# Spring框架

## 一、技术说明（技术介绍，技术优势以及发展史等）

## 1.1、什么是spring

* Spring是分层的JavaSE/EE full-stack 轻量级开源框架

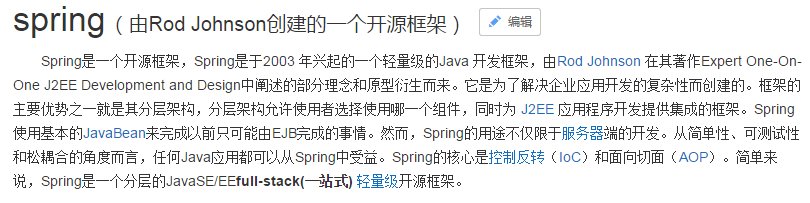
分层：三层体系结构，为每一个层都提供解决方案

web层：struts2、spring-mvc

service层：spring

dao层：hibernate、mybatis、jdbcTemplate(spring)

轻量级：使用时占用资源少，依赖程序少。比较：EJB



## 1.2、spring由来

Expert One-to-One J2EE Design and Development ，介绍EJB，使用，特点

Expert One-to-One J2EE Development without EJB ，不使用EJB，spring思想

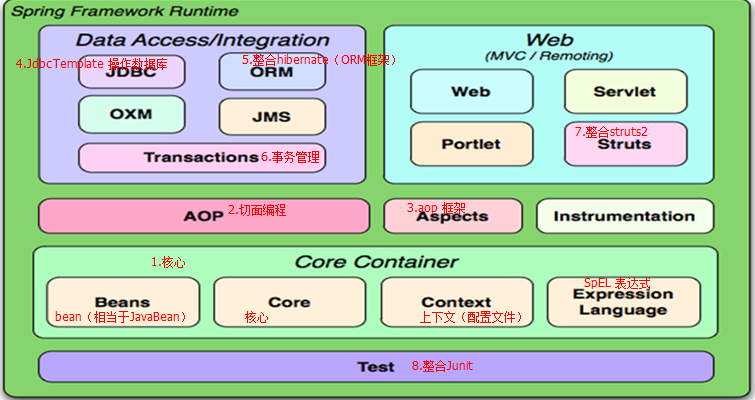
## 1.3、spring核心

* 以IoC（Inverse of Control 反转控制）和AOP（Aspect Oriented Programming 面向切面编程为内核）

## 1.4、spring优点

|  |
| --- |
| * 方便解耦，简化开发 （易扩展，易维护）   + 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 |

## 1.5、spring体系结构



* **spring 核心功能：beans、core、context、expression**

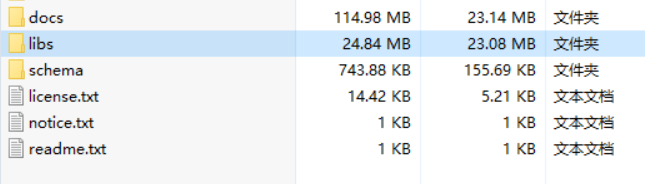
## 二、环境搭建（技术开发环境）

### 2.1、获取 Spring framework jar 包

## 1、spring官网下载

从官网下载spring 最新的相关jar包，官网download地址 http://www.springsource.org/springcommunity-download

下载完成后会发现三个目录，命名很明确。

Docs 目录相关文档。包括一份 API 和一份各种 spring 的使用说明(reference)，reference 提供了 HTML.PDF 版本，非常详细。 

## 2.spring包的核心包

搭建第一个用到 spring 依赖注册的程序

直接用 eclipse 建立一个 JAVA 项目

然后添加 spring 的 jar 包引入

spring-core-3.2.0.M1.jar 核心依赖 jar 包

spring-context-3.2.0.M1.jar Spring 容器包

spring-beans-3.2.0.M1.jar Spring beans 的管理包

spring-asm-3.2.0.M1.jar Spring

注：和 hibernate 一起用时这个 JAR 会冲突，解决方法删掉它就是了

|  |
| --- |
| org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'sessionFactory' defined in ServletContext resource [/WEB-INF/classes/applicationContext.xml]: Invocation of init method failed; nested exception is java.lang.NoSuchMethodError: org.objectweb.asm.ClassVisitor.visit(IILjava/lang/String;Ljava/lang/String;[Ljava/lang/String;Ljava/lang/String;)  Caused by:  java.lang.NoSuchMethodError: org.objectweb.asm.ClassVisitor.visit(IILjava/lang/String;Ljava/lang/String;[Ljava/lang/String;Ljava/lang/String;) |

spring-expression-3.2.0.M1.jar

除此之外，还有需要一个 Apache common 的 JAR 包

注：如果忘记添加会 commons-logging-1.1.1.jar 报错

|  |
| --- |
| Exception in thread "main" java.lang.NoClassDefFoundError: org/apache/commons/logging/LogFactory |

commons-logging-1.1.1.jar 日志记录

## 3. 配置 XML

Spring 的最大的作用就是提供 bean 的管理功能，在 spring 中 bean 的管理是通过 XML 实现的，要 用此功能，需要把 bean 配置到 spring 的 xml

### 1. 新建立一个 xml.名字任意，如 applicationContext.xml,或者 text.xml 都可以

### 2. 添加 xml 头定义

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <beans xmlns=http://www.springframework.org/schema/beans ①  xsi:schemaLocation="http://www.springframework.org/schema/beans ② http://www.springframework.org/schema/beans/spring-beans-3.2.xsd ③  ">  <bean> </bean> </beans> |

1. Xmlns(XML NameSpace) 声明命名空间，建议是用网址做命名空间，但并不会去访问改网址， 仅仅是 namespace 和 xsd（xsd 是 spring 这个 xml 的 schema 文件，里面定义配置内容）里 声明的 targetNamespace 保持一致 . 注：这里命名空间是改不了的，其实是在代码中也写死了，可以打开 spring-beans3.2.0.M1-sources.jar 包的 org\springframework\beans\factory\xml\BeanDefinitionParserDelegate.java 查看对 http://www.springframework.org/schema/beans这个namespace的定义。

② schemaLoacation .用于绑定命名空间的 schema 文件，通常是用 URL 值对，中间用空格隔 开,前面 URL 是命名空间，后面 URL 为 schema 的文件地址

③ xsd 的存放地址，如果没有声明,eclipse 会去网上下载.

在创建 xml 时，在 eclipse 编辑 xml 配置没有提示。

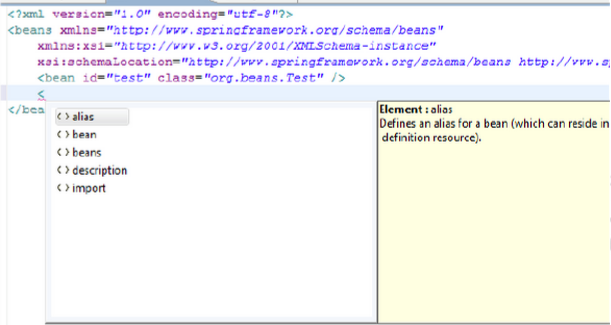
可以对 eclipse 中进行 schema 文件的添加

具体是

WindowsPreferences ,搜索 XML catalog，添加 schema 文件。

Spring 的 schema 文件在下载包里有，找到 spring-3.2.0.M1\schema\beans\spring-beans-3.2.xsd

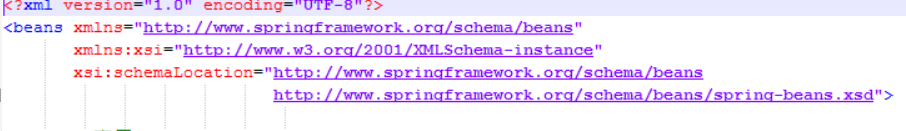
然后添加，编写 spring 配置文件就能有提示了



## 4. 依赖注入

### 4.1 、spring注入的简单案例（入门级）

新建一个 class 用于注入，



|  |
| --- |
| package org.beans;  public class Test { public void say(){  System.out.println("welcome"); } } |

把类通过 xml 配置注入

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?> <beans xmlns="http://www.springframework.org/schema/beans" xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"> <bean id="test" class="org.beans.Test" /> </beans> |

测试:

|  |
| --- |
| package org.beans;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class testBeans {  public static void main(String[] args) {  ApplicationContext ctx = new ClassPathXmlApplicationContext("test.xml");  Test test=(Test) ctx.getBean("test");  test.say(); }  } |

### 4.2、spring框架为我们提供了三种注入方式，分别是set注入，构造方法注入，接口注入。接口注入不作要求，下面介绍前两种方式。

#### 1、set注入

  采用属性的set方法进行初始化，就成为set注入。

##### 1）给普通字符类型赋值。

**[java]** [view plain](http://blog.csdn.net/lishuangzhe7047/article/details/20740835) [copy](http://blog.csdn.net/lishuangzhe7047/article/details/20740835)

[print?](http://blog.csdn.net/lishuangzhe7047/article/details/20740835)[在CODE上查看代码片](https://code.csdn.net/snippets/223910)

1. **public** **class** User{
2. privateString username;
4. publicString getUsername() {
5. returnusername;
6. }
7. publicvoid setUsername(String username) {
8. **this**.username= username;
9. }
10. }

   我们只需要提供属性的set方法，然后去属性文件中去配置好让框架能够找到applicationContext.xml文件的beans标签。标签beans中添加bean标签， 指定id，class值，id值不做要求，class值为对象所在的完整路径。bean标签再添加property 标签，要求，name值与User类中对应的属性名称一致。value值就是我们要给User类中的username属性赋的值。

1. **<bean** id="userAction"class="com.lsz.spring.action.User" **>**
2. **<span** style="white-space:pre"**>**  **</span><property** name="username" value="admin"**></property>**
3. **</bean>**

##### 2）给对象赋值

 同样提供对象的set方法

1. **public** **class** User{
2. **private** UserService userservice;
3. **public** UserServicegetUserservice() {
4. returnuser;
5. }
6. **public** **void** setUserservice(UserService userservice){
7. **this**.userservice= userservice;
8. }
9. }

   配置文件中要增加UserService的bean标签声明及User对象对UserService引用。

1. <!--对象的声明-->
2. **<bean** id="userService" class="com.lsz.spring.service.UserService"**></bean>**
4. **<bean** id="userAction"class="com.lsz.spring.action.User" **>**
5. **<property** name="userservice" ref="userService"**></property>**
6. **</bean>**

  这样配置，框架就会将UserService对象注入到User类中。

##### 3）给list集合赋值

 同样提供set方法

1. **public** **class** User{
2. privateList<String> username;
3. publicList<String> getUsername() {
4. returnusername;
5. }
6. publicvoid setUsername(List<String> username) {
7. **this**.username= username;
8. }
9. }
10. **<bean** id="userAction"class="com.lsz.spring.action.User" **>**
11. **<propertyname**propertyname="username"**>**
12. **<list>**
13. **<value>**zhang,san**</value>**
14. **<value>**lisi**</value>**
15. **<value>**wangwu**</value>**
16. **</list>**
17. **</property>**
18. **</bean>**

##### 4）给属性文件中的字段赋值

1. **public** **class** User{
2. privateProperties props ;
3. publicProperties getProps() {
4. returnprops;
5. }
6. publicvoid setProps(Properties props) {
7. **this**.props= props;
8. }
9. }
10. **<bean>**
11. **<propertyname**propertyname="props"**>**
12. **<props>**
13. **<propkey**propkey="url"**>**jdbc:oracle:thin:@localhost:orl**</prop>**
14. **<propkey**propkey="driverName"**>**oracle.jdbc.driver.OracleDriver**</prop>**
15. **<propkey**propkey="username"**>**scott**</prop>**
16. **<propkey**propkey="password"**>**tiger**</prop>**
17. **</props>**
18. **</property>**
19. **</bean>**

<prop>标签中的key值是.properties属性文件中的名称

**注意：**

**无论给什么赋值，配置文件中<property>标签的name属性值一定是和对象中名称一致。**

#### 2、构造方法注入

##### **1）构造方法一个参数**

1. **public** **class** User{
2. privateString usercode;
3. publicUser(String usercode) {
4. **this**.usercode=usercode;
5. }
6. }
7. **<bean** id="userAction"class="com.lsz.spring.action.User"**>**
8. **<**constructor-argvalue**constructor-argvalue**="admin"**></constructor-arg>**
9. **</bean>**

##### 2）构造函数有两个参数时

  当参数为非字符串类型时，在配置文件中需要制定类型，如果不指定类型一律按照字符串类型赋值。

  当参数类型不一致时，框架是按照字符串的类型进行查找的，因此需要在配置文件中制定是参数的位置

1. **<**constructor-argvalue**constructor-argvalue**="admin"index="0"**></constructor-arg>**
2. **<**constructor-argvalue**constructor-argvalue**="23" type="int"index="1"**></constructor-arg>**

  这样制定，就是构造函数中，第一个参数为string类型，第二个参数为int类型

## 5、Spring的[junit单元测试](http://blog.csdn.net/lishuangzhe7047/article/details/20741499)

单元测试以set注入的第一个实例为测试对象。进行单元测试。

### 1，拷贝jar包

 junit-3.8.2.jar(4.x主要增加注解应用)

### 2，写业务类

1. **public** **class** User{
2. privateString username;
4. publicString getUsername() {
5. returnusername;
6. }
7. publicvoid setUsername(String username) {
8. **this**.username= username;
9. }
11. //添加方法
12. publicString login() **throws** Exception{
13. **if**("admin".equals(username){
14. **return**"success";
15. }**else**{
16. **return**"error";
17. }
18. }
19. }

### 3，定义测试类

  测试类最好单独建立项目，或者单独定义文件夹存储，需要继承junit.framework.TestCase

### 4，增加测试方法

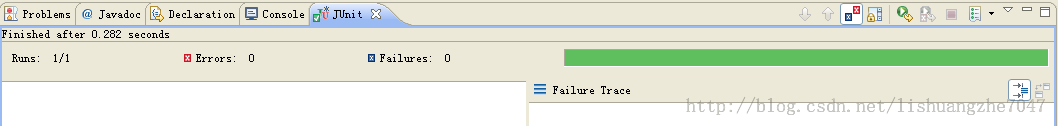
  测试方法必须是public,不应该有返回值，方法名必须以test开头，无参数

 测试方法是有执行先后顺序，按照方法的定义先后顺序

 多个测试方法对同一个业务方法进行测试，一般每个逻辑分支结构都有测试到。

1. **public** **class** TestUserextends TestCase{
2. publicvoid testUser\_Success() **throws** Exception{
3. //准备数据
4. Useraction = **new** User();
5. action.setUsername("admin");
7. //调用被测试方法
8. Stringresult = action.login();
10. //判断测试是否通过
11. assertEquals("success",result);
12. }
13. }

  运行程序，如果测试成功会出现如下图所示的结果



 如果运行失败，有方法没有通过测试，那么就会显示出在哪个方法出错了。上图中绿色的条会变成红色的。

### 5，测试类的生命周期方法

2. //用来进行初始化操作
3. @Override
4. protectedvoid setUp() **throws** Exception {
5. System.out.println("setUp...");
6. }
8. //用来做销毁操作
9. @Override
10. protectedvoid tearDown() **throws** Exception {
11. System.out.println("tearDown...");
12. }

setUp方法会在每一个测试方法前执行一次。tearDown方法会在每一个测试方法后执行一次

## 6、Spring对注解(Annotation)处理源码分析1——扫描和读取Bean定义

### 1、Spring　IoC容器对于类级别的注解和类内部的注解分以下两种处理策略：

(1).类级别的注解：如@Component、@Repository、@Controller、@Service以及JavaEE6的@ManagedBean和@Named注解，都是添加在类上面的类级别注解，Spring容器根据注解的过滤规则扫描读取注解Bean定义类，并将其注册到Spring IoC容器中。

(2).类内部的注解：如@Autowire、@Value、@Resource以及EJB和WebService相关的注解等，都是添加在类内部的字段或者方法上的类内部注解，SpringIoC容器通过Bean后置注解处理器解析Bean内部的注解。

下面将根据这两种处理策略，分别分析Spring处理注解相关的源码。

### 2.AnnotationConfigApplicationContext对注解Bean初始化：

Spring中，管理注解Bean定义的容器有两个：AnnotationConfigApplicationContext和 AnnotationConfigWebApplicationContex。这两个类是专门处理Spring注解方式配置的容器，直接依赖于注解作为容器配置信息来源的IoC容器。 AnnotationConfigWebApplicationContext是AnnotationConfigApplicationContext的web版本，两者的用法以及对注解的处理方式几乎没有什么差别，因此本文将以AnnotationConfigApplicationContext为例进行讲解。

AnnotationConfigApplicationContext的源码如下：

1. **public** **class** AnnotationConfigApplicationContext **extends** GenericApplicationContext {
2. //创建一个读取注解的Bean定义读取器，并将其设置到容器中
3. **private** **final** AnnotatedBeanDefinitionReader reader = **new** AnnotatedBeanDefinitionReader(**this**);
4. //创建一个扫描指定类路径中注解Bean定义的扫描器，并将其设置到容器中
5. **private** **final** ClassPathBeanDefinitionScanner scanner = **new** ClassPathBeanDefinitionScanner(**this**);
6. //默认构造函数，初始化一个空容器，容器不包含任何 Bean 信息，需要在稍后通过调用其register() //方法注册配置类，并调用refresh()方法刷新容器，触发容器对注解Bean的载入、解析和注册过程
7. **public** AnnotationConfigApplicationContext() {
8. }
9. //最常用的构造函数，通过将涉及到的配置类传递给该构造函数，以实现将相应配置类中的Bean
10. //自动注册到容器中
11. **public** AnnotationConfigApplicationContext(Class<?>... annotatedClasses) {
12. register(annotatedClasses);
13. refresh();
14. }
15. //该构造函数会自动扫描以给定的包及其子包下的所有类，并自动识别所有的Spring Bean，将其
16. //注册到容器中
17. **public** AnnotationConfigApplicationContext(String... basePackages) {
18. scan(basePackages);
19. refresh();
20. }
21. //为容器的注解Bean读取器和注解Bean扫描器设置Bean名称产生器
22. **public** **void** setBeanNameGenerator(BeanNameGenerator beanNameGenerator) {
23. **this**.reader.setBeanNameGenerator(beanNameGenerator);
24. **this**.scanner.setBeanNameGenerator(beanNameGenerator);
25. }
26. //为容器的注解Bean读取器和注解Bean扫描器设置作用范围元信息解析器
27. **public** **void** setScopeMetadataResolver(ScopeMetadataResolver scopeMetadataResolver) {
28. **this**.reader.setScopeMetadataResolver(scopeMetadataResolver);
29. **this**.scanner.setScopeMetadataResolver(scopeMetadataResolver);
30. }
31. //为容器注册一个要被处理的注解Bean，新注册的Bean，必须手动调用容器的
32. //refresh()方法刷新容器，触发容器对新注册的Bean的处理
33. **public** **void** register(Class<?>... annotatedClasses) {
34. **this**.reader.register(annotatedClasses);
35. }
36. //扫描指定包路径及其子包下的注解类，为了使新添加的类被处理，必须手动调用
37. //refresh()方法刷新容器
38. **public** **void** scan(String... basePackages) {
39. **this**.scanner.scan(basePackages);
40. }
41. }

通过对AnnotationConfigApplicationContext的源码分析，我们了解到Spring对注解的处理分为两种方式：

#### (1).直接将注解Bean注册到容器中：

可以在初始化容器时注册；也可以在容器创建之后手动调用注册方法向容器注册，然后通过手动刷新容器，使得容器对注册的注解Bean进行处理。

#### (2).通过扫描指定的包及其子包下的所有类：

在初始化注解容器时指定要自动扫描的路径，如果容器创建以后向给定路径动态添加了注解Bean，则需要手动调用容器扫描的方法，然后手动刷新容器，使得容器对所注册的Bean进行处理。

接下来，将会对两种处理方式详细分析其实现过程。

### 3.AnnotationConfigApplicationContext注册注解Bean：

当创建注解处理容器时，如果传入的初始参数是具体的注解Bean定义类时，注解容器读取并注册。

#### **(1).AnnotationConfigApplicationContext通过调用注解Bean定义读取器AnnotatedBeanDefinitionReader的register方法向容器注册指定的注解Bean**

注解Bean定义读取器向容器注册注解Bean的源码如下：

1. //注册多个注解Bean定义类
2. **public** **void** register(Class<?>... annotatedClasses) {
3. **for** (Class<?> annotatedClass : annotatedClasses) {
4. registerBean(annotatedClass);
5. }
6. }
7. //注册一个注解Bean定义类
8. **public** **void** registerBean(Class<?> annotatedClass) {
9. registerBean(annotatedClass, **null**, (Class<? **extends** Annotation>[]) **null**);
10. }
11. //Bean定义读取器注册注解Bean定义的入口方法
12. **public** **void** registerBean(Class<?> annotatedClass, Class<? **extends** Annotation>... qualifiers) {
13. registerBean(annotatedClass, **null**, qualifiers);
14. }
15. //Bean定义读取器向容器注册注解Bean定义类
16. **public** **void** registerBean(Class<?> annotatedClass, String name, Class<? **extends** Annotation>... qualifiers) {
17. //根据指定的注解Bean定义类，创建Spring容器中对注解Bean的封装的数据结构
18. AnnotatedGenericBeanDefinition abd = **new** AnnotatedGenericBeanDefinition(annotatedClass);
19. //解析注解Bean定义的作用域，若@Scope("prototype")，则Bean为原型类型；
20. //若@Scope("singleton")，则Bean为单态类型
21. ScopeMetadata scopeMetadata = **this**.scopeMetadataResolver.resolveScopeMetadata(abd);
22. //为注解Bean定义设置作用域
23. abd.setScope(scopeMetadata.getScopeName());
24. //为注解Bean定义生成Bean名称
25. String beanName = (name != **null** ? name : **this**.beanNameGenerator.generateBeanName(abd, **this**.registry));
26. //处理注解Bean定义中的通用注解
27. AnnotationConfigUtils.processCommonDefinitionAnnotations(abd);
28. //如果在向容器注册注解Bean定义时，使用了额外的限定符注解，则解析限定符注解。
29. //主要是配置的关于autowiring自动依赖注入装配的限定条件，即@Qualifier
30. //注解，Spring自动依赖注入装配默认是按类型装配，如果使用@Qualifier则按名称
31. **if** (qualifiers != **null**) {
32. **for** (Class<? **extends** Annotation> qualifier : qualifiers) {
33. //如果配置了@Primary注解，设置该Bean为autowiring自动依赖注入装//配时的首选
34. **if** (Primary.**class**.equals(qualifier)) {
35. abd.setPrimary(**true**);
36. }
37. //如果配置了@Lazy注解，则设置该Bean为非延迟初始化，如果没有配置，
38. //则该Bean为预实例化
39. **else** **if** (Lazy.**class**.equals(qualifier)) {
40. abd.setLazyInit(**true**);
41. }
42. //如果使用了除@Primary和@Lazy以外的其他注解，则为该Bean添加一
43. //个autowiring自动依赖注入装配限定符，该Bean在进autowiring
44. //自动依赖注入装配时，根据名称装配限定符指定的Bean
45. **else** {
46. abd.addQualifier(**new** AutowireCandidateQualifier(qualifier));
47. }
48. }
49. }
50. //创建一个指定Bean名称的Bean定义对象，封装注解Bean定义类数据
51. BeanDefinitionHolder definitionHolder = **new** BeanDefinitionHolder(abd, beanName);
52. //根据注解Bean定义类中配置的作用域，创建相应的代理对象
53. definitionHolder = AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder, **this**.registry);
54. //向IoC容器注册注解Bean类定义对象   BeanDefinitionReaderUtils.registerBeanDefinition(definitionHolder, this.registry);
55. }

##### （1）从上面的源码我们可以看出，注册注解Bean定义类的基本步骤：

###### a，需要使用注解元数据解析器解析注解Bean中关于作用域的配置。

###### b，使用AnnotationConfigUtils的processCommonDefinitionAnnotations方法处理注解Bean定义类中通用的注解。

###### c，使用AnnotationConfigUtils的applyScopedProxyMode方法创建对于作用域的代理对象。

###### d，通过BeanDefinitionReaderUtils向容器注册Bean。

下面我们继续分析这3步的具体实现过程

#### (2).AnnotationScopeMetadataResolver解析作用域元数据：

AnnotationScopeMetadataResolver通过processCommonDefinitionAnnotations方法解析注解Bean定义类的作用域元信息，即判断注册的Bean是原生类型(prototype)还是单态(singleton)类型

其源码如下：

1. //解析注解Bean定义类中的作用域元信息
2. **public** ScopeMetadata resolveScopeMetadata(BeanDefinition definition) {
3. ScopeMetadata metadata = **new** ScopeMetadata();
4. **if** (definition **instanceof** AnnotatedBeanDefinition) {
5. AnnotatedBeanDefinition annDef = (AnnotatedBeanDefinition) definition;
6. //从注解Bean定义类的属性中查找属性为”Scope”的值，即@Scope注解的值
7. // annDef.getMetadata().getAnnotationAttributes方法将Bean
8. //中所有的注解和注解的值存放在一个map集合中
9. Map<String, Object> attributes =
10. annDef.getMetadata().getAnnotationAttributes(**this**.scopeAnnotationType.getName());
11. //将获取到的@Scope注解的值设置到要返回的对象中
12. **if** (attributes != **null**) {
13. metadata.setScopeName((String) attributes.get("value"));
14. //获取@Scope注解中的proxyMode属性值，在创建代理对象时会用到
15. ScopedProxyMode proxyMode = (ScopedProxyMode) attributes.get("proxyMode");
16. //如果@Scope的proxyMode属性值为null、DEFAULT或者NO
17. **if** (proxyMode == **null** || proxyMode == ScopedProxyMode.DEFAULT) {
18. //设置proxyMode为NO
19. proxyMode = **this**.defaultProxyMode;
20. }
21. //为返回的元数据设置proxyMode
22. metadata.setScopedProxyMode(proxyMode);
23. }
24. }
25. //返回解析的作用域元信息对象
26. **return** metadata;
27. }

上述代码中的annDef.getMetadata().getAnnotationAttributes方法就是获取对象中指定类型的注解的值。

#### (3).AnnotationConfigUtils处理注解Bean定义类中的通用注解：

AnnotationConfigUtils类的processCommonDefinitionAnnotations在向容器注册Bean之前，首先对注解Bean定义类中的通用Spring注解进行处理

源码如下：

1. //处理Bean定义中通用注解
2. **static** **void** processCommonDefinitionAnnotations(AnnotatedBeanDefinition abd) {
3. //如果Bean定义中有@Primary注解，则为该Bean设置为autowiring自动依赖注入//装配的首选对象
4. **if** (abd.getMetadata().isAnnotated(Primary.**class**.getName())) {
5. abd.setPrimary(**true**);
6. }
7. //如果Bean定义中有@Lazy注解，则将该Bean预实例化属性设置为@lazy注解的值
8. **if** (abd.getMetadata().isAnnotated(Lazy.**class**.getName())) {
9. Boolean value = (Boolean) abd.getMetadata().getAnnotationAttributes(Lazy.**class**.getName()).get("value");
10. abd.setLazyInit(value);
11. }
12. //如果Bean定义中有@ DependsOn注解，则为该Bean设置所依赖的Bean名称，
13. //容器将确保在实例化该Bean之前首先实例化所依赖的Bean
14. **if** (abd.getMetadata().isAnnotated(DependsOn.**class**.getName())) {
15. String[] value = (String[]) abd.getMetadata().getAnnotationAttributes(DependsOn.**class**.getName()).get("value");
16. abd.setDependsOn(value);
17. }
18. }

#### (4).AnnotationConfigUtils根据注解Bean定义类中配置的作用域为其应用相应的代理策略：

AnnotationConfigUtils类的applyScopedProxyMode方法根据注解Bean定义类中配置的作用域@Scope注解的值，为Bean定义应用相应的代理模式，主要是在Spring面向切面编程(AOP)中使用

源码如下：

1. //根据作用域为Bean应用引用的代码模式
2. **static** BeanDefinitionHolder applyScopedProxyMode(
3. ScopeMetadata metadata, BeanDefinitionHolder definition, BeanDefinitionRegistry registry) {
4. //获取注解Bean定义类中@Scope注解的proxyMode属性值
5. ScopedProxyMode scopedProxyMode = metadata.getScopedProxyMode();
6. //如果配置的@Scope注解的proxyMode属性值为NO，则不应用代理模式
7. **if** (scopedProxyMode.equals(ScopedProxyMode.NO)) {
8. **return** definition;
9. }
10. //获取配置的@Scope注解的proxyMode属性值，如果为TARGET\_CLASS，则返
11. //回true，如果为INTERFACES，则返回false
12. **boolean** proxyTargetClass = scopedProxyMode.equals(ScopedProxyMode.TARGET\_CLASS);
13. //为注册的Bean创建相应模式的代理对象
14. **return** ScopedProxyCreator.createScopedProxy(definition, registry, proxyTargetClass);
15. }

这段为Bean引用创建相应模式的代理，如果在Spring面向切面编程(AOP)中涉及到再详细分析，这里不做深入的分析。

#### (5).BeanDefinitionReaderUtils向容器注册Bean：

BeanDefinitionReaderUtils向容器注册载入的Bean我们在第4篇博客中已经分析过，主要是校验Bean定义，然后将Bean添加到容器中一个管理Bean定义的HashMap中，这里就不做分析。

### 4.AnnotationConfigApplicationContext扫描指定包及其子包下的注解Bean：

当创建注解处理容器时，如果传入的初始参数是注解Bean定义类所在的包时，注解容器将扫描给定的包及其子包，将扫描到的注解Bean定义载入并注册。

#### (1).Spring中常用的注解：

##### a.Component注解：

1. @Target(ElementType.TYPE)
2. @Retention(RetentionPolicy.RUNTIME)
3. @Documented
4. **public** **@interface** Component {
5. String value() **default** "";
6. }

##### b.Service注解：

1. @Target({ElementType.TYPE})
2. @Retention(RetentionPolicy.RUNTIME)
3. @Documented
4. @Component
5. **public** **@interface** Service {
6. String value() **default** "";
7. }

##### c.Controller注解：

1. @Target({ElementType.TYPE})
2. @Retention(RetentionPolicy.RUNTIME)
3. @Documented
4. @Component
5. **public** **@interface** Controller {
6. String value() **default** "";
7. }

##### d.Repository注解：

1. @Target({ElementType.TYPE})
2. @Retention(RetentionPolicy.RUNTIME)
3. @Documented
4. @Component
5. **public** **@interface** Repository {
6. String value() **default** "";
7. }

通过分析Spring这4个常用的注解源码，我们看到：@Service、@Controller和@Repository注解都添加了一个@Component注解，因此他们都属于@Component

注解。

#### (2).ClassPathBeanDefinitionScanner扫描给定的包及其子包：

AnnotationConfigApplicationContext通过调用类路径Bean定义扫描器ClassPathBeanDefinitionScanner扫描给定包及其子包下的所有类，主要源码如下：

1. **public** **class** ClassPathBeanDefinitionScanner **extends** ClassPathScanningCandidateComponentProvider {
2. //创建一个类路径Bean定义扫描器
3. **public** ClassPathBeanDefinitionScanner(BeanDefinitionRegistry registry) {
4. **this**(registry, **true**);
5. }
6. //为容器创建一个类路径Bean定义扫描器，并指定是否使用默认的扫描过滤规则。
7. //即Spring默认扫描配置：@Component、@Repository、@Service、@Controller
8. //注解的Bean，同时也支持JavaEE6的@ManagedBean和JSR-330的@Named注解
9. **public** ClassPathBeanDefinitionScanner(BeanDefinitionRegistry registry, **boolean** useDefaultFilters) {
10. //调用父类ClassPathScanningCandidateComponentProvider构造方法设置过滤规则
11. **super**(useDefaultFilters);
12. Assert.notNull(registry, "BeanDefinitionRegistry must not be null");
13. //为容器设置加载Bean定义的注册器
14. **this**.registry = registry;
15. //如果注册器是资源加载器，则为容器设置资源加载器
16. **if** (**this**.registry **instanceof** ResourceLoader) {
17. setResourceLoader((ResourceLoader) **this**.registry);
18. }
19. }
20. //调用类路径Bean定义扫描器入口方法
21. **public** **int** scan(String... basePackages) {
22. //获取容器中已经注册的Bean个数
23. **int** beanCountAtScanStart = **this**.registry.getBeanDefinitionCount();
24. //启动扫描器扫描给定包
25. doScan(basePackages);
26. //注册注解配置(Annotation config)处理器
27. **if** (**this**.includeAnnotationConfig) {
28. AnnotationConfigUtils.registerAnnotationConfigProcessors(**this**.registry);
29. }
30. //返回注册的Bean个数
31. **return** **this**.registry.getBeanDefinitionCount() - beanCountAtScanStart;
32. }
33. //类路径Bean定义扫描器扫描给定包及其子包
34. **protected** Set<BeanDefinitionHolder> doScan(String... basePackages) {
35. //创建一个集合，存放扫描到Bean定义的封装类
36. Set<BeanDefinitionHolder> beanDefinitions = **new** LinkedHashSet<BeanDefinitionHolder>();
37. //遍历扫描所有给定的包
38. **for** (String basePackage : basePackages) {
39. //调用父类ClassPathScanningCandidateComponentProvider的方法
40. //扫描给定类路径，获取符合条件的Bean定义
41. Set<BeanDefinition> candidates = findCandidateComponents(basePackage);
42. //遍历扫描到的Bean
43. **for** (BeanDefinition candidate : candidates) {
44. //获取Bean定义类中@Scope注解的值，即获取Bean的作用域
45. ScopeMetadata scopeMetadata = **this**.scopeMetadataResolver.resolveScopeMetadata(candidate);
46. //为Bean设置注解配置的作用域
47. candidate.setScope(scopeMetadata.getScopeName());
48. //为Bean生成名称
49. String beanName = **this**.beanNameGenerator.generateBeanName(candidate, **this**.registry);
50. //如果扫描到的Bean不是Spring的注解Bean，则为Bean设置默认值，
51. //设置Bean的自动依赖注入装配属性等
52. **if** (candidate **instanceof** AbstractBeanDefinition) {
53. postProcessBeanDefinition((AbstractBeanDefinition) candidate, beanName);
54. }
55. //如果扫描到的Bean是Spring的注解Bean，则处理其通用的Spring注解
56. **if** (candidate **instanceof** AnnotatedBeanDefinition) {
57. //处理注解Bean中通用的注解，在分析注解Bean定义类读取器时已经分析过  AnnotationConfigUtils.processCommonDefinitionAnnotations((AnnotatedBeanDefinition) candidate);
58. }
59. //根据Bean名称检查指定的Bean是否需要在容器中注册，或者在容器中冲突
60. **if** (checkCandidate(beanName, candidate)) {
61. BeanDefinitionHolder definitionHolder = **new** BeanDefinitionHolder(candidate, beanName);
62. //根据注解中配置的作用域，为Bean应用相应的代理模式
63. definitionHolder = AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder, **this**.registry);
64. beanDefinitions.add(definitionHolder);
65. //向容器注册扫描到的Bean
66. registerBeanDefinition(definitionHolder, **this**.registry);
67. }
68. }
69. }
70. **return** beanDefinitions;
71. }
72. ……
73. }

类路径Bean定义扫描器ClassPathBeanDefinitionScanner主要通过findCandidateComponents方法调用其父类ClassPathScanningCandidateComponentProvider类来扫描获取给定包及其子包下的类。

#### (3).ClassPathScanningCandidateComponentProvider扫描给定包及其子包的类：

ClassPathScanningCandidateComponentProvider类的findCandidateComponents方法具体实现扫描给定类路径包的功能，主要源码如下：

1. **public** **class** ClassPathScanningCandidateComponentProvider **implements** ResourceLoaderAware {
2. //保存过滤规则要包含的注解，即Spring默认的@Component、@Repository、@Service、//@Controller注解的Bean，以及JavaEE6的@ManagedBean和JSR-330的@Named注解
3. **private** **final** List<TypeFilter> includeFilters = **new** LinkedList<TypeFilter>();
4. //保存过滤规则要排除的注解
5. **private** **final** List<TypeFilter> excludeFilters = **new** LinkedList<TypeFilter>();
6. //构造方法，该方法在子类ClassPathBeanDefinitionScanner的构造方法中被调用
7. **public** ClassPathScanningCandidateComponentProvider(**boolean** useDefaultFilters) {
8. //如果使用Spring默认的过滤规则，则向容器注册过滤规则
9. **if** (useDefaultFilters) {
10. registerDefaultFilters();
11. }
12. }
13. //向容器注册过滤规则
14. **protected** **void** registerDefaultFilters() {
15. //向要包含的过滤规则中添加@Component注解类，注意Spring中@Repository
16. //@Service和@Controller都是Component，因为这些注解都添加了@Component注解
17. **this**.includeFilters.add(**new** AnnotationTypeFilter(Component.**class**));
18. //获取当前类的类加载器
19. ClassLoader cl = ClassPathScanningCandidateComponentProvider.**class**.getClassLoader();
20. **try** {
21. //向要包含的过滤规则添加JavaEE6的@ManagedBean注解
22. **this**.includeFilters.add(**new** AnnotationTypeFilter(
23. ((Class<? **extends** Annotation>) cl.loadClass("javax.annotation.ManagedBean")), **false**));
24. logger.info("JSR-250 'javax.annotation.ManagedBean' found and supported for component scanning");
25. }
26. **catch** (ClassNotFoundException ex) {
27. // JSR-250 1.1 API (as included in <a href="http://lib.csdn.net/base/17" class='replace\_word' title="undefined" target='\_blank' style='color:#df3434; font-weight:bold;'>Java EE</a> 6) not available - simply skip.
28. }
29. **try** {
30. //向要包含的过滤规则添加@Named注解
31. **this**.includeFilters.add(**new** AnnotationTypeFilter(
32. ((Class<? **extends** Annotation>) cl.loadClass("javax.inject.Named")), **false**));
33. logger.info("JSR-330 'javax.inject.Named' annotation found and supported for component scanning");
34. }
35. **catch** (ClassNotFoundException ex) {
36. // JSR-330 API not available - simply skip.
37. }
38. }
39. //扫描给定类路径的包
40. **public** Set<BeanDefinition> findCandidateComponents(String basePackage) {
41. //创建存储扫描到的类的集合
42. Set<BeanDefinition> candidates = **new** LinkedHashSet<BeanDefinition>();
43. **try** {
44. //解析给定的包路径，this.resourcePattern=” \*\*/\*.class”，
45. //ResourcePatternResolver.CLASSPATH\_ALL\_URL\_PREFIX=“classpath:”
46. //resolveBasePackage方法将包名中的”.”转换为文件系统的”/”
47. String packageSearchPath = ResourcePatternResolver.CLASSPATH\_ALL\_URL\_PREFIX +
48. resolveBasePackage(basePackage) + "/" + **this**.resourcePattern;
49. //将给定的包路径解析为Spring资源对象
50. Resource[] resources = **this**.resourcePatternResolver.getResources(packageSearchPath);
51. **boolean** traceEnabled = logger.isTraceEnabled();
52. **boolean** debugEnabled = logger.isDebugEnabled();
53. //遍历扫描到的资源
54. **for** (Resource resource : resources) {
55. **if** (traceEnabled) {
56. logger.trace("Scanning " + resource);
57. }
58. **if** (resource.isReadable()) {
59. **try** {
60. //为指定资源获取元数据读取器，元信息读取器通过汇编(ASM)读//取资源元信息
61. MetadataReader metadataReader = **this**.metadataReaderFactory.getMetadataReader(resource);
62. //如果扫描到的类符合容器配置的过滤规则
63. **if** (isCandidateComponent(metadataReader)) {
64. //通过汇编(ASM)读取资源字节码中的Bean定义元信息
65. ScannedGenericBeanDefinition sbd = **new** ScannedGenericBeanDefinition(metadataReader);
66. //设置Bean定义来源于resource
67. sbd.setResource(resource);
68. //为元数据元素设置配置资源对象
69. sbd.setSource(resource);
70. //检查Bean是否是一个可实例化的对象
71. **if** (isCandidateComponent(sbd)) {
72. **if** (debugEnabled) {
73. logger.debug("Identified candidate component class: " + resource);
74. }
75. candidates.add(sbd);
76. }
77. **else** {
78. **if** (debugEnabled) {
79. logger.debug("Ignored because not a concrete top-level class: " + resource);
80. }
81. }
82. }
83. **else** {
84. **if** (traceEnabled) {
85. logger.trace("Ignored because not matching any filter: " + resource);
86. }
87. }
88. }
89. **catch** (Throwable ex) {
90. **throw** **new** BeanDefinitionStoreException(
91. "Failed to read candidate component class: " + resource, ex);
92. }
93. }
94. **else** {
95. **if** (traceEnabled) {
96. logger.trace("Ignored because not readable: " + resource);
97. }
98. }
99. }
100. }
101. **catch** (IOException ex) {
102. **throw** **new** BeanDefinitionStoreException("I/O failure during classpath scanning", ex);
103. }
104. **return** candidates;
105. }
106. //判断元信息读取器读取的类是否符合容器定义的注解过滤规则
107. **protected** **boolean** isCandidateComponent(MetadataReader metadataReader) **throws** IOException {
108. //如果读取的类的注解在排除注解过滤规则中，返回false
109. **for** (TypeFilter tf : **this**.excludeFilters) {
110. **if** (tf.match(metadataReader, **this**.metadataReaderFactory)) {
111. **return** **false**;
112. }
113. }
114. //如果读取的类的注解在包含的注解的过滤规则中，则返回ture
115. **for** (TypeFilter tf : **this**.includeFilters) {
116. **if** (tf.match(metadataReader, **this**.metadataReaderFactory)) {
117. **return** **true**;
118. }
119. }
120. //如果读取的类的注解既不在排除规则，也不在包含规则中，则返回false
121. **return** **false**;
122. }
123. ……
124. }

### 5.AnnotationConfigWebApplicationContext载入注解Bean定义：

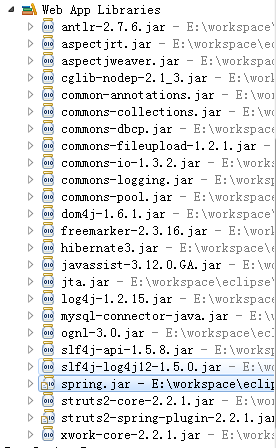
AnnotationConfigWebApplicationContext是AnnotationConfigApplicationContext的Web版，它们对于注解Bean的注册和扫描是基本相同的，但是AnnotationConfigWebApplicationContext对注解Bean定义的载入稍有不同，AnnotationConfigWebApplicationContext注入注解Bean定义源码如下：

1. //载入注解Bean定义资源
2. **protected** **void** loadBeanDefinitions(DefaultListableBeanFactory beanFactory) {
3. //为容器设置注解Bean定义读取器
4. AnnotatedBeanDefinitionReader reader = **new** AnnotatedBeanDefinitionReader(beanFactory);
5. //为容器设置类路径Bean定义扫描器
6. ClassPathBeanDefinitionScanner scanner = **new** ClassPathBeanDefinitionScanner(beanFactory);
7. //获取容器的Bean名称生成器
8. BeanNameGenerator beanNameGenerator = getBeanNameGenerator();
9. //获取容器的作用域元信息解析器
10. ScopeMetadataResolver scopeMetadataResolver = getScopeMetadataResolver();
11. //为注解Bean定义读取器和类路径扫描器设置Bean名称生成器
12. **if** (beanNameGenerator != **null**) {
13. reader.setBeanNameGenerator(beanNameGenerator);
14. scanner.setBeanNameGenerator(beanNameGenerator);
15. }
16. //为注解Bean定义读取器和类路径扫描器设置作用域元信息解析器
17. **if** (scopeMetadataResolver != **null**) {
18. reader.setScopeMetadataResolver(scopeMetadataResolver);
19. scanner.setScopeMetadataResolver(scopeMetadataResolver);
20. }
21. //获取容器定义的Bean定义资源路径
22. String[] configLocations = getConfigLocations();
23. //如果定位的Bean定义资源路径不为空
24. **if** (configLocations != **null**) {
25. **for** (String configLocation : configLocations) {
26. **try** {
27. //使用当前容器的类加载器加载定位路径的字节码类文件
28. Class<?> clazz = getClassLoader().loadClass(configLocation);
29. **if** (logger.isInfoEnabled()) {
30. logger.info("Successfully resolved class for [" + configLocation + "]");
31. }
32. reader.register(clazz);
33. }
34. **catch** (ClassNotFoundException ex) {
35. **if** (logger.isDebugEnabled()) {
36. logger.debug("Could not load class for config location [" + configLocation +
37. "] - trying package scan. " + ex);
38. }
39. //如果容器类加载器加载定义路径的Bean定义资源失败，则启用
40. //容器类路径扫描器扫描给定路径包及其子包中的类
41. **int** count = scanner.scan(configLocation);
42. **if** (logger.isInfoEnabled()) {
43. **if** (count == 0) {
44. logger.info("No annotated classes found for specified class/package [" + configLocation + "]");
45. }
46. **else** {
47. logger.info("Found " + count + " annotated classes in package [" + configLocation + "]");
48. }
49. }
50. }
51. }
52. }
53. }

## 7、ssh整合

**Struts2.2、spring2.5、hibernate3整合**

### 1、首先导入开发所必要的包，如下图：



### 2、配置web.xml文件：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-app id="WebApp\_ID" version="2.5"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns="http://java.sun.com/xml/ns/javaee"  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"  >  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>  classpath:bean.xml,  classpath:daoContext.xml,  classpath:serviceContext.xml  </param-value>  </context-param>  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener>    <filter>  <filter-name>struts2</filter-name>  <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>struts2</filter-name>  <url-pattern>\*.htm</url-pattern>  </filter-mapping>  <welcome-file-list>  <welcome-file>index.html</welcome-file>  </welcome-file-list>  </web-app> |

上面的配置已经将spring和struts整合到一起了，代码看上去好象只是分别启动了spring和struts2而已，其实这主要是struts-spring-plugin-\*.jar这个插件的功劳。在配置web.xml文件时，还应该注意以下两点：

* 如果spring采用多个配置文件的话，需要注意当把配置文件存放在classpath路径下，则contextConfigLocation的每个配置文件前都应该加上classpath,每个配置文件用逗号隔开，如：classpath:bean.xml,classpath:applicationContext.xml，如果把配置文件放在/WEB-INF/下，则只要写出路经即可，如：/WEB-INF/applicationContext.xml，/WEB-INF/daoContext.xml
* Struts2默认拦截的是.action的后缀，如果要更改这个后缀，需要修改struts2的配置文件struts.xml，即在struts.xml的<struts></struts>中添加<constant name=”struts.action.extension” value=”action,do,htm”>，如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?>  <!DOCTYPE struts PUBLIC  "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"  "http://struts.apache.org/dtds/struts-2.0.dtd">  <struts>  …  <constant name="struts.action.extension" value="action,htm,do" />  …  </struts> |

1. 接下来要做的就是spring和hibernate的整合。

我们都知道spring提供了一整套对MVC三层模型在支持，当然也提供了对hibernate的支持。通过spring和hibernate的整合，我们可以不用再写hibernate的配置文件hbm.cfg.xml，这些只需要在spring在bean中配置就可以了。如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  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-2.5.xsd">    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  <property name="url" value="jdbc:mysql://localhost:3306/test"/>  <property name="username" value="root"/>  <property name="password" value="root"/>  </bean>  <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">  <property name="dataSource" ref="dataSource"/>  <property name="mappingResources">  <list>  <value>com/sque/domain/User.hbm.xml</value>  </list>  </property>  <property name="hibernateProperties">  <value>  hibernate.dialect=org.hibernate.dialect.MySQLDialect  </value>  </property>  </bean>  这里可以写dao层的依赖注入  </beans> |

下面对上面的代码解释一下，dataSource这个bean是配置和数据库的链接。这里用的是mysql。sessionFactory中的mappingResources属性是用配置\*\*\*.hbm.xml的文件，就是对象和数据库映射的文件，而hibernateProperties属性用来配置hibernate的一些设置，如：hibernate.dialect，hibernate.show\_sql等

### 3、接下来就是在spring中配置事务的管理了，如下：

|  |
| --- |
| <?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"  xmlns:tx="http://www.springframework.org/schema/tx"  xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">  <bean id="myTxManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">  <property name="sessionFactory" ref="sessionFactory"/>  </bean>    <aop:config>  <!—这里配置需要事务处理的类 -->  <aop:pointcut id="userServiceMethods" expression="execution(\* com.sque.service.impl.UserServiceImpl.\*(..))"/>  <aop:advisor advice-ref="txAdvice" pointcut-ref="userServiceMethods"/>  </aop:config>  <!—这里配置需要事务应该满足的一些特性 -->  <tx:advice id="txAdvice" transaction-manager="myTxManager">    <!--<tx:attributes>  <tx:method name="increasePrice\*" propagation="REQUIRED"/>  <tx:method name="someOtherBusinessMethod" propagation="REQUIRES\_NEW"/>  <tx:method name="\*" propagation="SUPPORTS" read-only="true"/>  </tx:attributes>  -->  </tx:advice>  这里可以写service层的依赖注入  </beans> |

到这里，三个框架的整合配置就算是完成了。

### 4、在struts.xml文件中action中的class属性写的是对应的spring配置文件中的bean的id

## 8、【SpringMVC+Spring+Mybatis整合程序之整合】 对于mybatis开发持久层（DAO:DataBase Access Object 持久层访问对象）有两种。 第一种：传统的开发持久层方式即需要程序员开发持久层接口和持久层实现类 第二种：mybatis代理方式开发持久层只需要程序员提供持久层接口,既然能够对传统开发方式进行优化, 帮我们广大程序员省去了大部分工作的前提就是需要我们程序员遵循一些开发规范。既然是整合框架那我这边就不再使用原始开发持久层方式。

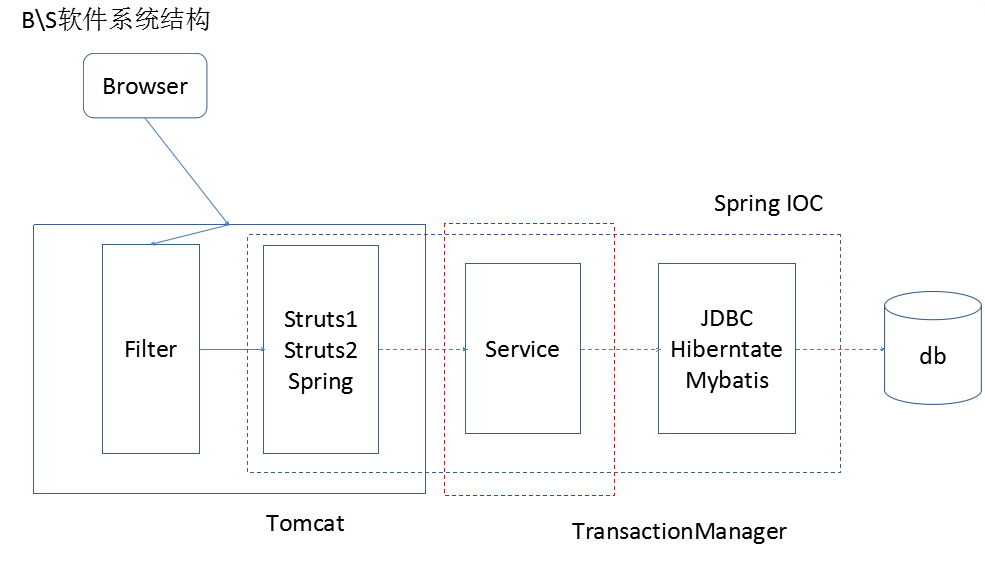
首先分析一下各个框架的职责：  
SpringMVC：负责表现层  
Service接口：处理业务  
Mapper:持久层

spring负责将各层之间整合  
通过Spring管理持久层的mapper(相当于Dao接口)  
通过Spring管理业务层的service,service中可以调用mapper接口  
Spring进行事务控制  
通过Spring管理表现层handler,handler中可以调用service接口

mapper、service、handler都属于javabean

1、第一步：整合dao层  
mybatis和spring整合，通过spring管理mapper接口。  
使用mapper的扫描器自动扫描mapper接口在spring中进行注册。

2、第二步：整合service层  
通过spring管理 service接口。  
使用配置方式将service接口配置在spring配置文件中。  
实现事务控制。

3、第三步：整合springmvc  
由于springmvc是spring的模块，不需要整合。  
没看懂?没关系再来看张图放松一下。  
  
这张图说明所有的组件都要在spring容器中运行。  
这里要说的是不熟悉spring的同学要辛苦点看了。对于其他两个框架大部分知识点我已经在之前的文章介绍过了。  
在正式动手之前先介绍一下我的开发环境：  
Eclipse Indigo-j2ee-64位、JDK-1.7.0\_67、Tomcat-7.0.65、Spring版本3.2、mybatis版本3.2.X、  
mysql-5.5.36-win32、数据库图形化操作工具：SQLyog-10.0.0-0  
**几个重要的配置文件**  
数据库脚本文件内容：直接复制执行提交

CREATE DATABASE /\*!32312 IF NOT EXISTS\*/`sms` /\*!40100 DEFAULT CHARACTER SET utf8 \*/;

USE `sms`;

/\*Table structure for table `t\_user` \*/

DROP TABLE IF EXISTS `t\_user`;

CREATE TABLE `t\_user` (

`id` int(10) unsigned NOT NULL AUTO\_INCREMENT COMMENT '唯一标识',

`username` varchar(32) DEFAULT NULL COMMENT '用户名称',

`age` int(11) DEFAULT NULL COMMENT '用户年龄',

`gender` varchar(10) DEFAULT NULL COMMENT '用户性别',

`birthday` varchar(64) DEFAULT NULL COMMENT '用户生日',

PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

log4j.properties和数据库打交道这个文件是少不了的：

log4j.rootLogger=DEBUG, Console

#Console

log4j.appender.Console=org.apache.log4j.ConsoleAppender

log4j.appender.Console.layout=org.apache.log4j.PatternLayout

log4j.appender.Console.layout.ConversionPattern=%d [%t] %-5p [%c] - %m%n

log4j.logger.java.sql.ResultSet=INFO

log4j.logger.org.apache=INFO

log4j.logger.java.sql.Connection=DEBUG

log4j.logger.java.sql.Statement=DEBUG

log4j.logger.java.sql.PreparedStatement=DEBUG

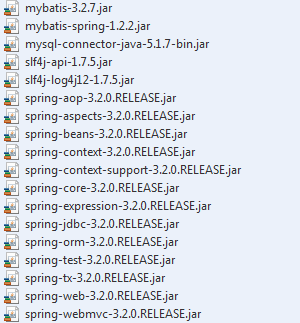
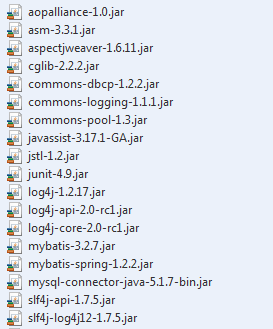
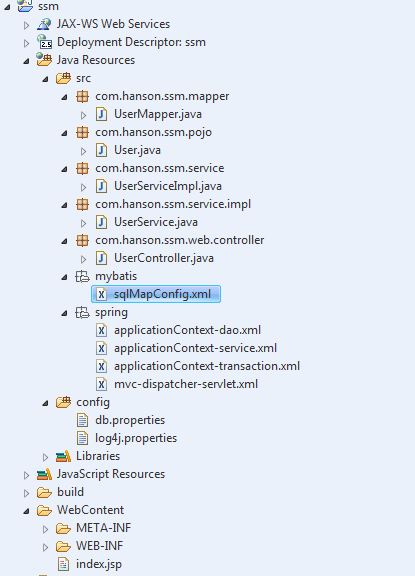
还有一个db.properties文件，我先解释一下为什么要用配置文件，因为配置文件编译的成本很小，不像一个.java文件需要经历打包、测试、发布等等环节，而改动配置文件只要对应的value是正确的就可以直接丢给现场顺手使用。

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/ssm

jdbc.username=root

jdbc.password=root

导入jar包：这边我就截图了  
  
**整体工程结构图**  
  
重点：程序中出现的配置文件介绍以及配置  
第一个文件：web.xml这个文件主要职责就是配置程序入口在这个文件中要配置上面说道的spring容器因为所有的组件都是在容器中运行的、接着要配置restful风格的url过滤器、请求参数过滤器、spring上下文监听器、以及前端控制器等等  
文件内容如下：

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"

id="WebApp\_ID" version="2.5">

<display-name>ssm</display-name>

<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

<!-- 加载spring容器 -->

<context-param>

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

<param-value>/WEB-INF/classes/spring/applicationContext-\*.xml</param-value>

</context-param>

<!-- 支持Restful风格的请求Url -->

<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>

<!-- 过滤中文乱码 -->

<filter>

<filter-name>CharacterEncodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>utf-8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>CharacterEncodingFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<!-- spring容器监听器 -->

<listener>

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

</listener>

<!-- 配置前端控制器 -->

<servlet>

<servlet-name>ssm</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!--加载前端控制器配置文件 上下文配置位置 -->

<init-param>

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

<param-value>classpath:spring/mvc-dispatcher-servlet.xml</param-value>

</init-param>

<!-- 随服务器启动 -->

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

</servlet>

<servlet-mapping>

<servlet-name>ssm</servlet-name>

<!-- restful风格url -->

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

applicationContext-dao.xml文件中主要负责配置：加载db.properties、配置数据源、配置SqlSessionFactoryBean、Mapper扫描器  
内容如下：

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mvc="http://www.springframework.org/schema/mvc"

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-3.2.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.2.xsd ">

<!-- 加载db.properties文件中的内容,db.properties文件中的key要有一定的特殊规则 -->

<context:property-placeholder location="classpath:db.properties"/>

<!-- 配置数据源,使用dbcp连接池 -->

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="${jdbc.driver}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

<property name="maxActive" value="30"/>

<property name="maxIdle" value="5"/>

</bean>

<!-- 配置SqlSessionFactory -->

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<!-- 数据源 -->

<property name="dataSource" ref="dataSource"/>

<!-- 加载mybatis的全局配置文件 -->

<property name="configLocation" value="classpath:mybatis/sqlMapConfig.xml" />

</bean>

<!-- 配置Mapper扫描器 -->

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<!-- 扫描包路径,如果需要扫描多个包中间用半角逗号隔开 -->

<property name="basePackage" value="com.hanson.ssm.mapper"/>

<!-- 这边不能使用ref="sqlSessionFactory"原因是因为上面加载配置文件导致这边引用会报错 -->

<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />

</bean>

</beans>

applicationContext-service.xml该文件主要负责扫描业务层组件  
内容如下：

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mvc="http://www.springframework.org/schema/mvc"

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-3.2.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.2.xsd ">

<!-- 扫描标注@Repository注解的service -->

<context:component-scan base-package="com.hanson.ssm.service.impl.\*"/>

</beans>

applicationContext-transaction.xml配置文件主要负责处理事务等（这个需要了解spring AOP概念、代理模式、反射（必须要会）等技术）  
文件主要内容：

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mvc="http://www.springframework.org/schema/mvc"

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-3.2.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.2.xsd ">

<!-- 事务管理器 对mybatis操作数据库事务控制，spring使用jdbc的事务控制类 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<!-- 数据源 dataSource在applicationContext-dao.xml中配置了 -->

<property name="dataSource" ref="dataSource" />

</bean>

<!-- 通知 -->

<tx:advice id="txAdvice" transaction-manager="transactionManager">

<tx:attributes>

<!-- 传播行为 -->

<tx:method name="save\*" propagation="REQUIRED" />

<tx:method name="delete\*" propagation="REQUIRED" />

<tx:method name="insert\*" propagation="REQUIRED" />

<tx:method name="update\*" propagation="REQUIRED" />

<tx:method name="find\*" propagation="SUPPORTS" read-only="true" />

<tx:method name="get\*" propagation="SUPPORTS" read-only="true" />

<tx:method name="select\*" propagation="SUPPORTS" read-only="true" />

</tx:attributes>

</tx:advice>

<!-- aop -->

<aop:config>

<aop:advisor advice-ref="txAdvice"

pointcut="execution(\* com.hanson.ssm.service.impl.\*.\*(..))" />

</aop:config>

</beans>

mvc-dispatcher-servlet.xml这个配置文件主要负责加载标注@Controller类、打开注解的处理器适配器、注解的处理器映射器、视图解析器等等  
文件内容如下：

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mvc="http://www.springframework.org/schema/mvc"

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-3.2.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.2.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.2.xsd ">

<!-- 组件扫描器扫描这一层组要扫描处理器 -->

<context:component-scan base-package="com.hanson.ssm.web.controller.\*"></context:component-scan>

<!-- 配置注解的映射器和适配器以及其他配置 -->

<mvc:annotation-driven></mvc:annotation-driven>

<!-- 处理静态资源问题 -->

<mvc:default-servlet-handler />

<!-- 配置视图解析器 -->

<bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/" />

<property name="suffix" value=".jsp" />

</bean>

</beans>

sqlMapConfig.xml这个配置文件主要配置配置mybatis框架的一些设置例如开启二级缓存、设置pojo的别名等等  
文件内容如下：

<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<!-- 全局setting配置,根据需要添加 -->

<!-- 配置别名 -->

<typeAliases>

<!-- 批量扫描设置别名 -->

<package name="com.hanson.ssm.pojo"/>

</typeAliases>

<!-- 配置Mapper

备注:由于使用Spring整合mybtais的整合包进行mapper扫描,这里不需要配置了

必须遵循:mapper.xml和mapper.java文件同名且在同一目录下

<mappers></mappers>

-->

</configuration>

总结：当这些文件都配置好，java类可以先只写类名加上注解，运行没有报错说明框架就被整合成功了。

## 9、spring新特性：spring boot、spring data

### 一、spring data

#### 第一章：Spring Data JPA入门

Spring Data是什么

Spring Data是一个用于简化数据库访问，并支持云服务的开源框架。其主要目标是使得对数据的访问变得方便快捷，并支持map-reduce框架和云计算数据服务。 Spring Data 包含多个子项目：

**Commons - 提供共享的基础框架，适合各个子项目使用，支持跨数据库持久化**

**JPA - 简化创建 JPA 数据访问层和跨存储的持久层功能**

Hadoop - 基于 Spring 的 Hadoop 作业配置和一个 POJO 编程模型的 MapReduce 作业

Key-Value  - 集成了 Redis 和 Riak ，提供多个常用场景下的简单封装

Document - 集成文档数据库：CouchDB 和 MongoDB 并提供基本的配置映射和资料库支持

Graph - 集成 Neo4j 提供强大的基于 POJO 的编程模型

Graph Roo AddOn - Roo support for Neo4j

JDBC Extensions - 支持 Oracle RAD、高级队列和高级数据类型

Mapping - 基于 Grails 的提供对象映射框架，支持不同的数据库

Examples - 示例程序、文档和图数据库

Guidance - 高级文档

Spring Data JPA是什么

由Spring提供的一个用于简化JPA开发的框架

nSpring Data JPA能干什么

可以极大的简化JPA的写法，可以在几乎不用写实现的情况下，实现对数据的访问和操作。除了CRUD外，还包括如分页、排序等一些常用的功能。

Spring Data JPA有什么

主要来看看Spring Data JPA提供的接口，也是Spring Data JPA的核心概念：

1：Repository：最顶层的接口，是一个空的接口，目的是为了统一所有Repository的类型，且能让组件扫描的时候自动识别。

2：CrudRepository ：是Repository的子接口，提供CRUD的功能

3：PagingAndSortingRepository：是CrudRepository的子接口，添加分页和排序的功能

4：JpaRepository：是PagingAndSortingRepository的子接口，增加了一些实用的功能，比如：批量操作等。

5：JpaSpecificationExecutor：用来做负责查询的接口

6：Specification：是Spring Data JPA提供的一个查询规范，要做复杂的查询，只需围绕这个规范来设置查询条件即可

#### HelloWorld

n环境构建

在Eclipse里面构建一个普通的Java工程，主要就是要加入一堆的jar包。

1：首先去官网下载Spring Data Common 和 Spring Data JPA的包，把里面dist的jar包加入到工程中，这里是spring-data-commons-1.5.0.RELEASE.jar和spring-data-jpa-1.3.2.RELEASE.jar

2：把Spring3.2.3的jar包添加到工程中

3：JPA的实现选用的是Hibernate4.2.0，总共还需要额外加入如下的jar：

|  |
| --- |
| antlr-2.7.7.jar  aopalliance-1.0.jar  asm-3.2.jar  aspectjrt-1.7.1.jar  aspectjweaver-1.7.1.jar  commons-beanutils-1.8.3.jar  commons-codec-1.7.jar  commons-collections-3.2.1.jar  commons-dbcp-1.4.jar  commons-fileupload-1.2.2.jar  commons-io-2.4.jar  commons-lang3-3.1.jar  commons-logging-1.1.1.jar  commons-pool-1.6.jar  dom4j-1.6.1.jar  hibernate-commons-annotations-4.0.1.Final.jar  hibernate-core-4.2.0.Final.jar  hibernate-entitymanager-4.2.0.Final.jar  hibernate-jpa-2.0-api-1.0.1.Final.jar  javassist-3.15.0-GA.jar  jboss-logging-3.1.0.GA.jar  jboss-transaction-api\_1.1\_spec-1.0.0.Final.jar  mysql-connector-java-5.1.9.jar  slf4j-api-1.7.3.jar |

n实体对象，就是以前的实现方式

|  |
| --- |
| @Entity  @Table(name="tbl\_user")  public class UserModel {  @Id  private Integer uuid;  private String name;  private Integer age;  //省略getter/setter  } |

nDAO的接口

|  |
| --- |
| public interface UserRepository extends JpaRepository<UserModel, Integer>{  //空的，可以什么都不用写  } |

无需提供实现，Spring Data JPA会为我们搞定一切

n写个逻辑层的Service，其实就相当于DAO的客户端，用来测试

|  |
| --- |
| @Service  @Transactional  public class Client {  @Autowired  private UserRepository ur;    public void testAdd(UserModel um){ ur.save(um); }    public static void main(String[] args) {  ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");    Client c = (Client)ctx.getBean("client");  UserModel um = new UserModel();  um.setAge(1);  um.setName("张三");  um.setUuid(1);    c.testAdd(um);  } } |

n同样需要在Spring的配置文件中配置，基本跟使用注解的配置类似：

|  |
| --- |
| <?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:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:jpa="http://www.springframework.org/schema/data/jpa"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.0.xsd  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.0.xsd  http://www.springframework.org/schema/data/jpa http://www.springframework.org/schema/data/jpa/spring-jpa.xsd  ">  <context:component-scan base-package="cn.javass">  <context:exclude-filter type="annotation“ expression="org.springframework.stereotype.Controller"/>  </context:component-scan>  <aop:aspectj-autoproxy proxy-target-class="true"/>  <!-- 开启注解事务 只对当前配置文件有效 -->  <tx:annotation-driven transaction-manager="transactionManager" proxy-target-class="true"/>  **<jpa:repositories**  **base-package="cn.javass"**  **repository-impl-postfix="Impl"**  **entity-manager-factory-ref="entityManagerFactory"**  **transaction-manager-ref="transactionManager">**  **</jpa:repositories>**  **<bean id="entityManagerFactory"**  **class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">**          <property name="dataSource" ref="dataSource"/>          <property name="packagesToScan" value="cn.javass"/>          <property name="persistenceProvider">              <bean class="org.hibernate.ejb.HibernatePersistence"/>          </property>          <property name="jpaVendorAdapter">              <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">                  <property name="generateDdl" value="false"/>                  <property name="database" value="MYSQL"/>                  <property name="databasePlatform" value="org.hibernate.dialect.MySQL5InnoDBDialect"/>                  <property name="showSql" value="true"/>              </bean>          </property>          <property name="jpaDialect">              <bean class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"/>          </property>          <property name="jpaPropertyMap">              <map>                  <entry key="hibernate.query.substitutions" value="true 1, false 0"/>                  <entry key="hibernate.default\_batch\_fetch\_size" value="16"/>                  <entry key="hibernate.max\_fetch\_depth" value="2"/>                  <entry key="hibernate.generate\_statistics" value="true"/>                  <entry key="hibernate.bytecode.use\_reflection\_optimizer" value="true"/>                  <entry key="hibernate.cache.use\_second\_level\_cache" value="false"/>                  <entry key="hibernate.cache.use\_query\_cache" value="false"/>              </map>          </property>      </bean>  <!--事务管理器配置-->      <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">          <property name="entityManagerFactory" ref="entityManagerFactory"/>      </bean>    <bean name="dataSource" class="org.apache.commons.dbcp.BasicDataSource">  <property name="driverClassName"><value>org.gjt.mm.mysql.Driver</value></property>  <property name="url"><value>jdbc:mysql://localhost:3306/cc?useUnicode=true&amp;characterEncoding=UTF-8</value></property>  <property name="username"> <value>root</value> </property>  <property name="password" value="cc"/>  </bean>  </beans> |

配置完成后，可以去运行Client测试一下了，当然数据库和表需要先准备好

也可以在<jpa:repositories>下面添加filter，形如：

|  |
| --- |
| <repositories base-package="com.acme.repositories">  <context:exclude-filter type="regex" expression=".\*SomeRepository" />  </repositories> |

#### 第二章：JpaRepository基本功能

JpaRepository的基本功能示范

具体的看代码演示

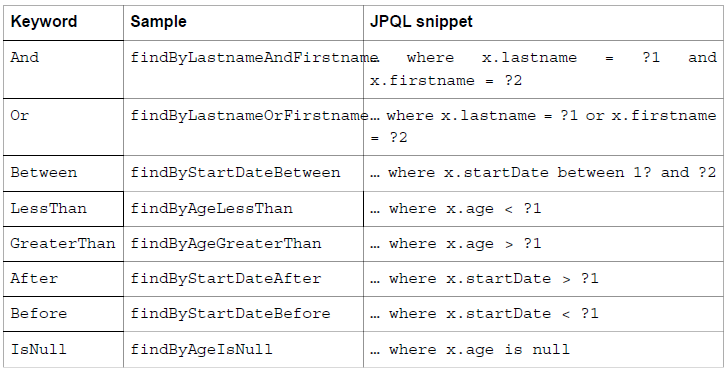
其中：Pageable接口的实现类是PageRequest，Page接口的实现类是PageImpl。

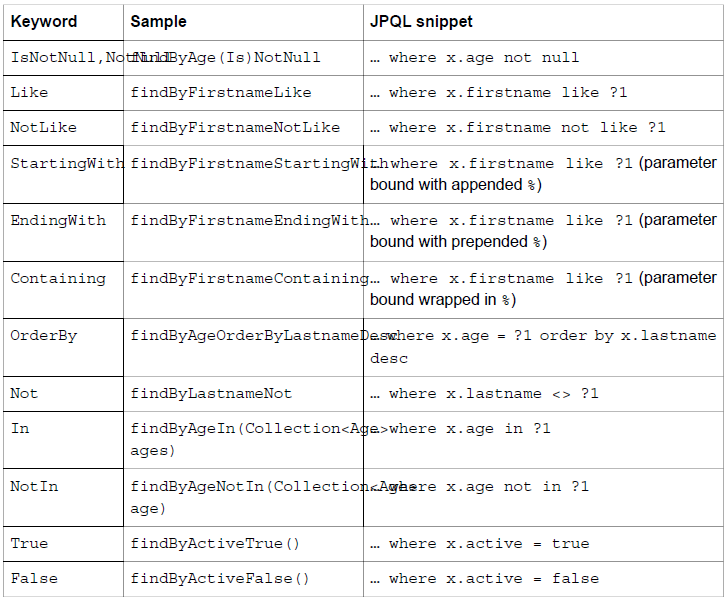
示例如下：

|  |
| --- |
| Page<UserModel> p =  ur.findAll(new PageRequest(0,2,new Sort(new Order(Direction. DESC,"uuid"))));  System. out.println("list="+p.getContent()); |

#### 第三章：JpaRepository的查询

直接在接口中定义查询方法，如果是符合规范的，可以不用写实现，目前支持的关键字写法如下：





Spring Data JPA框架在进行方法名解析时，会先把方法名多余的前缀截取掉，比如 find、findBy、read、readBy、get、getBy，然后对剩下部分进行解析。

假如创建如下的查询：findByUserDepUuid()，框架在解析该方法时，首先剔除 findBy，然后对剩下的属性进行解析，假设查询实体为Doc

1：先判断 userDepUuid （根据 POJO 规范，首字母变为小写）是否为查询实体的一个属性，如果是，则表示根据该属性进行查询；如果没有该属性，继续第二步；

2：从右往左截取第一个大写字母开头的字符串此处为Uuid），然后检查剩下的字符串是否为查询实体的一个属性，如果是，则表示根据该属性进行查询；如果没有该属性，则重复第二步，继续从右往左截取；最后假设user为查询实体的一个属性；

3：接着处理剩下部分（DepUuid），先判断 user 所对应的类型是否有depUuid属性，如果有，则表示该方法最终是根据 “ Doc.user.depUuid” 的取值进行查询；否则继续按照步骤 2 的规则从右往左截取，最终表示根据 “Doc.user.dep.uuid” 的值进行查询。

4:可能会存在一种特殊情况，比如 Doc包含一个 user 的属性，也有一个 userDep 属性，此时会存在混淆。可以明确在属性之间加上 "\_" 以显式表达意图，比如 "findByUser\_DepUuid()" 或者 "findByUserDep\_uuid()"

特殊的参数： 还可以直接在方法的参数上加入分页或排序的参数，比如：

Page<UserModel> findByName(String name, Pageable pageable);

List<UserModel> findByName(String name, Sort sort);

也可以使用JPA的NamedQueries，方法如下：

1：在实体类上使用@NamedQuery，示例如下：

@NamedQuery(name = "UserModel.findByAge",query = "select o from UserModel o where o.age >= ?1")

2：在自己实现的DAO的Repository接口里面定义一个同名的方法，示例如下：

public List<UserModel> findByAge(int age);

3：然后就可以使用了，Spring会先找是否有同名的NamedQuery，如果有，那么就不会按照接口定义的方法来解析。

##### 使用@Query

可以在自定义的查询方法上使用@Query来指定该方法要执行的查询语句，比如：

@Query("select o from UserModel o where o.uuid=?1")

public List<UserModel> findByUuidOrAge(int uuid);

**注意：**

1：方法的参数个数必须和@Query里面需要的参数个数一致

2：如果是like，后面的参数需要前面或者后面加“%”，比如下面都对：

@Query("select o from UserModel o where o.name like ?1%")

public List<UserModel> findByUuidOrAge(String name);

@Query("select o from UserModel o where o.name like %?1")

public List<UserModel> findByUuidOrAge(String name);

@Query("select o from UserModel o where o.name like %?1%")

public List<UserModel> findByUuidOrAge(String name);

当然，这样在传递参数值的时候就可以不加‘%’了，当然加了也不会错

n还可以使用@Query来指定本地查询，只要设置nativeQuery为true，比如：

@Query(value="select \* from tbl\_user where name like %?1" ,nativeQuery=true)

public List<UserModel> findByUuidOrAge(String name);

**注意：**当前版本的本地查询不支持翻页和动态的排序

使用命名化参数，使用@Param即可，比如：

@Query(value="select o from UserModel o where o.name like %:nn")

public List<UserModel> findByUuidOrAge(@Param("nn") String name);

同样支持更新类的Query语句，添加@Modifying即可，比如：

@Modifying

@Query(value="update UserModel o set o.name=:newName where o.name like %:nn")

public int findByUuidOrAge(@Param("nn") String name,@Param("newName") String newName);

**注意：**

1：方法的返回值应该是int，表示更新语句所影响的行数

2：在调用的地方必须加事务，没有事务不能正常执行

##### JpaRepository的查询功能

创建查询的顺序

Spring Data JPA 在为接口创建代理对象时，如果发现同时存在多种上述情况可用，它该优先采用哪种策略呢？

<jpa:repositories> 提供了 query-lookup-strategy 属性，用以指定查找的顺序。它有如下三个取值：

1：create-if-not-found：如果方法通过@Query指定了查询语句，则使用该语句实现查询；如果没有，则查找是否定义了符合条件的命名查询，如果找到，则使用该命名查询；如果两者都没有找到，则通过解析方法名字来创建查询。这是 query-lookup-strategy 属性的默认值

2：create：通过解析方法名字来创建查询。即使有符合的命名查询，或者方法通过 @Query指定的查询语句，都将会被忽略

3：use-declared-query：如果方法通过@Query指定了查询语句，则使用该语句实现查询；如果没有，则查找是否定义了符合条件的命名查询，如果找到，则使用该命名查询；如果两者都没有找到，则抛出异常

#### 第四章：客户化扩展JpaRepository

如果你不想暴露那么多的方法，可以自己订制自己的Repository，还可以在自己的Repository里面添加自己使用的公共方法

当然更灵活的是自己写一个实现类，来实现自己需要的方法

1：写一个与接口同名的类，加上后缀为Impl，这个在前面xml里面配置过，可以自动被扫描到。这个类不需要实现任何接口。

2：在接口中加入自己需要的方法，比如：

public Page<Object[]> getByCondition(UserQueryModel u);

3：在实现类中，去实现这个方法就好了，会被自动找到

**java代码：**

1. **public** **class** UserRepositoryImpl {
2. @PersistenceContext
3. **private** EntityManager em;
4. **public** Page<Object[]> getByCondition(UserQueryModel u){
5. String hql = "select o.uuid,o.name from UserModel o where 1=1 and o.uuid=:uuid";
6. Query q = em.createQuery(hql);
7. q.setParameter("uuid", u.getUuid());
8. q.setFirstResult(0);
9. q.setMaxResults(1);
10. Page<Object[]> page = **new** PageImpl<Object[]>(q.getResultList(),**new** PageRequest(0,1),3);
11. **return** page;
12. }}

#### 第五章：Specifications查询

Spring Data JPA支持JPA2.0的Criteria查询，相应的接口是JpaSpecificationExecutor。

Criteria 查询：是一种类型安全和更面向对象的查询

这个接口基本是围绕着Specification接口来定义的， Specification接口中只定义了如下一个方法：

Predicate toPredicate(Root<T> root, CriteriaQuery<?> query, CriteriaBuilder cb);

要理解这个方法，以及正确的使用它，就需要对JPA2.0的Criteria查询有一个足够的熟悉和理解，因为这个方法的参数和返回值都是JPA标准里面定义的对象。

##### **Criteria查询**基本概念

Criteria 查询是以元模型的概念为基础的，元模型是为具体持久化单元的受管实体定义的，这些实体可以是实体类，嵌入类或者映射的父类。

CriteriaQuery接口：代表一个specific的顶层查询对象，它包含着查询的各个部分，比如：select 、from、where、group by、order by等

**注意：**CriteriaQuery对象只对实体类型或嵌入式类型的Criteria查询起作用

Root接口：代表Criteria查询的根对象，Criteria查询的查询根定义了实体类型，能为将来导航获得想要的结果，它与SQL查询中的FROM子句类似

1：Root实例是类型化的，且定义了查询的FROM子句中能够出现的类型。

2：查询根实例能通过传入一个实体类型给 AbstractQuery.from方法获得。

3：Criteria查询，可以有多个查询根。

4：AbstractQuery是CriteriaQuery 接口的父类，它提供得到查询根的方法。

CriteriaBuilder接口：用来构建CritiaQuery的构建器对象

Predicate：一个简单或复杂的谓词类型，其实就相当于条件或者是条件组合。

###### **Criteria查询**

基本对象的构建

1：通过EntityManager的getCriteriaBuilder或EntityManagerFactory的getCriteriaBuilder方法可以得到CriteriaBuilder对象

2：通过调用CriteriaBuilder的createQuery或createTupleQuery方法可以获得CriteriaQuery的实例

3：通过调用CriteriaQuery的from方法可以获得Root实例

过滤条件

1：过滤条件会被应用到SQL语句的FROM子句中。在criteria 查询中，查询条件通过Predicate或Expression实例应用到CriteriaQuery对象上。

2：这些条件使用 CriteriaQuery .where 方法应用到CriteriaQuery 对象上

3：CriteriaBuilder也作为Predicate实例的工厂，通过调用CriteriaBuilder 的条件方法（ equal，notEqual， gt， ge，lt， le，between，like等）创建Predicate对象。

4：复合的Predicate 语句可以使用CriteriaBuilder的and, or andnot 方法构建。

构建简单的Predicate示例：

Predicate p1=cb.like(root.get(“name”).as(String.class), “%”+uqm.getName()+“%”);

Predicate p2=cb.equal(root.get("uuid").as(Integer.class), uqm.getUuid());

Predicate p3=cb.gt(root.get("age").as(Integer.class), uqm.getAge());

构建组合的Predicate示例：

Predicate p = cb.and(p3,cb.or(p1,p2));

当然也可以形如前面动态拼接查询语句的方式，比如：

**java代码：**

1. Specification<UserModel> spec = **new** Specification<UserModel>() {
2. **public** Predicate toPredicate(Root<UserModel> root,
3. CriteriaQuery<?> query, CriteriaBuilder cb) {
4. List<Predicate> list = **new** ArrayList<Predicate>();
6. **if**(um.getName()!=**null** && um.getName().trim().length()>0){
7. list.add(cb.like(root.get("name").as(String.**class**), "%"+um.getName()+"%"));
8. }
9. **if**(um.getUuid()>0){
10. list.add(cb.equal(root.get("uuid").as(Integer.**class**), um.getUuid()));
11. }
12. Predicate[] p = **new** Predicate[list.size()];
13. **return** cb.and(list.toArray(p));
14. }
15. };

也可以使用CriteriaQuery来得到最后的Predicate，示例如下：

**java代码：**

1. Specification<UserModel> spec = **new** Specification<UserModel>() {
2. **public** Predicate toPredicate(Root<UserModel> root,
3. CriteriaQuery<?> query, CriteriaBuilder cb) {
4. Predicate p1 = cb.like(root.get("name").as(String.**class**), "%"+um.getName()+"%");
5. Predicate p2 = cb.equal(root.get("uuid").as(Integer.**class**), um.getUuid());
6. Predicate p3 = cb.gt(root.get("age").as(Integer.**class**), um.getAge());
7. //把Predicate应用到CriteriaQuery中去,因为还可以给CriteriaQuery添加其他的功能，比如排序、分组啥的
8. query.where(cb.and(p3,cb.or(p1,p2)));
9. //添加排序的功能
10. query.orderBy(cb.desc(root.get("uuid").as(Integer.**class**)));
12. **return** query.getRestriction();
13. }
14. };

**多表联接**

n多表连接查询稍微麻烦一些，下面演示一下常见的1:M，顺带演示一下1:1

n使用Criteria查询实现1对多的查询

1：首先要添加一个实体对象DepModel，并设置好UserModel和它的1对多关系，如下：

|  |
| --- |
| @Entity  @Table(name="tbl\_user")  public class UserModel {  @Id  private Integer uuid;  private String name;  private Integer age;  @OneToMany(mappedBy = "um", fetch = FetchType. LAZY, cascade = {CascadeType. ALL})  private Set<DepModel> setDep;  //省略getter/setter  }    @Entity  @Table(name="tbl\_dep")  public class DepModel {  @Id  private Integer uuid;  private String name;  @ManyToOne()    @JoinColumn(name = "user\_id", nullable = false)  //表示在tbl\_dep里面有user\_id的字段  private UserModel um = new UserModel();  //省略getter/setter  } |

2：配置好Model及其关系后，就可以在构建Specification的时候使用了，示例如下：

|  |
| --- |
| Specification<UserModel> spec = new Specification<UserModel>() {  public Predicate toPredicate(Root<UserModel> root, CriteriaQuery<?> query, CriteriaBuilder cb) {  Predicate p1 = cb.like(root.get("name").as(String.class), "%"+um.getName()+"%");  Predicate p2 = cb.equal(root.get("uuid").as(Integer.class), um.getUuid());  Predicate p3 = cb.gt(root.get("age").as(Integer.class), um.getAge());  **SetJoin<UserModel,DepModel> depJoin =** **root.join(root.getModel().getSet("setDep",DepModel.class) , JoinType.LEFT);**    **Predicate p4 = cb.equal(depJoin.get("name").as(String.class), "ddd");**  **//把Predicate应用到CriteriaQuery去,因为还可以给CriteriaQuery添加其他的功能，比如排序、分组啥** **的**  **query.where(cb.and(cb.and(p3,cb.or(p1,p2)),p4));**  //添加分组的功能  query.orderBy(cb.desc(root.get("uuid").as(Integer.class)));  return query.getRestriction();  }}; |

n接下来看看使用Criteria查询实现1:1的查询

1：在UserModel中去掉setDep的属性及其配置，然后添加如下的属性和配置：

|  |
| --- |
| @OneToOne()  @JoinColumn(name = "depUuid")  private DepModel dep;  public DepModel getDep() { return dep;}  public void setDep(DepModel dep) {this.dep = dep;  } |

2：在DepModel中um属性上的注解配置去掉，换成如下的配置：

|  |
| --- |
| @OneToOne(mappedBy = "dep", fetch = FetchType. EAGER, cascade = {CascadeType. ALL}) |

3：在Specification实现中，把SetJoin的那句换成如下的语句：

|  |
| --- |
| Join<UserModel,DepModel> depJoin =  root.join(root.getModel().getSingularAttribute("dep",DepModel.class),JoinType.LEFT);  //root.join(“dep”,JoinType.LEFT); //这句话和上面一句的功能一样，更简单 |