().getInterfaces(), this);  
    }  
  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args)  
        throws Throwable {  
        System.out.println("代理前操作...");  
        Object result = method.invoke(target, args);  
        System.out.println("代理后操作...");  
        return result;  
    }  
}
```

见例子：
spring3_lec2_aop下
的a_jdkproxy

JDK动态代理

JDK Proxy 内存实现原理

```
@Test
public void demo2() {
    // 对目标进行代理
    IUserService target = new UserServiceImpl(); // 获得目标
    // 构造代理工厂 1
    JdkProxyFactory factory = new JdkProxyFactory(target);
    // 通过工厂创建代理
    IUserService proxy = (IUserService) factory.createProxy();

    proxy.login();
}
```

```
public class JdkProxyFactory implements InvocationHandler {

    // 被代理目标对象
    private Object target;

    public JdkProxyFactory(Object target) {
        this.target = target;
    }

    // 创建代理对象
    public Object createProxy() {
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
            target.getClass().getInterfaces(), this);
    }

    @Override
    // 拦截目标对象, 执行invoke方法
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("执行代理...");
        return method.invoke(target, args);
    }
}
```

```
class $$Proxy implements IUserService {
    InvocationHandler invocationHandler;

    public void login() {
        invocationHandler.invoke(loginmethod, args);
    }
}
```

target
UserServiceImpl



使用CGLIB生成代理

- 对于不使用接口的业务类，无法使用JDK动态代理
- CGLib采用非常底层字节码技术，可以为一个类创建子类，解决无接口代理问题

```
public class CglibProxy implements MethodInterceptor {  
  
    private Enhancer enhancer = new Enhancer();  
  
    public Object getProxy(Class clazz) {  
        enhancer.setSuperclass(clazz);  
        enhancer.setCallback(this);  
        return enhancer.create();  
    }  
  
    @Override  
    public Object intercept(Object obj, Method method, Object[] args,  
        MethodProxy proxy) throws Throwable {  
        System.out.println("代理前操作...");  
        Object result = proxy.invokeSuper(obj, args);  
        System.out.println("代理后操作...");  
        return result;  
    }  
}
```



使用CGLIB生成代理

➤ 关于intercept拦截方法

```
/**
 * @param obj CGLib根据指定父类生成的代理对象
 * @param method 拦截的方法
 * @param args 拦截方法的参数数组
 * @param proxy 方法的代理对象，用于执行父类的方法
 * @return
 */
public Object intercept(Object obj, Method method, Object[] args,
    MethodProxy proxy) throws Throwable {
    ... ..
}
```

最新版本Spring已经将CGLib开发类引入spring-core-3.2.0.RELEASE.jar



代理知识总结

- Spring在运行期，生成动态代理对象，不需要特殊的编译器
- Spring AOP的底层就是通过JDK动态代理或CGLib动态代理技术 为目标Bean执行横向织入
 - 1.若目标对象实现了若干接口，spring使用JDK的java.lang.reflect.Proxy类代理。
 - 2.若目标对象没有实现任何接口，spring使用CGLIB库生成目标对象的子类。
- 程序中应优先对接口创建代理，便于程序解耦维护
- 标记为final的方法，不能被代理，因为无法进行覆盖
 - ✓ JDK动态代理，是针对接口生成子类，接口中方法不能使用final修饰
 - ✓ CGLib 是针对目标类生产子类，因此类或方法 不能使final的
- Spring只支持方法连接点，不提供属性连接



Spring AOP增强类型

- AOP联盟为通知Advice定义了org.aopalliance.aop.Interface.Advice
- Spring按照通知Advice在目标类方法的连接点位置，可以分为5类
 - ✓ 前置通知 org.springframework.aop.MethodBeforeAdvice
 - 在目标方法执行前实施增强
 - ✓ 后置通知 org.springframework.aop.AfterReturningAdvice
 - 在目标方法执行后实施增强
 - ✓ 环绕通知 org.aopalliance.intercept.MethodInterceptor
 - 在目标方法执行前后实施增强
 - ✓ 异常抛出通知 org.springframework.aop.ThrowsAdvice
 - 在方法抛出异常后实施增强
 - ✓ 引介通知 org.springframework.aop.IntroductionInterceptor
 - 在目标类中添加一些新的方法和属性



引入aop的schema名称空间

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xmlns:aop="http://www.springframework.org/schema/aop"  
  xsi:schemaLocation="http://www.springframework.org/schema/beans  
    http://www.springframework.org/schema/beans/spring-beans.xsd  
    http://www.springframework.org/schema/aop  
    http://www.springframework.org/schema/aop/spring-aop.xsd">  
  
</beans>
```



通过execution函数定义切点表达式

- 通过execution函数，可以定义切点的方法切入
- 语法：
 - ✓ `execution(<访问修饰符>?<返回类型><方法名>(<参数>)<异常>)`
- 例如
 - ✓ 匹配所有类public方法 `execution(public * *(..))`
 - ✓ 匹配指定包下所有类方法 `execution(* cn.itcast.dao.*(..))` 不包含子包
 - ✓ `execution(* cn.itcast.dao..*(..))` **..*表示包、子孙包下所有类**
 - ✓ 匹配指定类所有方法 `execution(* cn.itcast.service.UserService.*(..))`
 - ✓ 匹配实现特定接口所有类方法
`execution(* cn.itcast.dao.GenericDAO+.*(..))`
 - ✓ 匹配所有save开头的方法 `execution(* save*(..))`



使用advisor定义增强

```
public class MyBeforeAdvice implements MethodBeforeAdvice{
    @Override
    public void before(Method method, Object[] args, Object target)
        throws Throwable {
        System.out.println("before...");
    }
}
```

```
public class HelloService {
    public void sayHello(){
        System.out.println("ok");
    }
}
```

```
<!-- 目标 -->
<bean id="helloService" class="cn.itcast.service.HelloService"></bean>
<!-- 增强 -->
<bean id="mybeforeAdvice" class="cn.itcast.advice.MyBeforeAdvice"></bean>
<!-- aop -->
<aop:config>
    <aop:pointcut expression="execution(* cn.itcast.service.HelloService.*(..))" id="pointcut"/>
    <aop:advisor advice-ref="mybeforeAdvice" pointcut-ref="pointcut"/>
</aop:config>
```



Spring AspectJ增强类型

- AspectJ是一个基于Java语言的AOP框架
- Spring2.0以后新增了对AspectJ切点表达式支持
- @AspectJ 是AspectJ1.5新增功能，通过JDK5注解技术，允许直接在Bean类中定义切面
- 新版本Spring框架，建议使用AspectJ方式来开发AOP
- 使用AspectJ 需要导入Spring AOP和 AspectJ相关jar包
 - ✓ `spring-aop-3.2.0.RELEASE.jar`
 - ✓ `com.springsource.org.aopalliance-1.0.0.jar`
 - ✓ `spring-aspects-3.2.0.RELEASE.jar`
 - ✓ `com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar`



AspectJ提供不同的通知类型

- **Before** 前置通知，相当于**BeforeAdvice**
- **AfterReturning** 后置通知，相当于**AfterReturningAdvice**
- **Around** 环绕通知，相当于**MethodInterceptor**
- **AfterThrowing** 抛出通知，相当于**ThrowAdvice**
- **After** 最终final通知，不管是否异常，该通知都会执行
- **DeclareParents** 引介通知，相当于**IntroductionInterceptor** (不要求掌握)



使用XML配置Aspect -- 前置通知

```
public class MyAspect {
    public void before(JoinPoint joinPoint) {
        System.out.println(joinPoint); // 切点信息
        System.out.println("前置通知");
    }
}
```

```
<!-- 通知所在Bean -->
<bean id="myAspect" class="cn.itcast.aspectJ.MyAspect"></bean>
<!-- 被代理类 -->
<bean id="userDAO" class="cn.itcast.aspectJ.UserDAO"></bean>

<!-- 定义切面 -->
<aop:config proxy-target-class="true">
    <aop:aspect ref="myAspect">
        <!-- 定义切点 -->
        <aop:pointcut id="userDAOPointcut" expression="execution(* cn.itcast.aspectJ.UserDAO.*(..))"/>
        <aop:before method="before" pointcut-ref="userDAOPointcut"/>
    </aop:aspect>
</aop:config>
```




使用XML配置Aspect -- 后置通知

```
public class MyAspect {  
    public void afterReturning(JoinPoint joinPoint, Object returnVal) {  
        System.out.println(joinPoint); // 切点信息  
        System.out.println(returnVal); // 方法返回值  
        System.out.println("后置通知");  
    }  
}
```

```
<!-- 通知所在Bean -->  
<bean id="myAspect" class="cn.itcast.aspectJ.MyAspect"></bean>  
<!-- 被代理类 -->  
<bean id="userDAO" class="cn.itcast.aspectJ.UserDAO"></bean>  
  
<!-- 定义切面 -->  
<aop:config proxy-target-class="true">  
    <aop:aspect ref="myAspect">  
        <!-- 定义切点 -->  
        <aop:pointcut id="userDAOPointcut" expression="execution(* cn.itcast.aspectJ.UserDAO.*(..))"/>  
        <aop:after-returning method="afterReturning" pointcut-ref="userDAOPointcut" returning="returnVal"/>  
    </aop:aspect>  
</aop:config>
```



使用XML配置Aspect -- 环绕通知

```
public class MyAspect {  
    public Object around(ProceedingJoinPoint joinPoint) throws Throwable {  
        System.out.println("方法执行前操作...");  
        Object returnVal = joinPoint.proceed(); // 目标方法执行  
        System.out.println("方法执行后操作...");  
        return returnVal;  
    }  
}
```

```
<!-- 通知所在Bean -->  
<bean id="myAspect" class="cn.itcast.aspectJ.MyAspect"></bean>  
<!-- 被代理类 -->  
<bean id="userDAO" class="cn.itcast.aspectJ.UserDAO"></bean>  
  
<!-- 定义切面 -->  
<aop:config proxy-target-class="true">  
    <aop:aspect ref="myAspect">  
        <!-- 定义切点 -->  
        <aop:pointcut id="userDAOPointcut" expression="execution(* cn.itcast.aspectJ.UserDAO.*(..))"/>  
        <aop:around method="around" pointcut-ref="userDAOPointcut"/>  
    </aop:aspect>  
</aop:config>
```



使用XML配置Aspect -- 抛出通知

```
public class MyAspect {  
    public void afterThrowing(JoinPoint joinPoint, Throwable ex) {  
        System.out.println(joinPoint); // 切点信息  
        System.out.println(ex.getMessage()); // 异常信息  
        System.out.println("发生异常后操作...");  
    }  
}
```

```
<!-- 通知所在Bean -->  
<bean id="myAspect" class="cn.itcast.aspectJ.MyAspect"></bean>  
<!-- 被代理类 -->  
<bean id="userDAO" class="cn.itcast.aspectJ.UserDAO"></bean>  
  
<!-- 定义切面 -->  
<aop:config proxy-target-class="true">  
    <aop:aspect ref="myAspect">  
        <!-- 定义切点 -->  
        <aop:pointcut id="userDAOPointcut" expression="execution(* cn.itcast.aspectJ.UserDAO.*(..))"/>  
        <aop:after-throwing method="afterThrowing" pointcut-ref="userDAOPointcut" throwing="ex"/>  
    </aop:aspect>  
</aop:config>
```



使用XML配置Aspect -- 最终通知

```
public class MyAspect {  
    public void after(JoinPoint joinPoint) {  
        System.out.println(joinPoint); // 切点信息  
        System.out.println("无论是否发生异常，都会执行...");  
    }  
}
```

```
<!-- 通知所在Bean -->  
<bean id="myAspect" class="cn.itcast.aspectJ.MyAspect"></bean>  
<!-- 被代理类 -->  
<bean id="userDAO" class="cn.itcast.aspectJ.UserDAO"></bean>  
  
<!-- 定义切面 -->  
<aop:config proxy-target-class="true">  
    <aop:aspect ref="myAspect">  
        <!-- 定义切点 -->  
        <aop:pointcut id="userDAOPointcut" expression="execution(* cn.itcast.aspectJ.UserDAO.*(..))"/>  
        <aop:after method="after" pointcut-ref="userDAOPointcut"/>  
    </aop:aspect>  
</aop:config>
```



Spring事务管理





Spring 事务管理

- **Spring**事务管理高层抽象主要包括3个接口
- **PlatformTransactionManager**
事务管理器
- **TransactionDefinition**
事务定义信息(隔离、传播、超时、只读)
- **TransactionStatus**
事务具体运行状态

```
PlatformTransactionManager
    ● getTransaction(TransactionDefinition) : TransactionStatus
    ● commit(TransactionStatus) : void
    ● rollback(TransactionStatus) : void
```

```
TransactionDefinition
    ● PROPAGATION_REQUIRED : int
    ● PROPAGATION_SUPPORTS : int
    ● PROPAGATION_MANDATORY : int
    ● PROPAGATION_REQUIRES_NEW : int
    ● PROPAGATION_NOT_SUPPORTED : int
    ● PROPAGATION_NEVER : int
    ● PROPAGATION_NESTED : int
    ● ISOLATION_DEFAULT : int
    ● ISOLATION_READ_UNCOMMITTED : int
    ● ISOLATION_READ_COMMITTED : int
    ● ISOLATION_REPEATABLE_READ : int
    ● ISOLATION_SERIALIZABLE : int
    ● TIMEOUT_DEFAULT : int
    ● getPropagationBehavior() : int
    ● getIsolationLevel() : int
    ● getTimeout() : int
    ● isReadOnly() : boolean
    ● getName() : String
```

```
TransactionStatus
    ● isNewTransaction() : boolean
    ● hasSavepoint() : boolean
    ● setRollbackOnly() : void
    ● isRollbackOnly() : boolean
    ● flush() : void
    ● isCompleted() : boolean
```



事务管理器PlatformTransactionManager

➤ Spring为不同的持久化框架提供了不同PlatformTransactionManager接口实现

事务	说明
<code>org.springframework.jdbc.datasource.DataSourceTransactionManager</code>	使用Spring JDBC或iBatis 进行持久化数据时使用
<code>org.springframework.orm.hibernate3.HibernateTransactionManager</code>	使用Hibernate3.0版本进行持久化数据时使用
<code>org.springframework.orm.jpa.JpaTransactionManager</code>	使用JPA进行持久化时使用
<code>org.springframework.jdo.JdoTransactionManager</code>	当持久化机制是Jdo时使用
<code>org.springframework.transaction.jta.JtaTransactionManager</code>	使用一个JTA实现来管理事务，在一个事务跨越多个资源时必须使用



事务隔离级别（四种）

隔离级别	含义
DEFAULT	使用后端数据库默认的隔离级别(spring中的的选择项)
READ_UNCOMMITTED	允许你读取还未提交的改变了的数据。可能导致脏、幻、不可重复读
READ_COMMITTED	允许在并发事务已经提交后读取。可防止脏读，但幻读和 不可重复读仍可发生
REPEATABLE_READ	对相同字段的多次读取是一致的，除非数据被事务本身改变。可防止脏、不可重复读，但幻读仍可能发生。
SERIALIZABLE	完全服从ACID的隔离级别，确保不发生脏、幻、不可重复读。这在所有的隔离级别中是最慢的，它是典型的通过完全锁定在事务中涉及的数据表来完成的。

脏读:一个事务读取了另一个事务改写但还未提交的数据,如果这些数据被回滚，则读到的数据是无效的。

不可重复读：在同一事务中，多次读取同一数据返回的结果有所不同。换句话说就是，后续读取可以读到另一事务已提交的**更新**数据。相反，“可重复读”在同一事务中多次读取数据时，能够保证所读数据一样，也就是，后续读取不能读到另一事务已提交的更新数据。

幻读：一个事务读取了几行记录后，另一个事务**插入**一些记录，幻读就发生了。再后来的查询中，第一个事务就会发现有些原来没有的记录。



事务隔离级别（四种）

- **DEFAULT** : mysql默认REPEATABLE_READ , oracle 默认 READ_COMMITTED ,每种隔离级别是为了解决事务使用并发导致问题
- **READ_UNCOMMITTED** : 可以发生脏读、不可重复读、虚读
- **READ_COMMITTED** : 避免脏读, 可以发生不可重复读和虚读
- **REPEATABLE_READ** : 避免脏读、不可重复读, 会发送虚读
- **SERIALIZABLE** : 可以阻止所有隔离问题发生



事务传播行为（七种）

事务传播行为类型	说明
PROPAGATION_REQUIRED	支持当前事务，如果不存在 就新建一个
PROPAGATION_SUPPORTS	支持当前事务，如果不存在，就不使用事务
PROPAGATION_MANDATORY	支持当前事务，如果不存在，抛出异常
PROPAGATION_REQUIRES_NEW	如果有事务存在，挂起当前事务，创建一个新的
PROPAGATION_NOT_SUPPORTED	以非事务方式运行，如果有事务存在，挂起当前事务
PROPAGATION_NEVER	以非事务方式运行，如果有事务存在，抛出异常
PROPAGATION_NESTED	如果当前事务存在，则嵌套事务执行 只对DataSourceTransactionManager 起效

事务传播行为

事务的传播行为

企业开发中事务管理通常在业务逻辑层

例一 ATM 取款

表现层

业务逻辑层

数据访问层

```
public void 取款() {  
    返回钱...  
}  
  
public void 打印凭条() {  
    返回凭条...  
}
```

调用打印凭条

问：如果打印凭条过程中发生了异常，取款的事务操作是否进行回滚？
不回滚！

例二 商城系统删除客户信息

表现层

业务逻辑层

数据访问层

```
public void 删除客户方法() {  
}  
  
public void 删除订单方法() {  
}
```

先删除客户订单

问：如果订单已经删除，删除客户时发生异常，是否对订单进行回滚？
回滚！



事务传播行为

- 当一个业务层代码，去调用另一个业务层代码，当被调用方发生异常，调用方代码是提交还是回滚，这就可以通过事务传播行为进行控制。
 - ✓ **REQUIRED**：将调用方和被调用方，放入同一个事务中（传播行为默认值），如果发生异常，整个事务回滚（了解 **SUPPORTS**、**MANDATORY**）-- 删除客户案例
 - ✓ **REQUIRES_NEW**：挂起原来事务，新建一个事务，调用方和被调用方法处于两个不同的事务，当其中一个事务发生异常，对另一个事务无影响 --- 取钱案例（了解 **NOT_SUPPORTED**、**NEVER**）
 - ✓ **NESTED**：嵌套事务（只对DataSourceTransactionManager 起效），就是使用JDBC3.0之后 SavePoint功能，当被调用方出现异常，可以选择将事务回滚到保存点，然后继续操作
- **REQUIRED**、**NESTED** 只有一个事务，**REQUIRES_NEW** 两个事务，**NESTED**可以实现**REQUIRES_NEW** 效果



NESTED 嵌套事务示例

```
Connection conn = null;
try {
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("update person set name='888' where id=1");
    Savepoint savepoint = conn.setSavepoint();
    try{
        conn.createStatement().executeUpdate("update person set name='222' where sid=2");
    }catch(Exception ex){
        conn.rollback(savepoint);
    }
    stmt.executeUpdate("delete from person where id=9");
    conn.commit();
    stmt.close();
} catch (Exception e) {
    conn.rollback();
}finally{
    try {
        if(null!=conn && !conn.isClosed()) conn.close();
    } catch (SQLException e) { e.printStackTrace(); }
}
}
```




Spring 事务管理

➤ Spring 支持两种方式事务管理

✓ 编程式的事务管理

- 在实际应用中很少使用
- 通过TransactionTemplate手动管理事务

✓ 使用XML配置声明式事务

- 开发中推荐使用（代码侵入性最小）
- Spring的声明式事务是通过AOP实现的



转账案例的环境准备

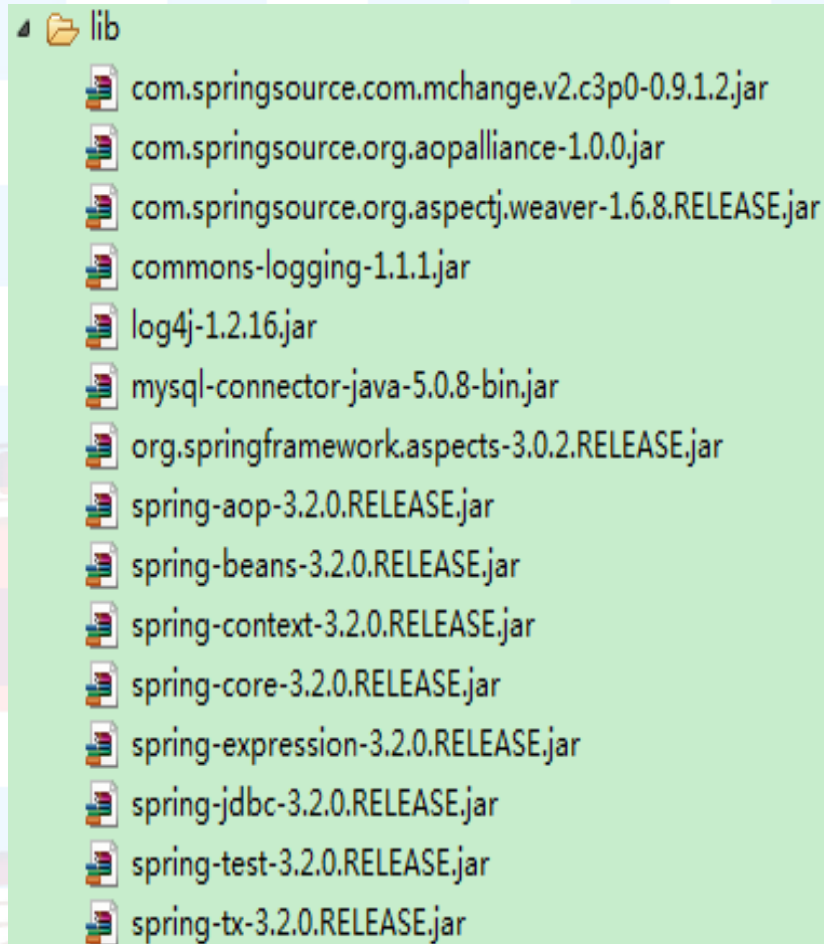
- 创建数据表account

```
CREATE TABLE `account` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `name` varchar(20) NOT NULL,  
  `money` double DEFAULT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;  
INSERT INTO `account` VALUES ('1', 'aaa', '1000');  
INSERT INTO `account` VALUES ('2', 'bbb', '1000');  
INSERT INTO `account` VALUES ('3', 'ccc', '1000');
```



转账案例的环境准备

- 创建web工程
- 导入jar包
- 在src目录编写配置
 - ✓ applicationContext.xml
 - ✓ log4j.properties
 - ✓ jdbc.properties





转账案例的环境准备

➤ 编写DAO注入注入JdbcTemplate

```
public class AccountDAO extends JdbcDaoSupport{  
    public void outMoney(String account, double money) {  
        this.getJdbcTemplate().update(  
            "update account set money = money - ? where name = ?", money, account);  
    }  
    public void inMoney(String account, double money) {  
        this.getJdbcTemplate().update(  
            "update account set money = money + ? where name = ?", money, account);  
    }  
}
```



转账案例的环境准备

➤ 编写Service注入DAO

```
public class AccountService {  
    private AccountDAO accountDAO;  
    // 转账的业务操作  
    public void transfer() {  
        // aaa 向 bbb 转账 200元  
        accountDAO.outMoney("aaa", 200);  
        int d = 1 / 0;  
        accountDAO.inMoney("bbb", 200);  
    }  
    public void setAccountDAO(AccountDAO accountDAO) {  
        this.accountDAO = accountDAO;  
    }  
}
```



转账案例的环境准备

- 配置applicationContext.xml
- `<context:property-placeholder location="classpath:jdbc.properties"/>`
- `<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">`
 - `<!-- ${key} 可以读取properties文件中配置 key对应value -->`
 - `<property name="driverClass" value="${jdbc.driver}"></property>`
 - `<property name="jdbcUrl" value="${jdbc.url}"></property>`
 - `<property name="user" value="${jdbc.username}"></property>`
 - `<property name="password" value="${jdbc.password}"></property>`
- `</bean>`
- `<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">`
 - `<property name="dataSource" ref="dataSource"></property>`
- `</bean>`
- `<bean id="accountService" class="cn.itcast.service.AccountService">`
 - `<property name="accountDAO" ref="accountDAO"></property>`
- `</bean>`
- `<bean id="accountDAO" class="cn.itcast.dao.AccountDAO">`
 - `<property name="jdbcTemplate" ref="jdbcTemplate"></property>`
- `</bean>`



转账案例的环境准备

➤ 编写测试用例

```
@RunWith(SpringJUnit4ClassRunner.class)
```

```
@ContextConfiguration(locations = "classpath:applicationContext.xml")
```

```
public class SpringTest {
```

```
    @Autowired
```

```
    private AccountService accountService;
```

```
    @Test
```

```
    public void demo() {
```

```
        accountService.transfer();
```

```
    }
```

```
}
```



使用XML配置声明式事务 基于tx/aop

➤ 引入aop和tx命名空间

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



使用XML配置声明式事务 基于tx/aop

```
<bean id="accountService" class="cn.itcast.service.AccountService">
    <property name="accountDAO" ref="accountDAO"></property>
</bean>

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

<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="transfer" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>

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



使用XML配置声明式事务 基于tx/aop

➤ 修改测试用例

```
@RunWith(SpringJUnit4ClassRunner.class)
```

```
@ContextConfiguration(locations = "classpath:applicationContext.xml")
```

```
public class SpringTest {
```

```
    @Autowired
```

```
    private AccountService accountService;
```

```
    @Test
```

```
    public void demo() {
```

```
        accountService.transfer();
```

```
    }
```

```
}
```