**包的简单介绍：**

Core和beans是spring框架最基础的部分，包括了ioc（把创建对象的任务交给程序）和di

Context在core和beans的基础上提供对象的访问方式，核心接口是application

Expression 主要是对象的获取以及其他对属性的操作

Aop 提供了面向切面编程的实现。

**Bean中引入其他配置文件：**

<import resource="services.xml"/>

<import resource="resources/messageSource.xml"/>

<import resource="/resources/themeSource.xml"/>

**Resource的路径相对于当前配置文件的路径：**（推荐使用classpath:/config/services.xml或jvm系统属性）

（由于这里的路径是相对的，因此建议 **不使用反斜杠**）

messageSource.xml和themeSource.xml必须在当前定义导入的文件路径下的resources路径下。

使用一个相对"../"路径引用父目录中的配置是允许的，但是不推荐这么做。如果这么做就产生了一个当前应用外的引用依赖。特别不推荐在使用"classpath:"路径的时候，在运行的时候解析选择“最近”的classpath跟路径，然后在找父目录。Classpath配置的更改可能会导致选择一个不同的、错误的目录。

通常情况下，你可以使用完全限定的资源位置来代替相对路径，例如："file:C:/config/services.xml"或"classpath:/config/services.xml"。但是请注意，你的应用可能和一个特定的绝对路径耦合了。通常更合适的方式是通过间接的方式来使用绝对路径，例如通过"${…}"占位符，在运行时解析JVM的系统属性。

**配置元数据：**编写配置文件或使用注解配置

**实例化容器：**applicationcontext是beanfactory的子接口；

ApplicationContext context =new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});

**使用容器：**使用 T getBean(name, requiredType) 方法，取得这些beans的实例。

*// 取得配置的实例*

PetStoreService service = context.getBean("petStore", PetStoreService.**class**);

**Bean的定义：**

| **属性名** | **说明…** |
| --- | --- |
| class | [Section 5.3.2, “实例化bean”](http://spring.cndocs.ml/beans.html#beans-factory-class) |
| name | [Section 5.3.1, “bean的命名”](http://spring.cndocs.ml/beans.html#beans-beanname) |
| Scope（单例,多例） | [Section 5.5, “Bean作用域”](http://spring.cndocs.ml/beans.html#beans-factory-scopes) |
| constructor arguments | [Section 5.4.1, “依赖注入”](http://spring.cndocs.ml/beans.html#beans-factory-collaborators) |
| properties | [Section 5.4.1, “依赖注入”](http://spring.cndocs.ml/beans.html#beans-factory-collaborators) |
| autowiring mode | [Section 5.4.5, “自动装配协作者”](http://spring.cndocs.ml/beans.html#beans-factory-autowire) |
| lazy-initialization mode | [Section 5.4.4, “延迟初始化bean”](http://spring.cndocs.ml/beans.html#beans-factory-lazy-init) |
| initialization method | [the section called “初始化回调函数”](http://spring.cndocs.ml/beans.html#beans-factory-lifecycle-initializingbean) |
| destruction method | [the section called “析构回调函数”](http://spring.cndocs.ml/beans.html#beans-factory-lifecycle-disposablebean) |

**Bean实例化：**

通过构造函数实例化；

使用静态工厂方法实例化；在类中包含相应的静态实例化方法

使用实例工厂方法实例化；指定对应工厂类

**依赖注入和控制反转：**

**依赖注入是控制反转另外的名字，**传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象

即应用程序依赖容器，容器为应用程序注入对象的外部资源

**依赖注入：**

构造器注入:不可变的一些

Setter注入：可选的依赖，可以重新注入或配置（注意：和属性没关系，只和set方法中set的字符串有关如：jdbc中jdbcurl属性的注入方法为setUrl（），则注入时property的name为url）

循环依赖：A类需要通过构造器注入B类的实例，并且B类又需要通过构造器注入A类的实例，可通过setter注入的方式，但是bean A 和bean B之间的循环依赖将会导致其中一个bean在被完全初始化的之前被注入到另一个bean里（先有鸡先有蛋的问题）。

在<list/>, <set/>, <map/>, 和<props/>元素中，你可以设置值和参数分别对应Java的集合类型List, Set, Map, 和 Properties

<bean id="moreComplexObject" class="example.ComplexObject">

*<!-- results in a setAdminEmails(java.util.Properties) call -->*

<property name="adminEmails">

<props>

<prop key="administrator">administrator@example.org</prop>

<prop key="support">support@example.org</prop>

<prop key="development">development@example.org</prop>

</props>

</property>

*<!-- results in a setSomeList(java.util.List) call -->*

<property name="someList">

<list>

<value>a list element followed by a reference</value>

<ref bean="myDataSource" />

</list>

</property>

*<!-- results in a setSomeMap(java.util.Map) call -->*

<property name="someMap">

<map>

<entry key="an entry" value="just some string"/>

<entry key ="a ref" value-ref="myDataSource"/>

</map>

</property>

*<!-- results in a setSomeSet(java.util.Set) call -->*

<property name="someSet">

<set>

<value>just some string</value>

<ref bean="myDataSource" />

</set>

</property>

</bean>

value="" 和<null/> 空字符串和null

**自动装配：@autowired和使用配置文件中autowire mode去指定相应bean或全局指定**

**默认情况下不开启自动装配，即bean的引用（property）必须使用ref定义**

**开启后会根据bean属性名或者类型自动在容器中寻找，并进行装配**

**好处**

当对象发生变化时自动装配可以更新配置。比如如果你需要给一个类添加依赖，那么这个依赖可以被自动满足而不需要你去修改配置。 因此自动依赖在开发时尤其有用，当系统趋于稳定时改为显式装配。

**Bean实例化回调函数：**

在XML配置元数据的情况下，使用 init-method 属性去指定方法名，并且该方法无参数签名。 例如，下面的定义：

<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>

这种方式避免了spring和应用代码的耦合性。

**BeanPostProcessor定制bean的后置处理器：**

如果你想在Spring容器完成bean的实例化、配置和其它的初始化后执行一些自定义逻辑，你可以插入一个或多个的BeanPostProcessor实现。（可以代理bean或者其他bean的处理操作）.

将beanpostprocessor实现类放入容器中即可：

<bean class="scripting.InstantiationTracingBeanPostProcessor"/>

**BeanfactoryPostprocessor定制beanfactory（容器）的后置处理器:**

BeanFactoryPostProcessor可以对bean的定义（配置元数据）进行处理。也就是说，Spring IoC容器允许BeanFactoryPostProcessor在容器实际实例化任何其它的bean之前读取配置元数据，并有可能修改它。(也就是设置beanfactory，可以定制容器中bean的元数据)

Spring中自带了许多beanfactory的后置处理器，如PropertyResourceConfigurer和PropertyPlaceholderConfigurer以及BeanNameAutoProxyCreator，它们用于对bean进行事务性包装或者使用其他的proxy进行包装（加载静态配置文件）。BeanFactoryPostProcessor也能被用来添加自定义属性编辑器。

### ApplicationContext是beanfactory的子接口

实例化：可以使用ContextLoader机制有两种方式，ContextLoaderListener 和ContextLoaderServlet

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>

</context-param>

<listener>

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

</listener>

*<!-- or use the ContextLoaderServlet instead of the above listener*

<servlet>

<servlet-name>context</servlet-name>

<servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>

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

</servlet>

-->

监听器首先检查contextConfigLocation参数，如果它不存在，它将使用/WEB-INF/applicationContext.xml作为默认值。如果已存在，它将使用分隔符（逗号、冒号或空格）将字符串分解成应用上下文件位置路径。可以支持ant-风格的路径模式，如/WEB-INF/\*Context.xml(WEB-INF文件夹下所有以"Context.xml"结尾的文件)。或者/WEB-INF/\*\*/\*Context.xml(WEB-INF文件夹及子文件夹下的以"Context.xml"结尾的文件)

**注解方式配置：**

@Autowired

应用于setter方法、应用于字段和构造函数

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Autowired

private MovieCatalog movieCatalog;

private CustomerPreferenceDao customerPreferenceDao;

@Autowired

public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {

this.customerPreferenceDao = customerPreferenceDao;

}

@Autowired

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

}

@resource和@autowired是功能是一样的只是@resource是javaee注解，而@autowird是spring注解

### Ant-风格路径：

**？代表可以匹配任意一个字符**

**\* 代表可以匹配零个或多个任意字符**

**\*\* 代表可以匹配零个或多个目录**

**Spring中数据绑定和校验（主要在controller层使用）：**

Spring 提供了我们称作DataBinder（该类可指定校验器进行数据校验）的对象来处理数据绑定，所谓的数据绑定就是将用户的输入 自动的绑定到我们的领域模型(或者说任意用来处理用户输入的对象)。Spring 的Validator和 DataBinder构成了validation包，这个包主要被Spring MVC框架使用，但绝不限于只能在该 框架使用。

**必须使用@Valid标注我们需要校验的参数**user，否则Spring不会对它进行校验

**自定义Controller层数据绑定和校验：**

指定controller范围内使用

1. 实现validator校验器接口（注意两个方法supports():校验是否支持该实体类校验，validate方法进行校验处理。
2. 使用@initBinder注解initBinder方法（该方法可进行数据绑定，设置校验器）
3. 使用@valid指定校验的参数，并在方法中传入errors类或子类bindingresult来查看是否在校验时有错误信息。

全局范围内使用

在**mvc:annotation-driven 中指定validator**

**Spel的应用**

1. Bean属性值动态引用：<property name="name" value="#{user.name}" />
2. 访问静态方法，静态变量：<property name="name" value="#{T(类全名).静态方法}" />
3. 操作一个表达式的值：  
   eq(==),lt(<),le(<=),gt(>),ge(>=)。  
   逻辑表达式：  
   and,or,not或!。  
   条件运算符：使用三元运算符

<property value="#{sonSelector.selecSOng()=='Jingle Bells'?piano:saxophone}" />

一个常见的三元运算符的使用场景是检查一个值是否为null。

<property name="song" value="#{kenny.song != null ? kenny.song : 'Greensleeves'}" />

虽然以上配置可以正常工作，但这里kenny.song的引用重复了两次。SpEL提供了三元运算符的变体来简化表达式：

<property name="song" value="#{kenny.song ?: 'Greensleeves'}" />

**Spring事务管理**

特点：

不同的 api（JTA JDBC, Hibernate, JPA,JDO ）使用一致的编程模型

声明式事务管理：配置文件声明事务

全局事务：可以混合使用事务资源,传统关系型数据库以及消息队列

本地事务：针对特定资源的, 例如一个和JDBC链接绑定在一起的事务. 本地事务有可能用起来更简单, 但是有一个 明显的弊端:他们不能运行在跨多种资源的事务中.

**声明式事务管理：仅支持本地事务管理**

|  |
| --- |
| <!-- 这是我们希望使之支持事务的服务层对象 -->  <bean id="fooService" class="x.y.service.DefaultFooService"/>  <!-- 事务化配置(请看下面的<aop:advisor/>)  tx:advice 设置事务的语义  -->  <tx:advice id="txAdvice" transaction-manager="txManager">  <!-- 事务语义... -->  <tx:attributes>  <!-- 所有用'get'开头的方法都是只读的 -->  <tx:method name="get\*" read-only="true"/>  <!-- 其他的方法使用默认的事务配置(看下面) -->  <tx:method name="\*"/>  </tx:attributes>  </tx:advice>  <!-- 使得上面的事务配置对FooService接口的所有操作有效 -->  <aop:config>  <aop:pointcut id="fooServiceOperation" expression="execution(\* x.y.service.FooService.\*(..))"/>  <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>  </aop:config>  <!-- 不要忘了DataSource -->  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">  <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>  <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>  <property name="username" value="scott"/>  <property name="password" value="tiger"/>  </bean>  <!-- 同样的, 也不要忘了PlatformTransactionManager -->  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource"/>  </bean> |

多数据源配置事务问题：

Service层只能处理当前数据源的事务处理，如果存在多个数据源就要使用分布式事务处理例如：JTA（java 事务api）

在spring和mybatis整合的项目中，如果不涉及事务管理，可以配置多数据源来实现对不同数据库资源mapper的支持。

**注解实现声明式事务：**

使用@transaction注解

|  |
| --- |
| <!-- 这就是我们想要使之支持事务的对象 -->  <bean id="fooService" class="x.y.service.DefaultFooService"/>  <!-- 要配置数据源-->  <!-- 使使用注解配置的事务行为生效 -->  <tx:annotation-driven transaction-manager="txManager"/><!-- 仍然需要一个PlatformTransactionManager -->  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <!-- (这个需要的对象是在其他地方定义的) -->  <property name="dataSource" ref="dataSource"/>  </bean> |

同样面对多数据源配置问题：

参考下面博客链接

博客：http://blog.sina.com.cn/s/blog\_8f61307b0100ynfb.html