目录

[1. Java 反射机制 4](#_Toc504036612)

[2. Java注解类的使用 4](#_Toc504036613)

[3. TestNG集成测试 5](#_Toc504036614)

[4. Spring 资源访问 5](#_Toc504036615)

[5. Spring Bean 基于XML配置 6](#_Toc504036616)

[a. Spring 容器结构 6](#_Toc504036617)

[b. Bean XML配置文件基本结构 6](#_Toc504036618)

[c. Bean XML基本配置参数 6](#_Toc504036619)

[d. Bean XML 属性设定 7](#_Toc504036620)

[e. Bean XML 构建参数设定 7](#_Toc504036621)

[f. Bean XML 工厂方法 7](#_Toc504036622)

[g. Bean XML FactoryBean使用方法 8](#_Toc504036623)

[h. Bean XML 特殊数值处理 8](#_Toc504036624)

[i. Bean XML 引用其他Bean数值 9](#_Toc504036625)

[j. Bean XML 设定参数为NULL 9](#_Toc504036626)

[k. Bean XML 引用内部Bean 9](#_Toc504036627)

[l. Bean XML使用List、Set、Map、Properties集合 9](#_Toc504036628)

[m. Bean XML配置简化 10](#_Toc504036629)

[n. Bean XML自动装配-lookup-method方法 10](#_Toc504036630)

[o. Bean XML自动装配-Method-Replacer方法替换 11](#_Toc504036631)

[p. Bean XML 继承、依赖、引用 11](#_Toc504036632)

[q. Bean XML导入其他Bean 11](#_Toc504036633)

[r. Bean XML作用域 11](#_Toc504036634)

[s. Bean XML AOP使用 11](#_Toc504036635)

[6. Spring Bean 基于注解配置 12](#_Toc504036636)

[a. Bean 注解基本结构 12](#_Toc504036637)

[b. Bean注解在不同应用层间的名称 12](#_Toc504036638)

[c. Bean注解扫描 12](#_Toc504036639)

[d. Bean注解自动装配 13](#_Toc504036640)

[e. Bean注解作用域 13](#_Toc504036641)

[f. Bean注解构建销毁配合函数 13](#_Toc504036642)

[7. Spring Bean 基于Java类型配置 13](#_Toc504036643)

[a. Bean Java类型定义基本结构 13](#_Toc504036644)

[8. Spring Bean基于Groovy的配置 13](#_Toc504036645)

[9. BeanFactory,ApplicationContext,Bean上下文套用工具 14](#_Toc504036646)

[a. BeanFactory基本使用方法（使用不方便，速度慢） 14](#_Toc504036647)

[b. ApplicationContext父子类套用方法 14](#_Toc504036648)

[c. BeanFactory中Bean的生命周期 15](#_Toc504036649)

[d. ApplicaitonContext中Bean的生命周期 16](#_Toc504036650)

[10. Spring 容器内部高级方法 16](#_Toc504036651)

[a. 内部工作流程(参考AbstractApplicationContext refresh()方法) 16](#_Toc504036652)

[b. 调用外部配置文件 17](#_Toc504036653)

[c. 调用外部加密配置文件 17](#_Toc504036654)

[d. 引用Bean的属性值 17](#_Toc504036655)

[e. 多国语言特性 17](#_Toc504036656)

[f. 容器事件 18](#_Toc504036657)

[11. Spring AOP 19](#_Toc504036658)

[a. 关键元素 19](#_Toc504036659)

[b. 动态代理技术 19](#_Toc504036660)

[c. AOP alliance动态代理技术 20](#_Toc504036661)

[d. 基于@AspectJ的AOP 25](#_Toc504036662)

[e. 基于Bean XML的AOP 26](#_Toc504036663)

[f. 不同AOP 的实现比较 27](#_Toc504036664)

[g. @AspectJ语法 27](#_Toc504036665)

[12. Spring SpEL（Spring动态表达式语言） 28](#_Toc504036666)

[a. 引用JavaScript 28](#_Toc504036667)

[13. Spring DAO 28](#_Toc504036668)

[a. 传统持久化计划错误获得方法 28](#_Toc504036669)

[b. 不同持久化技术对应的模板及支持类 29](#_Toc504036670)

[c. 常见的SpringDao异常类型 29](#_Toc504036671)

[d. DBCP(Database connection pool)数据源配置方法 29](#_Toc504036672)

[14. Spring 事务处理 30](#_Toc504036673)

[a. 默认事务行为 30](#_Toc504036674)

[b. SQL 92标准事务隔离级别 30](#_Toc504036675)

[c. Spring 执行事务的方法 30](#_Toc504036676)

[d. ThreadLocal 与 InheritableThreadLocal 31](#_Toc504036677)

[e. Spring事务管理 31](#_Toc504036678)

[f. 使用XML配置声明式事务 32](#_Toc504036679)

[g. Spring事务管理的难点 33](#_Toc504036680)

[15. Spring JDBC技术 33](#_Toc504036681)

[a. Spring JDBC访问数据库 33](#_Toc504036682)

[16. Spring 整合其他ORM（对象映射）技术 35](#_Toc504036683)

[a. Hibernate 框架技术 35](#_Toc504036684)

[b. Dao层设计 35](#_Toc504036685)

[17. Spring Cache技术 36](#_Toc504036686)

[a. 缓存过期策略 36](#_Toc504036687)

[b. Spring Cache的使用 36](#_Toc504036688)

[c. 缓存框架技术 36](#_Toc504036689)

[18. Spring 任务调度和异步执行 37](#_Toc504036690)

[a. Quartz任务调度技术 37](#_Toc504036691)

[b. Quartz Schedule对象内部分布 37](#_Toc504036692)

[19. Spring MVC 框架 38](#_Toc504036693)

[a. 工作流程 38](#_Toc504036694)

[b. 工作流程 38](#_Toc504036695)

[c. DispatcherSerlvet 内部工作流程 39](#_Toc504036696)

[d. 控制层控制方法 39](#_Toc504036697)

[e. 请求处理方法签名详细说明 40](#_Toc504036698)

[f. SpringMVC 数据绑定 42](#_Toc504036699)

# Java 反射机制

ClassLoader loader = Thread.currentThread.getContextLoader(); //获得加载类

Class myClass = loader.loadClass(“org.Lex.domain.Car”); 获得类型

……

# Java注解类的使用

注解类的申明:

@Retention(RetentionPolicy.RUNTIME) //申明注解的保留期限

@Target(ElementType.METHOD) //申明可以使用该注解的目标类型

public @interface NeedTest //定义的注解类

{

boolean value() default true; //注解类成员及成员默认值  
}

注解类的使用

Public class ForTest

{

@NeedTest(value=true)

Public void test()

{

System.out.println(“running”);

}  
}

注解类的访问

NeedTest nt = method.getAnnotation(NeetTest.class); //利用反射机制获得注解类

If(nt != null)

{

If(nt.value())

{

System.out.println(“需要测试”);

}

Else

{

System.out.println(“不需要测试”);

}

}

# TestNG集成测试

TestNG 测试时需要测试类需要继承AbstractTestNGSpringContextTests类型

# Spring 资源访问

String filePath = “D:\XXX.txt”; //文件位置

WritableResource res1 = new PathResource(filePath); //可写资源类型

Resource res2 = new ClassPathResouce(“conf/file1.txt”); //只读类型资源

可加载的资源URL类型

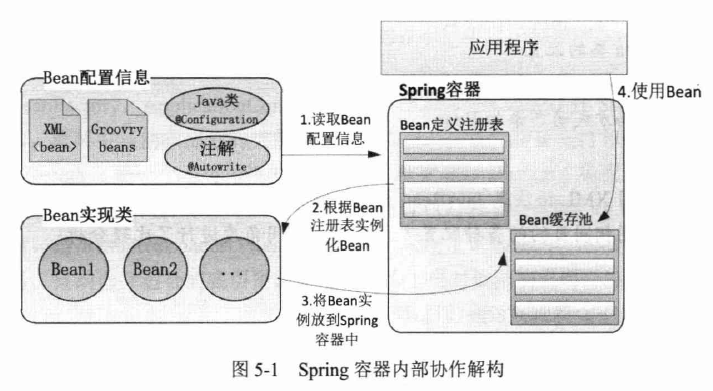
classpath:,file:,http://,ftp://或是没有前缀由ApplicationContext自行识别

识别用类型

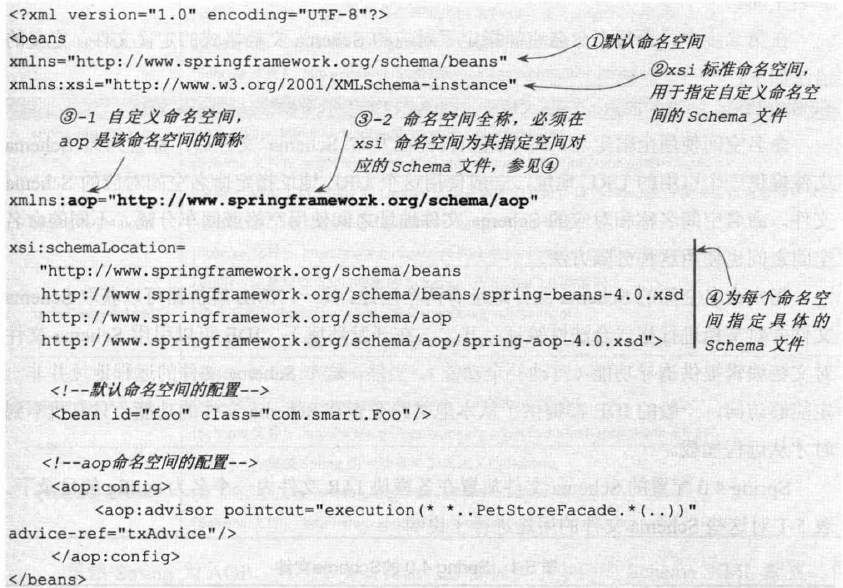
PathMatchingResourcePatternResolver(ResourcePatternResolver的实现类) 可用于解析URL

# Spring Bean 基于XML配置

## Spring 容器结构



## Bean XML配置文件基本结构



## Bean XML基本配置参数

<bean id=”myBeanID”/\*所要生成Bean的名称,同一容器中不能有重名\*/ class=”xxx.xxx.xxx”>

## Bean XML 属性设定

<bean id=”myBeanID” class=”xxx.xxx.xxx”>

<property name=”属性1”><value>”数值1”</value></property>

<property name=”属性2”><value>”数值2”</value></property>

<property name=”属性3”><value>”数值3”</value></property>

</bean>

## Bean XML 构建参数设定

<bean id=”myBeanID”>

<constructor-arg index=”0” /\*第几个参数\*/ type=”java.lang.String” / \*参数类型\*/>

<value>数值</value>

</constructor-arg>

</bean>

## Bean XML 工厂方法

首先使用Java建立实体工厂类型

public class CarFactory{

public Car creatCar(){

Car car = new Car();

return car;

}

}

然后在XML中调用（如不是动态工厂此步骤省略）

<bean id=”carFactory” class=”CarFactory”/>

最后在XML中使用

<bean id=”car5” factory-bean=” carFactory”/\*如不是动态工厂不用设定factory-bean\*/ factory-method=” creatCar”/>

## Bean XML FactoryBean使用方法

首先建立一个实现FactoryBean接口的类型

public class CarFactoryBean implements FactoryBean<Car>/\*FactoryBean支持模板类型\*/ {  
 private String carInfo;  
  
 public String getCarInfo() {  
 return carInfo;  
 }  
  
 public void setCarInfo(String carInfo) {  
 this.carInfo = carInfo;  
 }  
  
 public Car getObject() throws Exception {  
 Car car = new Car();  
 String[] infos = carInfo.split(",");  
  
 car.setId(Integer.valueOf(infos[0]));  
 car.setColor(infos[1]);  
 car.setCarName(infos[2]);  
  
 return car;  
 }  
  
 public Class<Car> getObjectType() {  
 return Car.class;  
 }  
  
 public boolean isSingleton() {  
 return false;  
 }  
}

然后再在Bean XML中调用

<bean id=”car1” class=” CarFactoryBean” p:carInfo=”1,Audi,Black” />

## Bean XML 特殊数值处理

使用HTML 转义字符转换 ，如”<” 则为&lt;

## Bean XML 引用其他Bean数值

一般情况:使用ref替代value

<bean id=”car” class=”Car”/>

<bean id=”boss” class=”Boss”>

<property name=”car”>

<ref bean=”car”></ref>

</property>

</bean>

父子容器情况:在ref替换value 的前提下，将”bean” 改为”parent”

## Bean XML 设定参数为NULL

不可以使用<value></value>的方式，正确方式为<null/>

## Bean XML 引用内部Bean

内嵌式的Bean(基本不使用)

<bean id=”myBeanID”>

<property name=”car”>

<bean class=”xxxx”>

…….

</bean>

</property>

</bean>

## Bean XML使用List、Set、Map、Properties集合

使用 util xmlns命名空间同样也可正常工作

List,Set

<list> /\*如为Set则改为Set\*/

<value>数值1</value>

<value>数值2</value>

<value>数值3</value>

</list>

Map

<map>

<entry>

<key><value>键名称</value></key>

<value>数值</value>

</entry>

</map>

Properties

<property name=”mail”>

<prop key=”键名称”>数值</prop>

</property>

合并集合

在List、Set、Map、Properties加入merge=”true”

## Bean XML配置简化

使用p XML命名空间

## Bean XML自动装配-lookup-method方法

MagicBossInterface是一个拥有getCar方法的接口

<bean id=”car” scope=”prototype”>

<bean id=”magicBoss” class=”MagicBossInterface”>

<lookup-method name=”getCar” bean=”car”/>

</bean>

## Bean XML自动装配-Method-Replacer方法替换

……

## Bean XML 继承、依赖、引用

继承

父类中使用abstract

子类中使用parent

依赖

depends-on

引用

p:carId=”car”

## Bean XML导入其他Bean

<import resource=”classpath:xxx/xxx/xxx”/>

## Bean XML作用域

singleton：在一个容器中只会出现一个

prototype：每次调用都会产生新的实例

request：每个Http请求产生新的实例

session：每个Http会话产生新的实例

globalSession：全局会话共享一个实例,只能用于WebApplicationContext

## Bean XML AOP使用

<bean id=”greetingAdvice” class=”GreetingAdvice”/> /\*增强类\*/

<bean id=”target” class=”NativeWaiter”/> /\*被代理的类型\*/

<bean id=”waiter” class=”org.springframework.aop.framework.ProxyFactoryBean”

p:proxyInterfaces=”WaiterInterface” /\*代理接口\*/

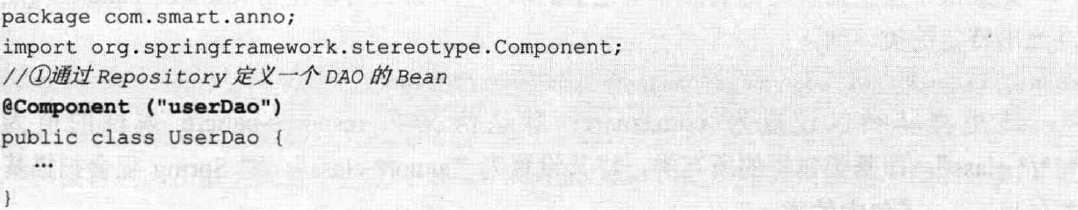
p:InterceptorNames=” greetingAdvice” /\*使用的增强\*/

p:target-ref=”target” /\*代理对象\*/

/>

# Spring Bean 基于注解配置

## Bean 注解基本结构



## Bean注解在不同应用层间的名称

@Component： 无专用定义的一般对象

@Repository：DAO(数据访问对象)层

@Service：Service(服务)层

@Controller：Controller(页面控制)层

这几个注解名称实质上是等效的，名称不同只是为了用于标记/区分不同层间的对象

## Bean注解扫描

在XML中使用context命名空间中的component-scan对象对使用的类型进行扫描

<context:component-scan base-package="org.Lex.configer" /\*命名空间位置\*/ />

扫描特定Pattern的类型

<context:component-scan base-package="org.Lex.configer" resource-pattern="anno/\*.class" />

支持正则表达式过滤功能

<context:component-scan base-package="org.Lex.configer" resource-pattern="anno/\*.class" >  
 <context:include-filter type="regex" expression="xx.xxx1"/>  
 <context:exclude-filter type="regex" expression="xx.xxx2" />  
</context:component-scan>

使用use-default-filters属性表示是否使用默认的过滤机制,如@Compoent,@Repository,@Service,@Controller等

## Bean注解自动装配

使用@Autowired 注解,其参数required表示在无法找到对应类型后是否提示错误

使用@Qualifier注解，表示指定Bean的名称

## Bean注解作用域

使用@Scope 注解，参数singleton表示单例模式，参数prototype表示原型模式

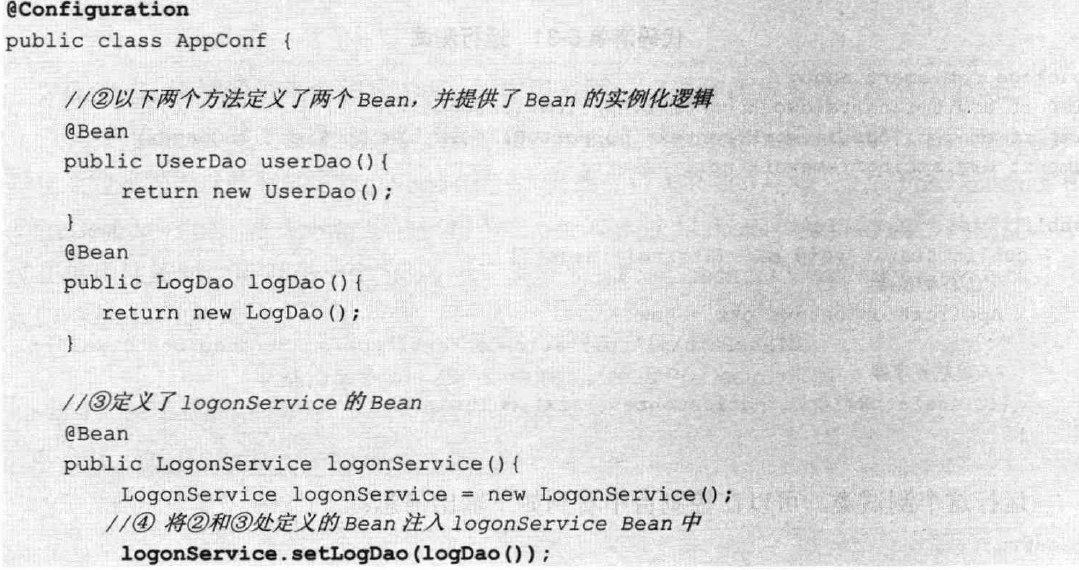
## Bean注解构建销毁配合函数

构建后调用方法：使用@PostConstruct

销毁前调用方法：使用@PreDestroy

# Spring Bean 基于Java类型配置

## Bean Java类型定义基本结构



# Spring Bean基于Groovy的配置

……

# BeanFactory,ApplicationContext,Bean上下文套用工具

## BeanFactory基本使用方法（使用不方便，速度慢）

DefaultListableBeanFactory(BeanFactory的实现类)

XmlBeanDefinitionReader(XML Bean定义读取类)

DefaultListableBeanFactory factory = new DefaultListableBeanFactory();

XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);

reader.loadBeanDefinition(Resource/\*使用spring resource工具获得的Resource类型\*/);

类型 GotClass = factory.getBean(“name”/\*类型名称\*/,name.class);

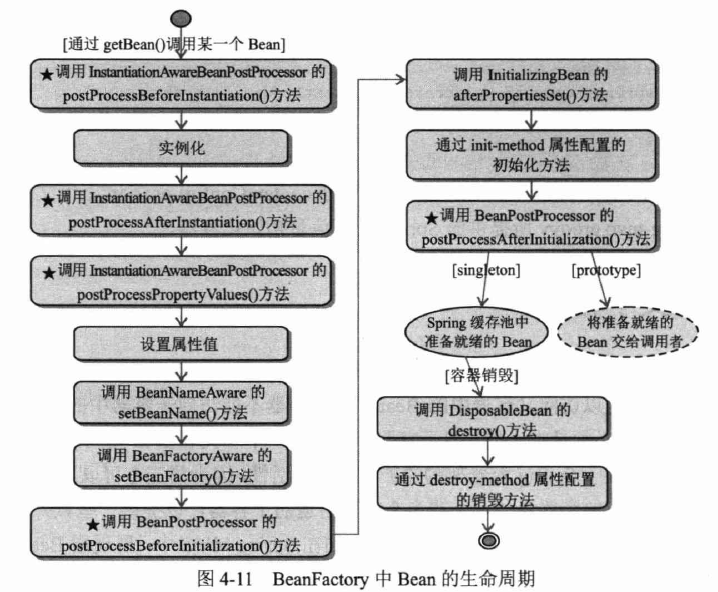
## ApplicationContext父子类套用方法

ClassPathXmlApplicationContext pFactory = new ClassPathXmlApplicationContext(“xxx1.xml”);

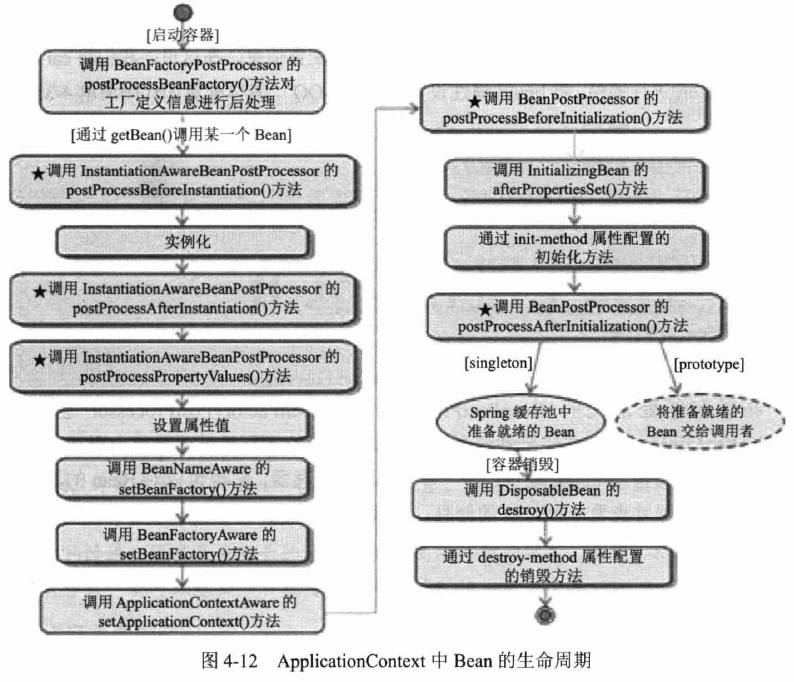
ApplicationContext factory = new ClassPathXmlApplicationContext(“xxx2.xml”,pFactory);

此时xxx1.xml是xxx2.xml的父容器

## BeanFactory中Bean的生命周期



## ApplicaitonContext中Bean的生命周期



# Spring 容器内部高级方法

## 内部工作流程(参考AbstractApplicationContext refresh()方法)

初始化BeanFactory:根据配置文件实例化BeanFactory

使用BeanFactoryPostProcessor接口：调用postProcessBeanFactory接口方法

注册Bean后处理器

初始化消息源

初始化应用上下文时间广播器

注册事件监听器

初始化singleton Bean

发布上下文刷新事件

## 调用外部配置文件

在XML中使用PropertyPlaceholderConfigurer类型配置外部文件，使用location、fileEncoding配置外部文件的路径、编码类型,

使用方法

XML方式：在其他XML中使用${var}方式调用

注解方式：使用@Value(“${var}”)注解方式注入

## 调用外部加密配置文件

扩展PropertyPlaceholderConfigurer并覆盖convertProperties(Properties prop),convertProperty(String propertyName,String propertyValue),convertPropertyValue(String orignalValue)

## 引用Bean的属性值

使用#{var.property}用于引用其他已有Bean的属性值

## 多国语言特性

组成语言特征的两个参数为语言代码(法语以外为小写字母)，地区/国家代码(大写字母)，如中国大陆简体中文应为 zh CN，使用java.util.Locale类建立多语言基础

Locale locale1 = new Locale(“zh”,”CN”); //中国大陆简体中文

Locale locale1 = new Locale(“us”,”EN”); // 美国英语

数字格式化工具(NumberFormat)

Local locale1 = new Locale(“zh”,”CN”);//申明中国大陆地区

NumberFormat currFmt = NumberFormat.getCurrencyInstance(locale1);//申明数字格式

double amt = 123456.78;

System.out.println(currFmt.format(amt));格式化数字输出

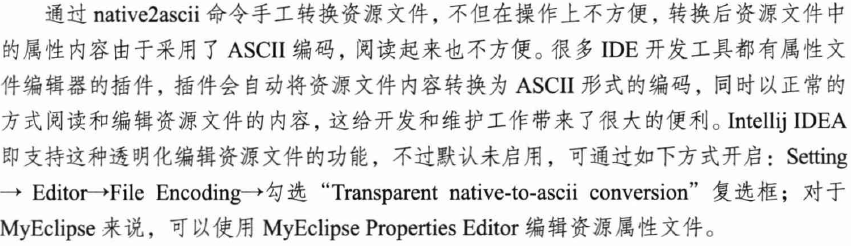
日期格式化工具(DateFormat)

和NumberFormat类似

消息格式化工具(MessageFormat，和NumberFormat、DateFormat功能类似，但是具有pattern功能)

国际化资源文件(ResourceBoundle)

命名规则：<资源名>\_<语言代码>\_<国家/地区代码>.properties，如中国大陆简体中文resource\_zh\_CN.properties



……

## 容器事件

主要对象：ApplicationContextEvent，RequestHandledEvent(WEB专用)

ApplicationContextEvent 事件：

ContextClosedEvent：容器关闭

ContextRefreshEvent：容器刷新

ContextStartedEvent：容器启动

ContextStoppedEvent：容器停止

事件监听器：ApplicationListener接口

# Spring AOP

## 关键元素

连接点：方法运行前后、方法异常前后及方法调用前后

切点：连接点的查询条件

增强：织入的代码

目标对象：增强使用的目标类

引介：为目标对象添加属性和方法以实现接口功能

织入：织入增强的过程

代理：织入完成后产生的类型

切面：=切点+增强

## 动态代理技术

JDK动态代理技术(只能动态代理接口类型,并对所有方法生效,并适合prototype生命周期):

Proxy类，利用InvocationHandler接口定义一个实例，生成目标类的代理对象

InvocationHandler接口，用于实现接口定义横切逻辑，并通过反射调用目标类  
  
public class PerformceHandler implements InvocationHandler {  
  
 private Object target;  
  
 public PerformceHandler(Object target)  
 {  
 this.target = target;  
 }  
  
  
 public Object invoke(Object o, Method method, Object[] objects) throws Throwable {  
  
 XXX.XXX.XXX.XXX织入内容1  
  
 Object obj = method.invoke(method,objects);  
  
 XXX.XXX.XXX.XXX织入内容2  
  
 return obj;  
 }  
}

CGLib动态代理技术(只能动态代理实例类型,并对所有方法生效,并适合singleton生命周期)

public class CglibProxy implements MethodInterceptor {  
  
 private Enhancer enhancer = new Enhancer();  
  
 public Object getProxy(Class clazz)  
 {  
 enhancer.setSuperclass(clazz); //设置需要创建的子类  
 enhancer.setCallback(this);  
 return enhancer.create(); //通过字节码技术创建子类实例  
 }  
   
 public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable { //拦截父类所有的方法  
  
 XXXX.XXX.XXXX 织入增强1  
  
 Object result = methodProxy.invokeSuper(o,objects); //通过代理类调用父类中的方法  
  
 XXXX.XXX.XXXX 织入增强2  
  
 return result;  
  
 }  
}

## AOP alliance动态代理技术

AOP alliance主要增强接口: org.aopalliance.aop.Adivce

前置增强：BeforeAdvice

后置增强：AfterReturningAdivce

环绕增强：MethodInteceptor

异常抛出增强：ThrowsAdvice

引介增强：IntroductionInterceptor

使用方法：

建立增强  
public class GreetingBeforeAdvice implements MethodBeforeAdvice {  
 public void before(Method method, Object[] args, Object target) throws Throwable {  
 System.out.println("hello");  
 }  
}

使用代理

NativeWaiter target = new NativeWaiter();  
GreetingBeforeAdvice advice = new GreetingBeforeAdvice();  
ProxyFactory pf = new ProxyFactory();

pf.setInterfaces(target.getClass().getInterfaces()); //代理接口  
pf.setOptimize(true); //启用优化  
pf.setTarget(target); //设定对象  
pf.addAdvice(advice); //添加增强  
  
NativeWaiter proxy = (NativeWaiter) pf.getProxy();  
proxy.sayHello();  
proxy.sayBye();

对指定方法使用增强(创建切面)：

主要用于描述切点的类型：org.springframework.aop.Pointcut接口描述点,并由ClassFilter和MethodMatcher构成，ClassFilter用于定位到具体类，MethodMatcher用于定位到具体方法

切点类型

静态方法切点：org.springframework.aop.support.StaticMethodMatcherPointcut

动态方法切点：org.springframework.aop.support.DynamicMethodMatcherPointcut

注解切点：org.springframework.aop.support.AnnotationMatcherPointcut

表达式(AspectJ)切点：org.springframework.aop.support.ExpressionPointcut

流程切点：org.springframework.aop.support.ControlFlowPointcut

复合切点：org.springframework.aop.support.ComposablePointcut

切面类型

Advisor：代表一般切面(指类中的所有方法都应用增强)

PointcutAdvisor：代表具有连接点的切面(可以指定对应类型的对应方法)

IntroductionAdvisor：引介切面(可以指定对应类型并增强属性)

静态方法切点完整实例

类型

public class NativeWaiter

{

public void greetTo(String Name){ /\*Do something.\*/ };

}

静态切面完整使用范例

public class ReportPointcutAdvisor extends StaticMethodMatcherPointcutAdvisor{

public matches(Method method,Class<?> targetClass)

{

return “greetTo”.equals(method.getName()); //核对方法名称  
}

public ClassFilter getClassFilter(){ //类名称核对(可选)

return new ClassFilter() {  
 public boolean matches(Class<?> clazz) {  
 return Waiter.class.isAssignableFrom(clazz);  
 }  
};

}

}

增强

Public class ReportAdvice implements MethodBeforeAdvice

{

Public void before(Method method,Object[] args,Object obj) throws Throwable{

Date date = new Date(); //获得当前日期

SimpleDateFormat df = new SimpleDateFormat(“yyyy-MM-dd HH:mm:ss”); //建立日期格式

System.out.println(“Report ” + df.format(date)); //输出日志信息

}  
}

Bean XML 使用定义

<bean id="waiterTarget" class="org.Lex.domain.NativeWaiter"/> /\* 原始类型\*/  
<bean id="greetingBeforeAdvice" class="org.Lex.report.GreetingBeforeAdvice"/> /\*增强\*/  
<bean id="greetingAdvisor" class="org.Lex.report.ReportPointcutAdvisor" p:advice-ref="greetingBeforeAdvice" /> /\*切面\*/  
  
<bean id="parent" abstract="true" class="org.springframework.aop.framework.ProxyFactoryBean"  
 p:interceptorNames="greetingAdvisor" p:proxyTargetClass="true"  
/> /\*父类proxyFactoryBean\*/  
  
<bean id="waiter" parent="parent" p:target-ref="waiterTarget" /> /\*最终代理类型\*/

正则表示式(Regexp)型定义切面法

主要类型：org.springframework.aop.support.RegexpMethodPointcutAdvisor

……

<property name=”pattern”>

<list>

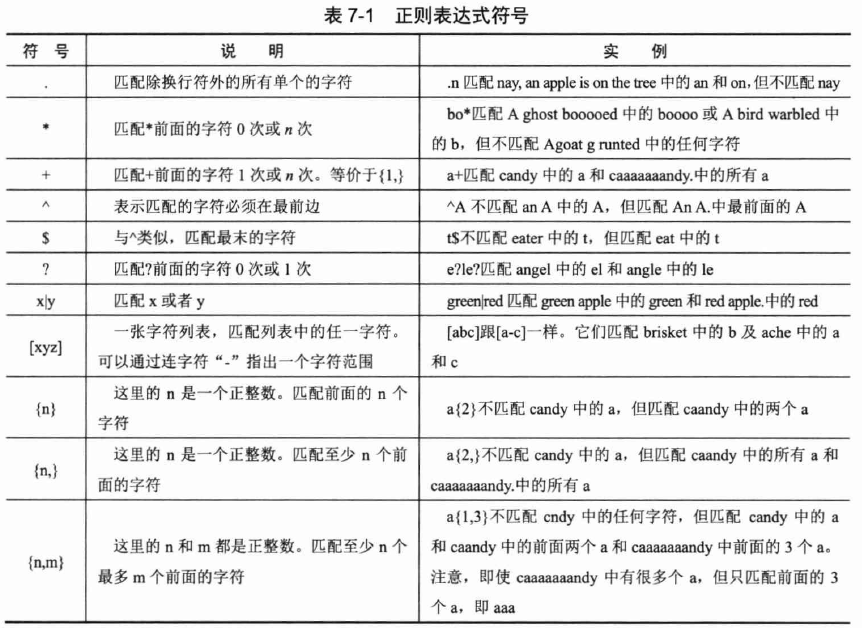
<value>.\*greet\*.</value> //匹配模式串

</list>

</property>

…..

正则表达式符号



动态切面

……

## 基于@AspectJ的AOP

原理类型

Public class NativeWaiter implements Waiter

{

Public void greetTo(String clientName)

{

System.out.println(“hello ” + clientName);

}

Public void serveTo(String clientName)

{

System.out.println(“serveTo : ” + clientName);

}

}

@AspectJ切点与增强类型

@AspectJ

Public class PreGreeting

{

@Before(“execution(\* greetTo(..))”) //定义切点和增强

Public void beforeGreeting() //增强横切逻辑

{

System.out.println(“In doing something”);  
}

}

引介增强使用方法：

@Aspect  
public class EnableSellerAspect {  
  
 @DeclareParents(value = "org.Lex.domain.NativeWaiter", //添加接口的类型

defaultImpl = SmartSeller.class) // 默认接口的实现  
 public Seller seller; //要实现的目标接口  
}

环绕增强使用方法：

@Aspect

public class EnableSellerAspect {  
  
 @Around(“execution (\* greetTo(..))”)

public void joinPointAccess(ProcessingJoinPoint pjt) throws Throwable

{

//方法前

Pjt.proceed(); //目标方法

//方法后  
}

}

## 基于Bean XML的AOP

<aop:config proxy-target-class="true"> //AOP配置组  
 <aop:aspect ref="reportPointcutAspectJ" > //AOP配置增强类  
 <aop:around method="processingJoinPoint" pointcut="execution(\* greetTo(..))"/> //切面定义  
 </aop:aspect>  
  
  
</aop:config>  
  
<bean id="reportPointcutAspectJ" class="org.Lex.report.ReportPointcutAspectJ" /> //AOP类 宣告   
<bean id="waiterTarget" class="org.Lex.domain.NativeWaiter"/> //被增强的类型

## 不同AOP 的实现比较



## @AspectJ语法

@Before(“execution(\* greetTo(..))”)

方法切点函数,execution 用于指定方法，@annotation表示指定对应的注解类

入参切点函数,args表示对应指定的入参的方法，@args表示指定入参的注解类

目标类切点函数,within类名匹配串，target表示指定类，@winthin表示注解类，@target表示注解类

入参通配符

\*:任意字符串，只能用于匹配单个元素

..:任意字符串，可以匹配上下文的多个元素

+:表示按照类型匹配，包含其继承类型

增强类型

@Before 前置增强,相当于BeforeAdvice

@AfterReturning 后置增强,相当于AfterReturningAdvice

@Around环绕增强,相当于MethodInterceptor

@AfterThrowing异常抛出增强，相当于ThrowsAdvice

@After无论有无异常，在方法执行完成后运行增强

@DeclareParents 引介增强，相当于IntroductionInterceptor

# Spring SpEL（Spring动态表达式语言）

## 引用JavaScript

主要类型：ScriptEngineManager，ScriptEngine，(Invocable)ScriptEngine.invokeFunction(“function”,Param1,Param2)

……

# Spring DAO

## 传统持久化计划错误获得方法

获得错误码 (由数字组成)：getErrorCode();

获得错误代理(由字符组成)：getSQLState();

## 不同持久化技术对应的模板及支持类

JDBC:org.springframwork.jdbc.core.JDBCTemplate(模板)，org.springframework.jdbc.core.JDBCSupport(支持)

Hibernate: org.springframwork.jdbc.core. HibernateTemplate(模板)，org.springframework.jdbc.core.HibernateSupport(支持)

JPA: org.springframwork.jdbc.core. JpaDaoTemplate(模板)，org.springframework.jdbc.core.JpaDaoSupport(支持)

JDO: org.springframwork.jdbc.core. JdoDaoTemplate(模板)，org.springframework.jdbc.core.JdoDaoSupport(支持)

## 常见的SpringDao异常类型

CleanupFailureDataAcessException：释放数据资源异常，经常发生于关闭connection时

ConcurrencyFailureException：并发操作时发生的异常

DataAccessResourceFailureException：访问数据资源时产生的异常

DataRetrievalFailureException：获取数据失败

DataIntergrityViolationException：数据一致性异常，如使用了相同的主键，或不存在的外键

InvalidDataAccessApiUsageException：不正确的使用某种持久化技术

InvalidDataAccessResourceException：使用了不正确的方法导致的错误

PermissionDeniedDataAccessException：没有适当的权限

UncategorizedDataAccessException：其他类型的异常

## DBCP(Database connection pool)数据源配置方法

<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close"  
p:driverClassName="com.mysql.jdbc.Driver"  
p:url="jdbc:mysql://127.0.0.1:3309/sampledb"  
p:username="root"  
p:password="123456"  
/>

# Spring 事务处理

## 默认事务行为

在不进行任何设定的前提下，数据库是默认以执行一条指令就是一个事务的方式工作的，故如需要实现原子性，则需要将获得的DBCP连接的自动提交功能关闭setAutoCommit(false)。

## SQL 92标准事务隔离级别



## Spring 执行事务的方法

Try

{

Conn.setAutoCommit(false); //关闭自动提交事务

Conn.setTransactionIsolation(Connection.TRANSACTION\_SERIALZABLE);//设定隔离机制

Statement stmt = conn.createStatement();

Savepoint svpt = conn.setSavepoint(“SavePoint1”); //设定事务保存点,JDBC可支持多个回滚点

Int rows = stmt.executeUpdate(“Insert XXXXXXXX”); // 具体语句-1

Rows = stmt.executeUpdate(“Update XXXXXXX”); // 具体语句-2

Conn.commit();//提交事务

}

Catch(exception)

{

Conn.rollback(); //事务回滚 , 可添加Savepoint参数，以回滚至对应阶段，如svpt。

}

Finally

{

XXXXXXXXXXX; //关闭数据连接等等

}

## ThreadLocal 与 InheritableThreadLocal

ThreadLocal：子线程不可继承父线程

InheritableThreadLocal：子线程可继承父线程

## Spring事务管理

主要要有3个类

TransactionDefinition：建立事务定义

可使用的隔离定义（参考SQL 92）：ISOLATION\_READ\_UNCOMMITED、ISOlATION\_READ\_COMMITED、ISOLATION\_REPEATABLE\_READ、ISOLATION\_SERIALIZABLE、ISOLATION\_DEFAULT(数据库底层默认隔离级别)

可定义的传播方式：PROPAGATION\_REQUIRED、 PROPAGATION\_SUPPORTS、PROPAGATION\_REQUIRES\_NEW、PROPAGATION\_NOT\_SUPPORT、PROPAGATION\_NEVER

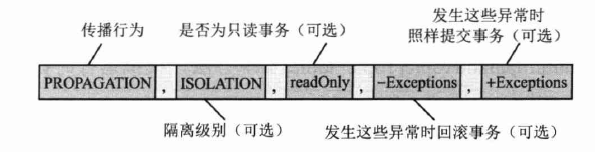
PlatformTransactionManager：通过使用TransactionDefinition建立事务，通过事务池管理事务

TransactionStatus：通过事务池了解事务的工作状态

Hibernate框架的主要连接类型，LocalSessionFactoryBean，DataSource，HibernateTransactionManager

## 使用XML配置声明式事务

使用TransactionProxyFactoryBean时(推荐使用AOP+TX)，Prop（properties）属性设定值顺序



基于aop+tx命名空间方式

<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close" //配置的数据源  
p:driverClassName="com.mysql.jdbc.Driver"  
p:url="jdbc:mysql://127.0.0.1:3309/sampledb"  
p:username="root"  
p:password="123456"  
/>  
  
<bean class="org.springframework.jdbc.datasource.DataSourceTransactionManager" id="transactionManager" p:dataSource-ref="dataSource" /> //事务管理器  
  
<aop:config proxy-target-class="true"> //AOP切点配置  
 <aop:pointcut id="serviceMethod" expression="execution(\* org.Lex.dao.\* (..))" />  
 <aop:advisor pointcut-ref="serviceMethod" advice-ref="xx" />  
</aop:config>  
  
  
  
<tx:advice id="xx" transaction-manager="transactionManager"> //事务增强  
 <tx:attributes>  
 <tx:method name="get\*" no-rollback-for="Exception"/>  
 </tx:attributes>  
</tx:advice>

使用@Transactional 注解功能

<tx:annotation-driven transaction-manager="transactionManager" /> //对添加了@Transactional注解的方法/类型进行事务处理

## Spring事务管理的难点

# Spring JDBC技术

## Spring JDBC访问数据库

主要类型JdbcTemplate,

单次更新:

JdbcTemplate.update(sql,params..,sqlType/\*Type.VARCHAR等\*/) 对应SQL DML Insert Update

JdbcTemplate.query 对应SQL DML Select

JdbcTemplate.execute(sql) 执行所有SQL DML

JdbcTemplate.execute(sql,CallableStatementCallBack<Integer>(){…….}); 执行存储过程，CallableStatementCallBack接口用于设定输入与输出

update回调接口

PreparedStatementSetter() 用于设定参数

PreparedStatementCreator(),KeyHolder(),设定RETURN\_GENERATED\_KEYS后用于获得更新后的主键,getKey()单条主键,getKeys()复合主键,getKeyList()多条主键

query回调接口

单值数据不使用接口

RowCallbackHandler()用于获得数据

RowMapper<T>()用于实现List返回

批量更新：

BatchPreparedStatementSetter 用于设定批量参数

Spring配置DAO分为4个部分

1.定义dataSource

2.定义jdbcTemplate

参数

queryTimeout： 查询超时时间

fetchSize： ResultSet返回行数

maxRows：ResultSet返回最大的行数

ignoreWarnings：是否忽略SQL的警告信息

3.定义抽象Bean

4.配置具体的DAO

定义存储过程

delimiter // 将语句结束符定义为//否则存储过程中的；会无解析

CREATE PROCEDURE SP\_NAME(IN input\_parameter INT,OUT output\_parameter INT)

BEGIN 过程开始

SELECT COUNT(\*) FROM t\_person;

END 过程结束

// 提交定义

delimiter ; 重新将结束语句定义为；

BLOB(二进制大对象) CLOB(字符大对象)类型数据操作

…

不使用数据自带的自增键方法

主要类型：DataFieldMaxValueIncrementer

nextIntValue() 获取下一个主键值,类型为Int

nextLongValue() 获取下一个主键值,类型为Long

nextStringValue() 获取下一个主键值,类型为String

主键方案

应用层主键方案 使用UUID方法

数据库层主键方案 效率要求不高的情况下使用，通过回调KeyHolder获取值

RowSet 对象和ResultSet对象的区别

RowSet对象是一次性返回的(在不设定maxSize的前提下)，ResultSet则是分批返回的

# Spring 整合其他ORM（对象映射）技术

## Hibernate 框架技术

注意：

* Spring 4.3版本开始才支持Hibernate GetFlushMode()方法
* Maven导入包时需要导入spring-tx
* 某些Spring中可能存有旧版本的包也同样会导致无法启动

HibernateCallBack回调接口用于调用Hibernate底层API

处理LOB类型

……

Hibernate事件监听器

……

openSessionInViewFilter 拦截器

……

## Dao层设计

设置Dao基类，将常用的CRUD操作放置于基类中，特殊实现放置在子类中

查询设计，使用OrderQueryParam，或Map作为参数

分页查询（客户端分页，数据端分页，服务端分页），使用OrderQueryParam作业内容查询参数，PageParam作业分页参数

# Spring Cache技术

## 缓存过期策略

LFU：最近最少使用策略优先替换策略

LRU：最久未使用策略替换策略

FIFO：先进先出替换策略

主流的缓存框架：EhCache,Radis

需要缓存化的对象尽量实现serializable接口,不然会有出现序列化异常的现象抛出

## Spring Cache的使用

使用@Cacheable(cacheNames = “users”)注解 添加缓存

@CacheEvict移除缓存

@Caching 用于定义cachable数据缓存对象组

@CacheConfig 可用于定义类全局缓存对象

在XML Bean中定义cacheManager、cacheManager属性ConcurrentMapCacheFactoryBean、cache:annotation-driven

缓存工具-键生成器

Key = “XXX.XXXX”根据值自动生成键

KeyGenerator接口自定义键

Condition，条件化缓存策略

## 缓存框架技术

Ehcache

<bean id="cacheManager" class="org.springframework.cache.ehcache.EhCacheCacheManager"/> //cacheManager  
<bean id="ehcache" p:configLocation="classpath:ehcache.xml" class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"/> //定义的cacheFactoryBean

ehcache.xml //配置文件

<ehcache>

<cache name="users"

maxElementsInMemory="1000" />

</ehcache>

# Spring 任务调度和异步执行

## Quartz任务调度技术

Job （无状态）接口：定义需要执行的任务,可并发执行

StatefulJob（有状态）接口：定义需要执行的任务，不可并发执行

JobDetial接口：Job接口的实现类，用于描述Job名称，描述，关联的监听器等

Trigger类型：触发器，表示Job的触发规则

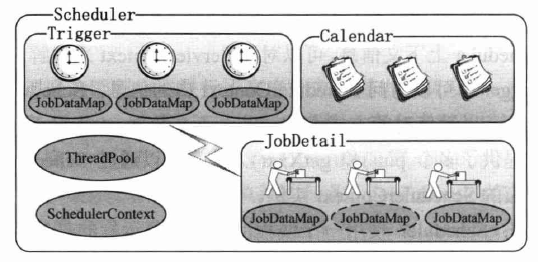
Calendar：日期

Scheduler：Trigger与JobDetial注册表

JobDataMap：任务关系映射

ThreadPool：Scheduler运行使用的线程池

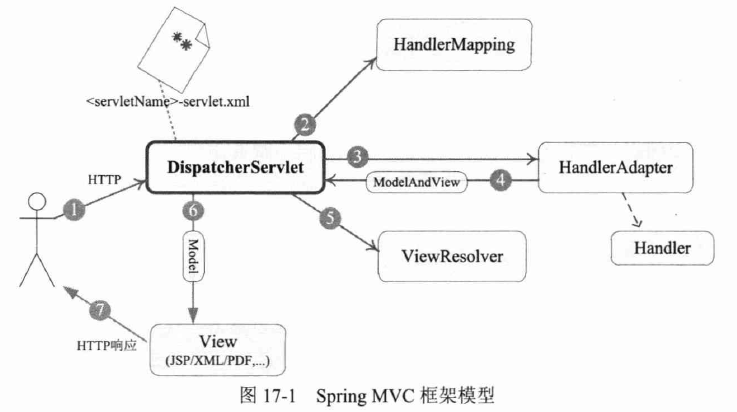
## Quartz Schedule对象内部分布



…….

# Spring MVC 框架

## 工作流程



## 工作流程

定义MVC 入口Servlet

<context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath:Test-Context.xml</param-value> //上下文配置  
</context-param>  
  
<listener>  
 <listener-class>  
 org.springframework.web.context.ContextLoaderListener //上下文加载侦听器  
 </listener-class>  
</listener>  
  
<servlet>  
 <servlet-name>Test</servlet-name> //MVC入口Servlet   
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <load-on-startup>1</load-on-startup>  
</servlet>  
  
<servlet-mapping>  
 <servlet-name>Test</servlet-name> //Servlet 映射  
 <url-pattern>\*.html</url-pattern>   
</servlet-mapping>

定义MVC入口Servlet的容器，容器名称为”Servlet名称-servlet.xml”

在xml bean中需要定义InternalResourceViewResolver对象，定义视图解析器，将viewClass定义为JstlView,并定义视图文件的具体位置(JSP文件的位置)

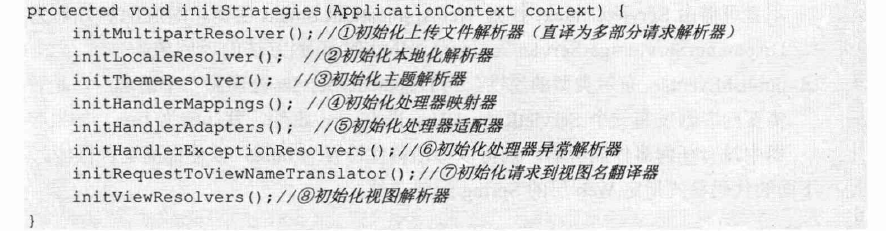
注：MVC与业务层、持久层并不是属于同一个容器中(他们是父子容器的结构，MVC可以访问业务层和持久层、但业务层和持久层不能访问MVC的对象)

注：在Idea 中设定Web模块，并设定根目录的位置

SpringMVC导向静态资源的方法

将dispatcherServlet.xml中定义<mvc:resources mapping=”xxx/xxx”/\*mapping表示映射的请求位置\*/ location=”xxxx”/\*实际位置\*/>

## DispatcherSerlvet 内部工作流程

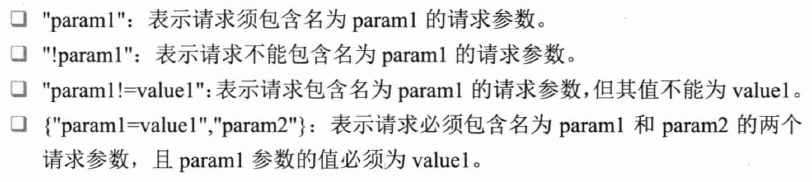


## 控制层控制方法

@RequestMapping(path=”/xxx/xxx” /\*可以直接填路径不用path\*/) 表示请求路径

@RequestMapping(value=”xxxx” /\*请求URL\*/,method=”xxxx” /\*请求方法http get、post等\*/,params=”userId”/\*请求参数\*/,header=”context-type=text/\*” /\*报文头映射\*/)

@PathVariable(“xxx”) 可将路径变量绑定入参



@RequestMapping(“/{userId}”)

Public ModelAndView showDetial(@PathVariable(“userId”) String userId) //路径userId绑定入参userId

{

…

…

}

使用矩阵变量绑定参数(@MatrixVariable)

多个变量使用”;”分割

/books;author=Tom;year=2016

一个变量对应多个值使用”,”分割

Author = smart1,smart2,smart3

注：如需要使用空值不报错功能时，需要使用@MatrixVariable(required = false)注解

## 请求处理方法签名详细说明

@RequestParam(value=””/\*参数名称\*/,required=false/\*是否强制需要\*/,defaultValue=”123”/\*默认值\*/)

@CookieValue(value=”sessionId”/\*cookie名称\*/,required=false/\*是否强制需要\*/)绑定请求中的Cookie值

@RequestHeader(“Accept-Encoding”)绑定Http请求表头

@RequestMapping(path=”/handle14”) 通过表单对象绑定

Public String handle14(User user) // 请求地址为/handle14?username=123&userid=156

{

……  
}

使用图片等资源作为输出流

@RequestMapping(“/handle31”)

Pubic void handle31(OutputStream outputStream) throws IOException

{

Resource imgResource = new ClassPathResource(“/img.jpeg”); //获得图片流

FileCopyUtils.copy(imgResource.getInputStream(),os); //使用FileCopyUtils复制图片至输出流  
}

使用HttpMessageConverter<T>（Http对象转换器）接口

主要类型org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter

转换类

org.springframework.http.converter.\*

系统默认已定义了RequestMappingHandlerAdapter，若自行定义则会自动覆盖系统默认的RequestMappingHandlerAdapter,系统默认转换类型是(StringHttpMessageConverter、ByteArrayHttpMessageConverter、SourceHttpMessageConverter 、AllEncompassingFormHttpMessageConverter)

使用HttpEntity<T>作为入参,ResponseEntity<T>作为返回数，可用于实现HttpMessageConverter<T>的使用

实现请求/返回XML、JSON所使用的HttpMessageConverter类型

XML:MarshallingHttpMessageConverter、Jaxb2RootElementHttpMessageConverter

JSON:MappingJackson2HttpMessageConverter

XML转换时需要申明使用Xstream技术的marshaller和unmarsharller,并在marshaller对象中加入annotedClass属性对象

使用RestController和AsyncRestTemplate

@RestController = @Controller + @ResponseBody

AsyncRestTemplate

Spring作为客户端时可通过Callback实现异步访问

处理模型数据(ModelAndView)

使用@ModelAttribute(“xxxx”) 将对象放入到返回对象中

如果入参为ModelMap或是Map类型对象，则Spring会自动返回给请求方

@SessionAttributes表示保存对象进入会话中

返回String 中为”redirect:\*”(重导向)或是”forward:\*”(转发)会进行对应的转发

## SpringMVC 数据绑定

数据绑定的工作流程



定义ConversionServiceFactoryBean

<bean id="conversionService2" class="org.springframework.context.support.ConversionServiceFactoryBean" >  
 <property name="converters">  
 <util:list>  
 <bean class="org.Lex.util.myConverter" /> //转换用的类型  
 </util:list>  
 </property>  
</bean>

转换用的类型可以实现以下三种接口中的其中一个

Converter<S,T>

GenericConverter

ConverterFactory

配置定义转换服务

<mvc:annotation-driven conversion-service="conversionService2"/>

使用@InitBinder装配自定义编辑器

功能与ConversionService一致，

@InitBinder

Public void initBinder(WebDataBinder binder)

{

Binder.registerCustomEditor(User.class,new UserEditor() /\*UserEditor实现PropertyEditor\*/)；  
}

数据格式化

主要接口Formatter<T> extends Printer<T> ,Parser<T>

P618