# Spring介绍

## IOC介绍

底层原理：反射

IOC（Inversion of Control，控制反转）：后来改名为DI（Dependency Injection，依赖注入），即将类之间的关系通过外部的Spring容器注入到程序中，而不需要类自己编码实现。

根据类名（字符串方式）来动态地生成对象，这种编程方式可以让对象在生成时才被决定到底调用哪个对象。

好处：解耦，即降低对象间依赖关系。可以通过注入不同的接口实现类来完成对象直接的调用。

2种方式实现：1）构造方法 2）setter方法

所有的bean都是通过spring容器来管理。Spring把这种相互协作的关系称为依赖关系。如A组件调用B组件的方法，可称为A组件依赖于B组件。

在实际应用开发中，需要降低耦合度，即尽量避免和降低对象之间的依赖关系。用new来创建的对象是在程序的内部代码来控制的，这种实现方式会造成组件之间的耦合。而Ioc就是解决了一般业务对象之间、业务对象与持久层之间存在的各种依赖关系。

简单实例：

interface Car {

public void speed();

}

class CarA implements Car {

public void speed() {

System.out.println("CarA speed...");

}

}

class CarB implements Car {

public void speed() {

System.out.println("CarB speed...");

}

}

class Factory {

private Car car;

public static fruit getInstance(String className) {

try {

car = (Car)Class.forName(className).newInstance();

} catch(Exception e) {

e.printStackTrace();

}

return car;

}

}

### IOC工作流程

Spring IOC容器的整个工作流程大致分为两个阶段：

第一阶段：容器启动阶段

该阶段主要完成一些准备性工作，更侧重于bean对象管理信息的收集，包括一些校验性或辅助性的工作。

经过第一阶段，所有bean定义都通过BeanDefinition的方式注册到BeanDefinitionRegistry中（未了解BeanDefinition接口和BeanDefinitionRegistry接口）。

容器启动时会通过某种途径加载ConfigurationMetaData。比如使用BeanDefinitionReader对加载的ConfigurationMetaData进行解析和分析，并将分析后的信息组装成BeanDefinition，最后把这些保存了bean定义的BeanDefinition注册到相应的BeanDefinitionRegistry，这样容器的启动工作就完成了。

模拟BeanFactory从配置文件加载bean的定义及依赖关系：

// 默认容器实现DefaultListableBeanFactory

BeanDefinitionRegistry beanRegistry = new DefaultListableBeanFactory();

// XmlBeanDefinitionReader 用于加载配置文件bean

XmlBeanDefinitionReader reader = new XmlBeanDefinitionReaderImpl(beanRegistry);

reader.loadBeanDefinitions("classpath:spring-bean.xml")

// 从容器中获取bean实例

BeanFactory container = (BeanFactory) beanRegistry;

Business business=(Business)container.getBean("beanName");

第二阶段：bean的实例化阶段

调用容器的getBean时,会触发第二阶段。

容器首先检查请求的对象之前是否已实例化。若否则根据注册的BeanDefinition所提供的信息实例化被请求对象，并为其注入依赖。当该对象装配完毕后，容器立即将其返回给请求方法使用。

BeanFactory 只是 Spring IoC 容器的一种实现，如果没有特殊指定，它采用采用延迟初始化策略：只有当访问容器中的某个对象时，才对该对象进行初始化和依赖注入操作。而在实际场景下，我们更多的是 ApplicationContext，它具有 BeanFactory 的所有能力，还提供对国际化支持、aop、事务等。它管理的 bean，在容器启动时全部完成初始化和依赖注入操作。

### 工厂模式与Spring IOC区别

相关链接：http://www.uml.org.cn/sjms/201003263.asp

区别：

-1）工厂模式中修改代码后需重新编译工厂类，而重新编译类文件需要重新启动JVM才能使用新编译的类。

-2）ioc中的依赖关系在xml配置，在xml配置修改后，spring会立即发现并重新读取一遍xml，不需要重启JVM。

-3）主要区别体现在A类对象引用了B接口的实现类B1，ioc若想引用实现类B2不需要嵌入任何工厂模式的代码。这样使用ioc彻底解耦了A和B之间的联系。

### 相关问题

控制反转：

普通控制方式：A类有个x类引用，我们创建A对象时，需要先创建x对象，在将x对象set到A对象中去，因此A主动创建X，因此A依赖于X

控制反转方式：

实现依赖注入：

扫描包获得类定义

初始化Bean，并实现依赖注入

解决Bean初始化顺序问题

问1：什么是IOC？IOC容器有多少种？

问2：BeanFactory和FactoryBean有什么区别？

BeanFactory是IOC容器的核心接口，是一个工厂类，所有bean都是由它来管理，提供获取bean的各种方法。它的职责有：实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。具体实现有XmlBeanFactory、ApplicationContext等。

FactoryBean也是一个接口，它在IOC容器的基础上给bean的实现加上了一个简单工厂模式和装饰模式。在Spring源码中有很多FactoryBean的实现类。

问3：BeanFactory和ApplicationContext有什么不同？

原始的BeanFactory无法支持spring的许多插件，如AOP功能、Web应用等。

ApplicationContext接口,它由BeanFactory接口派生而来，ApplicationContext包含BeanFactory的所有功能，还提供消息国际化、事件传播、载入多个（有继承关系）上下文使得每个上下文都专注一个特定的层次，比如应用的web层。

问4：Spring Bean创建过程中的设计模式？

## AOP介绍

### 基本概念

（1）通知（Advise）：

也称增强，就是你想要的功能，如日志、事务、权限认证等。其实就是向各程序内部注入一些逻辑代码从而增强原有程序的功能。

存在5种Advise注解：

-1）注解式：

@Before,前置通知

@AfterReturning,后置返回通知，成功执行之后

@After(finally),后置最终通知，不管有没有异常都会执行

@AfterThrowing,异常通知，抛出异常之后

@Around,环绕通知，包括前面四种

-2）存在5种上述对应xml配置式

（2）连接点(JoinPoint)：

就是Spring允许你使用通知advise的地方，声明哪些地方可以使用advise，可以在方法前、后及异常抛出等。

（3）切入点(Pointcut)：

在连接点的基础上添加，进而寻找合适的连接点进行切入通知(advise)。

（4）切面(Aspect)：

即通知和切入点的结合。通知说明了干什么和什么时候干（什么时候通过方法名中的before,after，around等就能知道），而切入点说明了在哪干（指定到底是哪个方法），这就是一个完整的切面定义。

（5）目标对象(Target Object)：

引入中所提到的目标类，也就是要被通知的对象，也就是真正的业务逻辑。

（6）引入(Introduction)：

允许我们向现有的类添加新方法属性。就是把切面(也就是新方法属性：通知定义的)用到目标类中。

（7）AOP代理(AOP proxy)：

应用通知的对象，详细内容参见设计模式的动态代理模式。

（8）织入(Weaving)：

把切面应用到目标对象来创建新的代理对象的过程。

-1）织入一般发生的三个时机：

1．编译时：当一个类文件被编译时织入，这需要特殊的编译器才可以做得到，例如AspectJ的织入编译器；

2．类加载时：使用特殊的ClassLoader在目标类被加载到程序之前增强类的字节代码；

3．运行时：切面在运行时的某时刻被织入，SpringAOP就是以这种方式织入切面，原理是使用了JDK的动态代理技术。

Spring采用动态代理织入，而AspectJ采用编译器织入和类装载期织入。

### 原理

相关链接：

http://blog.csdn.net/qukaiwei/article/details/50367761

http://blog.csdn.net/zhangliangzi/article/details/52334964 --不错的aop讲解

Spring用代理类包裹切面，把它们织入到Spring管理的bean中。也就是说代理类伪装成目标类，它会截取对目标类中方法的调用，让调用者对目标类的调用者都先变成调用伪装类，伪装类中就先执行了切面，在把调用转发给真正的目标bean。

Spring提供了两种方式来生成代理对象: JDKProxy和Cglib，具体使用由配置决定（注意两种区别:前者目标方法需要有接口实现，后者不需要）

-1）使用AOP的几种方式：

经典的基于代理的AOP

@AspectJ注解驱动的切面

纯POJO切面

注入式AspectJ切面

### AOP注解

SpringAOP面向切面编程，可以用来配置事务、日志、token校验、权限验证、在用户请求时做一些处理等等。

1.声明切面

@Aspect //将类声明为一个切面

@Component

public class LogAspect {

private static final Logger logger = LoggerFactory.getLogger(LogAspect.class); //日志对象

private static final ObjectMapper OBJECT\_MAPPER = new ObjectMapper(); //JSON对象

}

2.定义切点

private final String POINT\_CUT = "execution(public \* com.yjy.springboot.controller.\*.\*(..))";

@Pointcut(POINT\_CUT) //切入点，包括两部分：表达式、方法签名。方法签名必须是public及void型

public void pointCut(){}

@Pointcut("execution(\* \*..get\*(Long,..))") //所有get开头的方法

3.五个通知增强

@Before --前置增强，在切点方法之前执行

@After --final增强，不管是抛出异常或者正常退出都会执行

@AfterReturning --后置增强，在方法正常退出&正常返回后执行，抛出异常时不执行

@AfterThrowing --异常抛出增强，在抛出异常时执行

@Around --环绕增强，用于编写包裹业务模块执行的代码，第一个参数类型为ProceedingJoinPoint

执行顺序：@Around start -> @Before -> @Around end -> @After -> @AfterReturning

连接点(JoinPoint)相关方法：

getThis() --aop代理对象

getTarget() --被代理对象

getSignature() --连接点的方法签名对象，例如：Result com.yjy.test.controller.UserController.testData(Integer,String)

getArgs() --获取目标方法参数

ProceedingJoinPoint point

Object[] args = point.getArgs(); //

Object returnValue = point.proceed(args); //得到返回结果

### Spring AOP实现

SpringAOP主要用的是动态代理，也就是代理模式。

SpringAOP使用了两种代理机制：一种是基于JDK的动态代理，一种是CGLib的动态代理。

SpringAOP是纯java实现，它在运行期通过代理方式向目标类织入增强代码，它更侧重于提供一种和SpringIOC容器整合的AOP实现。在Spring中，我们可以无缝地将AOP、IOC、AspectJ整合在一起。

分类（2种）：jdk动态代理和CGLib动态代理

-1）有接口时：

jdk动态代理主要通过实现InvocationHandler接口的invoke()方法和使用Proxy.newProxyInstance方法来实现的。

-2）无接口时：

CGLib采用底层的字节码技术，为一个类创建子类，并在子类中采用方法拦截器的结束拦截所有父类方法的调用，并顺势织入。主要通过实现MethodInterceptor接口的getProxy()方法和intercept()方法。

#### JDK动态代理

jdk动态代理主要涉及java.lang.reflect包下的两个类：Proxy类和InvocationHandler接口

（1）InvocationHandler接口说明：

public interface InvocationHandler {

//proxy-被代理的对象，method-要调用的方法，args-方法调用时所需的参数

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable;

}

（2）Proxy类的newProxyInstance静态方法说明：

// loader-类加载器，interfaces-得到全部接口，h-得到InvacationHandler接口的子类实例

public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h) throws IllegalArgumentException

测试实例：

--1）编写业务接口及其实现类

public interface UserService {

void removeUser(int userId);

}

public class UserServiceImpl implements UserService {

public void removeUser(int userId) throws Exception {

System.out.println("模拟删除用户：" + userId);

Thread.currentThread().sleep(200);

}

}

--2）横切逻辑代码

public class TestAspect {

public void before() {

System.out.println("before: " + new Date());

}

public void afterReturning() {

System.out.println("afterReturning: " + new Date());

}

public void afterFinally() {

System.out.println("afterFinally: " + new Date());

}

public void afterThrowing() {

System.out.println("AfterThrowing: " + new Date());

}

}

--3）二者关联（将横切代码织入业务代码中）

public class AspectHandler implements InvocationHandler {

private Object target; // target为目标业务类

public AspectHandler(Object target) { //构造器

this.target = target;

}

public Object invoke(Object arg0, Method arg1, Object[] arg2) throws Throwable {

Object object = null;

try {

before(); //1

object = arg1.invoke(target, arg2);// 通过反射机制调用目标对象的方法

afterReturning(); //2

return obj;

} catch(Exception e) {

AfterThrowing(); //3

} finally {

afterFinally(); //4

}

return object;

}

}

--4）测试

public class Main {

public static void main(String[] args) {

UserService target = new UserServiceImpl(); // 被代理对象

AspectHandler handler = new AspectHandler(target); //处理器

// 创建代理实例

UserSer proxy = (UserSer) Proxy.newProxyInstance(

target.getClass().getClassLoader(),

target.getClass().getInterfaces(),

handler);

proxy.removeUser(3);

}

}

输出：

before..

模拟删除用户：3

afterReturning...

afterFinally...

#### CGLib动态代理

采用底层的字节码技术，为一个类创建子类，并在子类中采用方法拦截器的结束拦截所有所有父类方法的调用，并顺势织入。

getProxy(Class clazz)：为一个类创建动态代理对象，该代理对象通过扩展clazz创建代理对象。在这个代理对象中，我们织入性能监视的横切逻辑.

intercept(Object obj,Method method,Object[] args,MethodProxy proxy)：是CGLib定义的Intercept接口的方法，它拦截所有目标类方法的调用，obj：目标类的实例；method：目标类方法的反射对象；args：方法的动态入参；proxy：代理类实例。

测试实例：

--1）编写代理类

public class CglibProxy implements MethodInterceptor{

private Enhancer enhancer = new Enhancer();

public Object getProxy(Class clazz){

enhancer.setSuperclass(clazz); //设置需要创建子类的类

enhancer.setCallback(this);

return enhancer.create(); //通过字节码技术动态创建子类实例

}

public Object intercept(Object obj,Method method,Object[] args,MethodProxy proxy) throws Throwable{// 拦截父类所有方法的调用

PerformanceMonitor.begin(obj.getClass().getName()+"."+method.getName());

Object result = proxy.invokeSuper(obj,args); //通过代理类调用父类中的方法

PerformanceMonitor.end();

return result;

}

}

--2）测试

public class TestForumService{

public static void main(String[] args){

CglibProxy proxy = new CglibProxy();

UserService userService = (UserServiceImpl)proxy.getProxy(UserServiceImpl.class);

forumService.removeForum(10);

forumService.removeTopic(1023);

}

}

### 相关问题

什么是 AOP？

point cut，advice，Join point是什么？

join point 和 point cut 的区别？

怎么理解面向切面编程的切面？

谈谈对SpringAOP Weaving（织入）的理解？

谈谈SpringAOP Introduction（引入）的理解？

讲解OOP与AOP的简单对比？

讲解JDK 动态代理和 CGLIB 代理原理以及区别？

讲解Spring 框架中基于 Schema 的 AOP 实现原理？

讲解Spring 框架中如何基于 AOP 实现的事务管理？

## 事务管理

相关链接：

http://www.mamicode.com/info-detail-1248286.html

### 概述

定义：事务是将一系列的动作综合在一起，组成一个完整的工作单元。这些动作必须全部完成，如果有一个失败则事务回滚到最开始的状态，彷佛什么都没发生一样。

事务的4个特性：

-1）原子性（Atomicity）：事务是一个原子操作，由一系列动作组成。确保动作要么全部完成，要么完全不起作用。

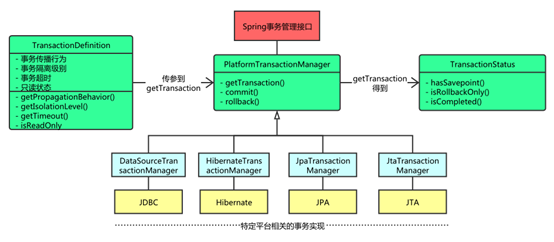
-2）一致性（Consistency）：一旦事务完成(失败或成功)，系统必须确保它所建模的业务处于一致的状态，而不会是部分完成部分失败。

-3）隔离性（Isolation）：一个事物的执行，不受其他事务的干扰，即并发执行的事物之间互不干扰。

-4）持久性（Durability）：一个事物一旦提交，它对数据库的改变就是永久的。

### Spring事务管理接口

事务管理接口图：



事务管理高层抽象接口（3个）：

（1）PlatformTransactionManager --事务管理器

描述：Spring并不直接管理事务，而是提供了多种事务管理器，它们将事务管理的职责委托给Hibernate或MyBatis等持久化机制锁提供的相关平台的事务来实现。通过该接口，Spring为各平台（如JDBC、Hibernate等）提供了对应的事务管理器。

接口如下：

public interface PlatformTransactionManager()...{

// 由TransactionDefinition得到TransactionStatus对象

TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;

void commit(TransactionStatus status) throws TransactionException; // 提交

void rollback(TransactionStatus status) throws TransactionException; // 回滚

}

Spring为不同框架提供了不同的PlatformTransactionManager接口的实现类：

1）org.springframework.jdbc.datasource.DataSourceTransactionManager --使用Spring JDBC或MyBatis进行持久化数据时使用

2）org.springframework.orm.hibernate4.HibernateTransactionManager --使用Hibernate4.0版本进行持久化数据时使用

3）org.springframework.orm.jpa.JpaTransactionManager --使用jpa进行持久化数据时使用

4）jdo的持久化机制

5）jta，在一个事务跨越多个资源时必须使用

（2）TransactionDefinition --事务定义信息（传播行为、隔离级别、只读、超时）

事务属性5个方面：传播行为、隔离级别、回滚规则、事务超时、是否只读

接口如下：

public interface TransactionDefinition {

int getPropagationBehavior(); // 返回事务的传播行为

int getIsolationLevel(); // 返回事务的隔离级别，事务管理器根据它来控制另外一个事务可以看到本事务内的哪些数据

int getTimeout(); // 返回事务必须在多少秒内完成

boolean isReadOnly(); // 事务是否只读，事务管理器能够根据这个返回值进行优化，确保事务是只读的

}

（3）TransactionStatus --事务具体运行状态

### 传播行为（7种）

描述：方法A在事务中运行中调用了方法B，那么方法B可能继续在现有事务中运行，也可能开启一个新事务并在自己的事务中运行。

7种类型描述：

-1）propagation\_required:表示当前方法必须运行在事务中。若不在则启动新的事务。

-2）propagation\_supports:表示当前方法不需要事务上下文。事务有无都不影响。

-3）propagation\_mandatory:表示该方法必须在事务中运行。若不在则抛出异常。

-4）propagation\_required\_new:在该方法执行期间，当前事务会被挂起。

-5）propagation\_not\_supported:表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。

-6）propagation\_never:表示当前方法不应该运行在事务上下文中。若存在当前事务时会抛出异常。

-7）propagation\_nested:表示如果当前已经存在一个事务，那么该方法将会在嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。

实例操作：

methodA() { //事务属性 PROPAGATION\_REQUIRED

methodB();

}

methodB(){ //事务属性 PROPAGATION\_REQUIRED

... //省略

}

--1）单独调用methodB()方法时：

Spring保证在methodB方法中所有的调用都获得到一个相同的连接。在调用methodB时，没有一个存在的事务，所以获得一个新的连接，开启了一个新的事务。

Main {

Connection con = null;

try{

con = getConnection();

con.setAutoCommit(false);

methodB(); //调用methodB

con.commit();

} catch(RuntimeException e) {

con.rollback();

} finnally {

closeCon();

}

}

--2）单独调用methodA时，又会调用methodB：

调用MethodA时，环境中没有事务，所以开启一个新的事务。当在MethodA中调用MethodB时，环境中已经有了一个事务，所以methodB就加入当前事务。

Main {

Connection con = null;

try {

con = getConnection();

methodA();

con.commit();

} catch(RuntimeException e) {

con.rollback();

} finally {

closeCon();

}

}

### 隔离级别（mysql-4种、oracle-2种）

事务在并发操作种可能出现脏读、不可重复读、幻读问题：

-1）脏读：指一个事务读取了另一个事务未提交的数据。操作流程：改(T1改两条数据)->读(T2在改完第一条数据后读取)

-2）不可重复读：指一个事务范围内多次查询却返回不同的数据值。例如事务T1在读取某一数据，而事务T2立马修改了这个数据并且提交，事务T1再次读取该数据后就得到了不同的结果。操作流程：读(T1)->改(T2提交)->读(T1 前后结果不一样)

-3）幻读（虚读）：事务非独立执行时发生的一种现象。操作流程：改(T1)->插(T2插入与修改前一致的数据)->读(T1觉得数据还未修改一样)

MySQL提供4种隔离级别：

-1）Serializable(串行化)：最高级别，可避免脏读、不可重复读、幻读问题。

-2）Repeatable read(可重复读)：默认，将读取的数据加锁，明确数据读取出来就是为了更新用的，所以要加一把锁，防止别人修改它。可避免脏读、不可重复读问题。

-3）Read committed(读已提交)：能够读取已提交的数据。可避免脏读问题。

-4）Read uncommitted(读未提交)：能够读取未提交的数据。最低级别，任何情况都无法保证。

Oracle数据库只支持Serializable(串行化)和Read committed(读已提交，默认)两种级别。

### 事务配置方式（3种）

分类：

编程式事务：在代码层直接定义事务，能精确定义事务边界

声明式事务：基于aop，有利于将操作和事务规则解耦

（1）编程式事务（原始）

略

（2）基于AspectJ的XML配置

--1）配置事务管理器(jdbc/MyBatis)

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

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

</bean>

--2）配置事务通知(Advise)

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

<tx:attributes>

<tx:method name="insert\*" propagation="REQUIRED" read-only="false" rollback-for="java.lang.Exception" />

<tx:method name="update\*" propagation="REQUIRED" read-only="false" rollback-for="java.lang.Exception" />

<tx:method name="delete\*" propagation="REQUIRED" read-only="false" rollback-for="java.lang.Exception" />

<tx:method name="find\*" propagation="SUPPORTS" />

</tx:attributes>

</tx:advice>

--3）通过aop完成事务切入

<aop:config>

<!-- 配置切入点 -->

<aop:pointcut id="pointcut1" expression="execution(public \* com.yjy.service.\*.\*(..))" />

<!-- 配置切面(通知+切入点) -->

<aop:advisor advice-ref="txAdvice" pointcut-ref="pointcut1"/>

</aop:config>

（3）注解形式

--1）开启注解

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

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

</bean>

<tx:annotation-driven transaction-manager="transactionManager" />

--2）在类或方法上加入注解

// propagation:传播行为(7种) isolation:隔离级别(4种) readOnly:只读 rollbackFor:发生哪些异常回滚

@Transactional(propagation=Propagation.REQUIRED, isolation=Isolation.REPEATABLE\_READ, readOnly=true)

### 相关问题

问1：@Transactional注解能否加在private、protected方法上？

不能。@Transactional注解只能应用到public方法上

问2：为什么抛出了异常，事务却没有回滚？

Spring的事务机制是只在发生未被捕获的RuntimeException时才会回滚；

在service层方法中有try-catch代码块，且里面有多个方法时，当后一个方法抛出异常时，前面已执行了的方法也是不会回滚的，这里需要在catch中处理完逻辑后手动抛出throw new RuntimeException()，这样前面已执行的方法也会回滚。

总结：

抛出RuntimeException。自定义异常是直接继承Exception时，需要把它改成继承自RuntimeException类。

针对抛出Exception，在事务声明中添加@Transactional(rollbackFor=Exception.class)

问3：spring事务是如何保证线程安全的？

事务与线程安全有联系的？

问4：事务的传播属性作用，嵌套事务到底优势怎么回事？

问5：怎样使用注解式事务？

<tx:advice /> <aop:config>标签创建事务切面。

## 拦截器（Interceptor）

拦截器(Interceptor)和过滤器(Filter)区别:

层级关系（由外到里）：Tomcat容器-->过滤器(Filter)-->Servlet容器-->拦截器(Interceptor)-->Controller

-1）Filter是Servlet容器支持，仅用于web环境；Interceptor是Spring容器支持，能使用Spring里的任何资源（如service对象、事务等）。

-2）Filter只在Servlet前后起作用；而拦截器能够深入到某个方法前后、异常抛出前后等，因此拦截器的使用具有更大的弹性。

-3）在Spring架构中，优先使用拦截器。

过滤器Filter：

在请求进入tomcat容器后、进入servlet之前进行预处理；在在servlet处理完后，返回给前端之前进行后处理。

因此过滤器的doFilter(ServletRequest request, ServletResponse response, FilterChain chain)的入参是ServletRequest ，而不是httpservletrequest。因为filter是在httpServlet之前。

# SpringMVC介绍

## 相关类及接口

基础接口：https://www.cnblogs.com/baiduligang/p/4247164.html

SpringMVC的核心及三大组件：

1）核心：DispatcherServlet：核心控制器

--接收http请求

2）三大组件：

--1）HandlerMapping：接口，处理器映射器

--根据请求url，找到处理本次请求的处理器handler，将url和controller以<key, Object>关联起来。

// Determine handler for the current request 对当前请求决定交给哪个handler

mappedHandler = getHandler(processedRequest);

主要实现类：

①SimpleUrlHandlerMapiing --通过xml文件，把URL映射到Controller类上

②DefaultAnnotationHandlerMapping --通过注释，把URL映射到Controller类上

--2）HandlerAdapter：接口，处理器适配器

--对映射器查找的controller中的方法进行调用，返回ModelAndView

// mappedHandler.getHandler()即是我们的Controller

HandlerAdapter ha = getHandlerAdapter(mappedHandler. getHandler());

不同的处理器对应着不同的适配器，但是所有适配器最终都返回ModelAndView对象，SpringMvc通过对ModelAndView对象解析，获得需要响应客户端的数据和视图。

主要实现类：

①AnnotationMethodHandlerAdapter --通过注解，把URL映射到Controller类的方法上

--3）ViewResolver：接口，视图解析器

//mappedHandler.getHandler()即是我们的Controller

ModelAndView mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

主要实现类：

①UrlBasedViewResolver --实现类，通过xml文件把一个视图名交给一个View来处理

②InternalResourceViewResolver --实现类，与上面的类相比加入了JSTL的支持

3）其它组件：

--1）Controller：接口，控制器。

用来分发跳转页面。

添加了@Controller注解注解的类就可以担任控制器（Action）的职责，因此我们并没有用到这个接口。

--2）HandlerInterceptor：接口，拦截器

我们自己实现这个接口来完成相应的拦截器工作。

--3）View：接口

①JstlView --实现类

--4）LocalResolver：接口

--5）HandlerExceptionResolver：接口，异常处理

①SimpleMappingExceptionResolver --实现类

--6）ModelAndView：类

## 执行流程

相关链接：https://blog.csdn.net/zuoluoboy/article/details/19766131/

流程描述：

1）客户端发送请求url，核心控制器DispatcherServlet接收用户请求，并将其解析得到请求资源标识符uri。

2）核心控制器根据该uri，找到对应处理类handler（即Controller）及相应拦截器，并将1个处理器和N个拦截器合成一个处理执行链（HandlerExecutionChain）返回给核心控制器。

3）核心控制器拿着执行链寻找对应的处理器适配器（HandlerAdapter）。由适配器调用指定的handler(处理器)，最后将得到ModelAndView 对象返回给核心控制器。

4）核心控制器将返回的ModelAndView给视图解析器（ViewResolver），根据ModelAndView中设置的View解析具体视图，将该视图返回给核心控制器。

5）核心控制器将解析的结果进行渲染和解析，最终将结果相应到客户端。

处理器适配器准备执行handler细节：

1）提取request中的模型数据，填充执行链入参，开始执行Handler（Controller）。

2）Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象

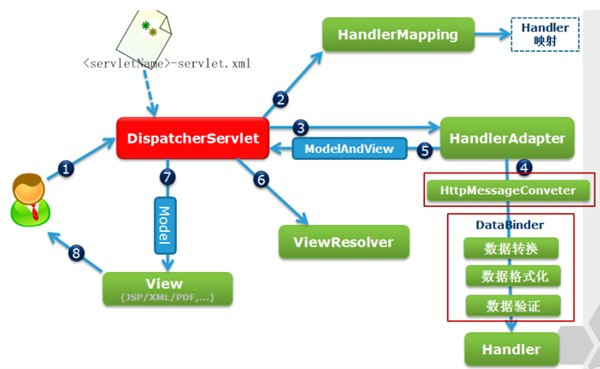
在入参过程中，Spring会做一些额外工作：

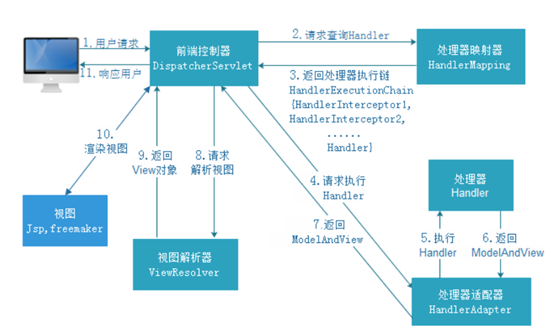
HttpMessageConveter：将请求消息（如Json、xml等数据）转换成一个对象，将对象转换为指定的响应信息

数据转换：对请求消息进行数据转换。如String转换成Integer、Double等

数据格式化：对请求消息进行数据格式化。如将字符串转换成格式化数字或格式化日期等

数据验证： 验证数据的有效性（长度、格式等），验证结果存储到BindingResult或Error中





## SpringMVC的默认配置

在DispacherServlet.properties中配置：

核心控制器（DispatcherServlet）加载处理器映射器（HandlerMapping）、适配器（HandlerAdapter）、视图解析器（ViewResolver）等组件。

web.xml文件配置说明：

-1）ContextLoaderListener说明：

--加载ioc容器

-2）spring的8个监听器6个事件

RequestContextListener说明：

RequestContextListener implements ServletRequestListener

ServletRequestListener 监听HTTP请求事件，Web服务器接收的每次请求都会通知该监听器。

-3）CharacterEncodingFilter说明：

--设置项目的编码格式

-4）OpenEntityManagerInViewFilter说明：

--如果没使用OpenEntityManagerInViewFilter，session会在service.find()方法后就被关闭，用了以后session在整个view层结束后才关闭。

### 配置核心控制器

--web.xml配置SpringMVC的servlet：

<servlet>

<servlet-name>springmvc</servlet-name>

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

<init-param>

<description>加载SpringMVC配置</description>

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

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

</init-param>

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

</servlet>

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

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

</servlet-mapping>

### 映射器和适配器配置：

--映射器(HandlerMapping)和适配器(HandlerAdapter)配置：

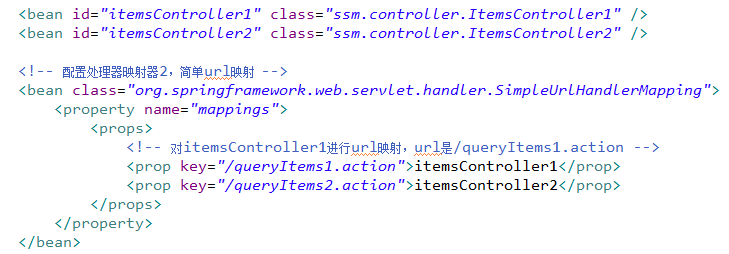
（1）非注解方式：

--1）处理器映射器：

--BeanNameUrlHandlerMapping根据bean的name作为url进行查找

C://Users/user/AppData/Local/YNote/data/qq40776862BDB29BFBF77CFF6728DF2B3F/b352b32442e748bf84017a1e890bbfdd/clipboard.png

--SimpleUrlHandlerMapping直接配置url指定一个Handler



--2）处理器适配器：

--SimpleControllerHandlerAdapter执行实现了Controller接口的Handler

C://Users/user/AppData/Local/YNote/data/qq40776862BDB29BFBF77CFF6728DF2B3F/5124c92e58f944b7b9311872ab95f5ad/clipboard.png

--HttpRequestHandlerAdapter执行实现了HttpRequestHandler接口的Handler

C://Users/user/AppData/Local/YNote/data/qq40776862BDB29BFBF77CFF6728DF2B3F/08b1df9194204e12bbc76a8c6361d89e/clipboard.png

（2）注解方式：

--spring3.1之前

org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping --映射器

org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter --适配器

--spring3.1之后

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping --映射器

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter --适配器

（3）优化配置：

<!-- 1.扫描包 -->

<context:component-scan base-package="com.yjy.controller">

<!-- 2.加载注解的映射器和适配器 -->

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

--mvc:annotation-driven默认加载很多参数绑定方法，比如json转换就默认加载了。

扫描包配置：

1.spring-commom.xml：

<context:component-scan base-package="com.zsoft">

<context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller" />

<context:exclude-filter type="annotation" expression="org.springframework.web.bind.annotation.ControllerAdvice" />

</context:component-scan>

2.spring-mvc.xml:

<context:component-scan base-package="com.zsoft" use-default-filters="false">

<context:include-filter type="annotation" expression="org.springframework.stereotype.Controller" />

<context:include-filter type="annotation" expression="org.springframework.web.bind.annotation.ControllerAdvice" />

</context:component-scan>

### 视图解析器配置：

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

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

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

</bean>

### 配置测试

实例：注解形式配置

--1）在springmvc.xml中配置

<!-- 注解映射器 -->

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"/>

<!-- 注解适配器 -->

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"/>

<!-- 使用mvc:annotation-driven代替上边注解映射器和注解适配器配置 -->

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

--2）开发注解Handler

@Controller

public class UserController {

@RequestMapping("login")

public ModelAndView login(String username, String password) {

ModelAndView mv = new ModelAndView();

if("root".equals(username)) { //登录成功

mv.addObject("username", username);

modelAndView.setViewName("/WEB-INF/jsp/success.jsp");

} else {

mv.addObject("msg", "登陆失败");

modelAndView.setViewName("/WEB-INF/jsp/login.jsp");

}

return mv;

}

}

--3）在springmvc.xml中加载Handler（两种）

<!-- <bean class="com.yjy.controller.UserController" /> -->

<!-- 扫描包 -->

<context:component-scan base-package="com.yjy.controller">

## HandlerMapping（处理器映射器）

相关链接：http://blog.csdn.net/wangbiao007/article/details/50524268

### 为什么引入映射器HandlerMapping？

核心控制器根据该uri，找到对应处理类handler（即Controller）及相应拦截器，并将1个处理器和N个拦截器合成一个处理执行链（HandlerExecutionChain）返回给核心控制器。

HandlerMapping的使用主要分两步：注册和查找

注册：根据配置文件的配置将一个字符串和一个Controller类似Map<key, value>的形式保存，key对应url中某个字段。

查找：HandlerMapping根据url中某个字段，查找Map中Controller类，并将Controller类封装成一个包含了Controller类和一组拦截器的执行链对象handlerExecutionChain。

### 具体查找流程

入口：在DispatchserServlet的doDispatch()方法中

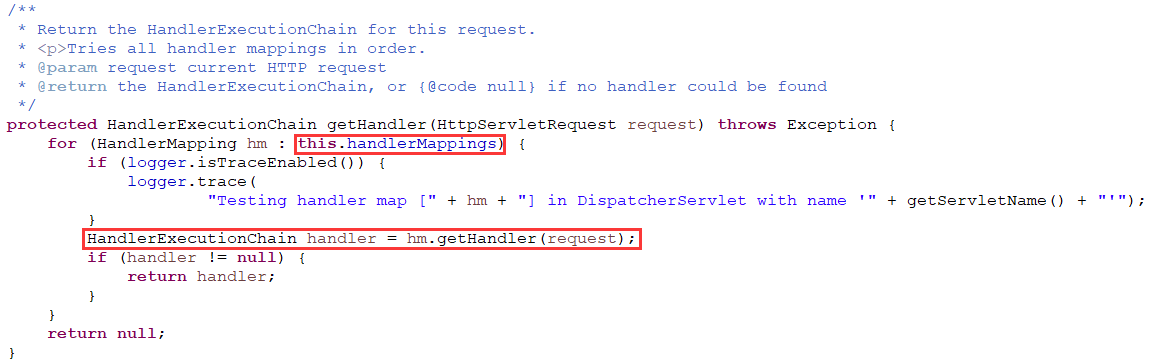
//对当前请求决定交给哪个handler, 当前请求地址过来

//Determine handler for the current request;

HandlerExecutionChain mappedHandler = getHandler(processedRequest);

查看getHandler方法：

AbstractHandlerMapping中的getHandler方法，这个方法的主要作用是根据url找到controller后，并将controller封装成一个执行链（handlerExecutionChain）对象。



handlerMappings属性：表示当前springMVC所支持的处理器映射器的集合。这里有N个处理器映射器，但处理客户端请求的处理器只有一个，就是handler处理器。

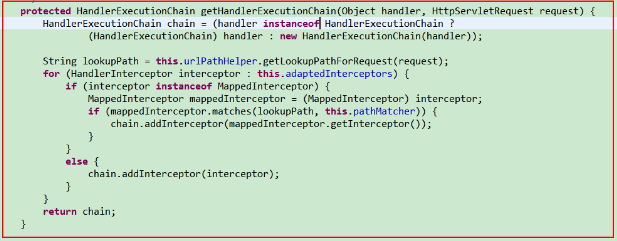


查看getHandler()方法：



查看getHandlerExecutionChain()方法：

由图可知， HandlerExecutionChain中只包含一个handler，并包含N个拦截器。



判断handler：

判断handler是不是执行链，如果是则把handler强制转换成执行链；不是则new一个执行链并把handler赋给执行链。

遍历所有拦截器，并添加进执行链中：

adaptedInterceptors属性：存放所有拦截器的集合

-1）若属于MappedInterceptor类型，则先强转类型，然后和查找的路径做匹配，匹配的添加到执行链的拦截器中；

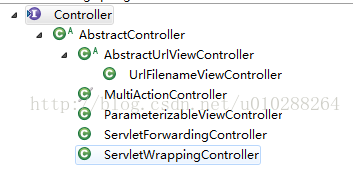
-2）若不属于MappedInterceptor类型则直接添加到执行链的拦截器中。

## HandlerAdapter（处理器适配器）

相关链接：http://blog.csdn.net/u010288264/article/details/53835185

### 为什么引入适配器HandlerAdapter？

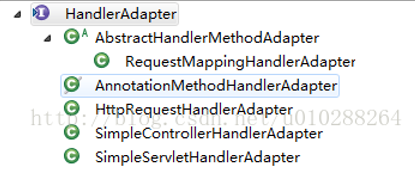
-1）Controller有很多实现子类，它们的实现方式多样，调用方式各不相同。



有图可知，处理器Controller有很多实现子类，它们的实现方式多样，调用方式各不相同。

假如直接调用Controller的方法，需要调用时不停使用ifelse来判断是哪个子类然后执行。这样的做法很不好，违反了开闭原则（对修改关闭，对扩展开放）。

-2）SpringMVC创建一个适配器接口（HandlerAdapter）使得每一种处理器都有一种适配器实现类。



让适配器代替处理器去执行相应的方法。这样在扩展Controller时，只需要增加一个适配器类就完成了SpringMVC的扩展了。

### 接口描述

HandlerAdapter接口如下：

public interface HandlerAdapter {

boolean support(Object handler); //传入处理器，判断是否支持当前适配器

ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception; //代理Controller来执行请求的方法并返回结果

}

SimpleControllerHandlerAdapter是SpringMVC中最常见的一个适配器类：

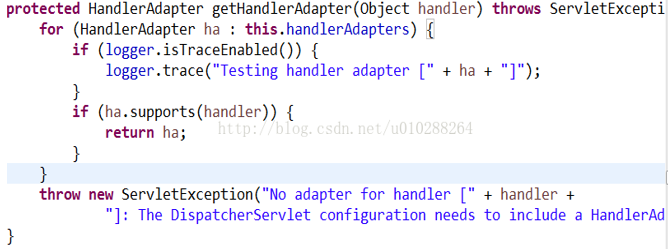


### 使用

-1）获取HandlerAdapter实例：

HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

通过调用DispatcherServlet类中的getHandlerAdapter方法传入Controller，来获取对应的HandlerAdapter实现子类，从而做到使得每一种Controller有一种对应的适配器子类。



-2）通过HandlerAdapter实例执行Controller请求方法，并得到ModelAndView对象

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

### 模拟实例

模拟SpringMVC中的适配器(Adapter):

相关链接：http://blog.csdn.net/bu123dian/article/details/52778870

--1）定义一个Adapter接口

public interface HandlerAdapter {

public boolean support(Object handler);

//模拟HandlerAdapter的handler方法

public void handle(Object handler);

}

--2）定义Controller接口及三种实现

public interfaces Controller {

}

public class HttpController implements Controller {

public void doHttpHandler() {

System.out.println("http...");

}

}

public class SimpleController implements Controller {

public void doSimplerHandler() {

System.out.println("simple...");

}

}

public class AnnotationController implements Controller{

public void doAnnotationHandler(){

System.out.println("annotation...");

}

}

--3）编写适配类（对应上面三种Controller）

public class SimpleHandlerAdapter implements HandlerAdapter {

public void handle(Object handler) {

((SimpleController)handler).doSimpleHandler();

}

public boolean supports(Object handler) {

return (handler instanceof SimpleController);

}

}

public class HttpHandlerAdapter implements HandlerAdapter {

public void handle(Object handler) {

((HttpController)handler).doHttpHandler();

}

public boolean supports(Object handler) {

return (handler instanceof HttpController);

}

}

public class AnnotationHandlerAdapter implements HandlerAdapter {

public void handle(Object handler) {

((AnnotationController)handler).doAnnotationHandler();

}

public boolean supports(Object handler) {

return (handler instanceof AnnotationController);

}

}

--4）模拟一个DispatcherServlet

public class DispatchServlet {

//初始化

public static List<HandlerAdapter> handlerAdapters = new ArrayList<HandlerAdapter>();

public DispatchServlet(){

handlerAdapters.add(new AnnotationHandlerAdapter());

handlerAdapters.add(new HttpHandlerAdapter());

handlerAdapters.add(new SimpleHandlerAdapter());

}

public void doDispatch(){

//此处模拟SpringMVC从request取handler的对象，仅仅new出，可以出，

//不论实现何种Controller，适配器总能经过适配以后得到想要的结果

//HttpController controller = new HttpController();

//AnnotationController controller = new AnnotationController();

SimpleController controller = new SimpleController();

//得到对应适配器 ！！

HandlerAdapter adapter = getHandler(controller);

//通过适配器执行对应的controller对应方法 ！！

adapter.handle(controller);

}

public HandlerAdapter getHandler(Controller controller){

for(HandlerAdapter adapter: this.handlerAdapters){

if(adapter.supports(controller)){

return adapter;

}

}

return null;

}

public static void main(String[] args){

new DispatchServlet().doDispatch();

}

}

## ViewResolver（视图解析器）

作用：

把handler返回的逻辑视图名称解析未视图View对象。进而通过View对象的视图渲染把最终的结果展现给用户。

View视图渲染的原理，简单说就是把模型数据填充到视图模板，最终交由Servlet的response进行渲染展示。

流程如下：

1）View接口是SpringMVC提供的视图渲染接口，定义了render方法对给定的模型数据进行视图渲染。

2）AbstractView是实现View接口的抽象类，实现了render方法。

3）InternalResourceView继承自AbstractView，并实现renderMergedOutputModel方法。

4）View渲染视图的大致流程结束，也就是Spring MVC基本完成了整个流程，剩下的渲染工作交由Servlet去处理。

## WebDataBinder数据类型转换

作用：WebDataBinder实现将请求字符string到特定属性的转换。

原因：一般的String、int、long会自动绑定到参数，但是自定义的格式spring不知道如何绑定了。

何时：在适配器（HandlerAdapter）调用Controller的时候生效

实现：1.编辑自定义Editor类（继承PropertyEditorSupport，并重写setAsText方法）2.将自定义Editor类注册到WebDataBinder对象中，并在该方法上使用@InitBinder注解

1.数据转换：

首先表单数据（form表单）通过WebDataBinder进行绑定到命令对象，内部通过PropertyEditor实现

2.数据验证：

在控制器中的功能处理方法中，需要显示的调用Spring的Validator实现并将错误信息添加到BindingResult对象中

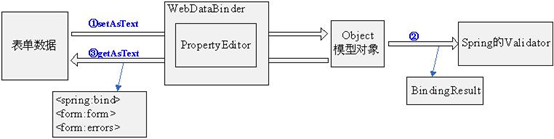
3.格式化显示：

在表单页面可以通过PropertyEditor格式化的数据和错误信息

### 流程

在SpringMVC环境中，数据类型转换、验证及格式化流程通常是：

1. 数据转换：首先表单数据（form表单）通过WebDataBinder进行绑定到命令对象，内部通过PropertyEditor实现
2. 数据验证：在控制器中的功能处理方法中，需要显示地调用Spring的Validator实现并将错误信息添加到BindingResult对象中。
3. 格式化显示：在表单页面可以通过PropertyEditor格式化的数据和错误信息。



### 自定义数据类型转换

实例：

--1）控制器定义

@Controller

public class DataBinderTestController {

@RequestMapping(value = "/dataBind")

public String test(DataBinderTestModel command) {

//输出command对象看看是否绑定正确

System.out.println(command);

model.addAttribute("dataBinderTest", command);

return "bind/success";

}

}

--2）使用WebDataBinder进行控制器级别注册PropertyEditor(控制器独享)

@ControllerAdvice //需要在spring-mvc.xml配置

public class GlobalBinderController {

@InitBinder

public void initBinder(WebDataBinder binder) {

// 统一去除前后空格Editor，如果空串时会设置为null

binder.registerCustomEditor(String.class, new StringTrimmerEditor(true));

// sqlDate的Editor

binder.registerCustomEditor(java.sql.Date.class, new SqlDateEditor());

}

}

--3）自定义Editor（需继承PropertyEditorSupport）

public class SqlDateEditor extends PropertyEditorSupport { //自定义SqlDate编辑器

@Override

public void setAsText(String text) {

if (Strings.isNotBlank(text)) {

setValue(new java.sql.Date(LocalDateTime.parse(text).toDate().getTime()));

} else {

setValue(null);

}

}

}

## @ControllerAdvice注解

### 介绍

大体意思是控制器增强，内部使用@ExceptionHandler、@InitBnder、@ModelAttribute注解的方法应用到所有的@RequestMapping注解的方法上。

实例：

--1）编写自定义测试

@ControllerAdvice

public class ControllerAdviceTest {

@ModelAttribute

public User newUser() {

System.out.println("====应用到所有@RequestMapping注解方法，在其执行之前把返回值放入Model");

return new User();

}

@InitBinder

public void initBinder(WebDataBinder binder) {

System.out.println("====应用到所有@RequestMapping注解方法，在其执行之前初始化数据绑定器");

//参考WebDataBinder章节

}

@ExceptionHandler(UnauthenticatedException.class)

@ResponseStatus(HttpStatus.UNAUTHORIZED)

public String processUnauthenticatedException(NativeWebRequest request, UnauthenticatedException e) {

System.out.println("====应用到所有@RequestMapping注解的方法，在其抛出UnauthenticatedException异常时执行");

return "viewName"; //返回一个逻辑视图名

}

}

--2）在spring-mvc.xml中配置

<context:component-scan base-package="com.sishuok.es" use-default-filters="false">

<context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>

<context:include-filter type="annotation" expression="org.springframework.web.bind.annotation.ControllerAdvice"/> #新加入

</context:component-scan>

## Controller方法的参数解析

相关链接：

https://www.cnblogs.com/fangjian0423/p/springMVC-request-param-analysis.html

### 前言

SpringMVC中Controller的方法参数可以是Integer、Double、自定义对象、ServletRequest、ServletResponse、ModelAndView等等。

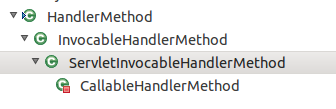
SpringMVC处理请求的流程！！！

HandlerAdapter在内部对于每个请求，都会实例化一个ServletInvocableHandlerMethod进行处理，ServletInvocableHandlerMethod在进行处理时会分列宁部分对请求跟响应进行处理。

### HandlerMethod

HandlerMethod及子类主要用于封装方法调用相关信息，子类还提供调用、参数准备和返回值处理的职责。

接口说明：



HandlerMethod：封装方法定义相关的信息，如类、方法、参数等

使用场景：HandlerMapping时会使用

InvocableHandlerMethod：添加参数准备、方法调用功能

使用场景：执行使用@ModelAttribute注解会使用

ServletInvocableHandlerMethod：添加返回值处理职责，ReponseStatus处理

使用场景：执行http相关方法会使用，比如调用处理执行。

HanlderMethod有两个重要的接口：

HandlerMethodArgumentResolver //请求方法参数的处理

HandlerMethodReturnValueHandler //响应返回值的处理

ServletInvocableHandlerMethod对请求及响应的处理说明：

-1）处理请求时，会根据ServletInvocableHandlerMethod的属性argumentResolvers进行处理。

-2）处理响应时，会根据ServletInvocableHandlerMethod的属性returnValueHandlers进行处理。

SerlvetInvocableHandlerMethod的returnValueHandlers和argumentResolvers这两个属性都是在ServletInvocableHandlerMethod实例化时被赋值的。(使用RequestMappingHandlerAdapter的属性进行赋值)。

RequestMappingHandlerAdapter的argumentResolvers和returnValueHandlers这两个属性是在RequestMappingHandlerAdapter进行实例化的时候被Spring容器注入的。

HandlerMethodArgumentResolver的实现类：

-1）RequestParamMethodArgumentResolver

支持带有@RequestParam注解的参数或带有MultipartFile类型的参数

-2）RequestParamMapMethodArgumentResolver

支持带有@RequestParam注解的参数 && @RequestParam注解的属性value存在 && 参数类型是实现Map接口的属性

-3）PathVariableMethodArgumentResolver

支持带有@PathVariable注解的参数 且如果参数实现了Map接口，@PathVariable注解需带有value属性

-4）MatrixVariableMethodArgumentResolver

支持带有@MatrixVariable注解的参数 且如果参数实现了Map接口，@MatrixVariable注解需带有value属性

-5）RequestResponseBodyMethodProcessor

支持的请求类型是Controller方法参数中带有@RequestBody注解，支持的响应类型是Controller方法带有@ResponseBody注解

-6）ServletRequestMethodArgumentResolver

参数类型是实现或继承或是WebRequest、ServletRequest、MultipartRequest、HttpSession、Principal、Locale、TimeZone、InputStream、Reader、HttpMethod这些类。

-7）ServletResponseMethodArgumentResolver

参数类型是实现或继承或是ServletResponse、OutputStream、Writer这些类

-8）RedirectAttributesMethodArgumentResolver

参数是实现了RedirectAttributes接口的类

# 工具

## 手动获取spring的ApplicationContext和bean对象

描述：当一个类拿到了ApplicationContext对象，就可以方便地获取ApplicationContext中的所有bean。

使用步骤：

-1）定义工具类，实现ApplicationContextAware接口，并重写setApplicationContext方法

public class SpringContextUtils implements ApplicationContextAware {

private static ApplicationContext applicationContext;

/\*\*

\* 实现ApplicationContextAware接口, 注入Context到静态变量中.

\*/

public void setApplicationContext(ApplicationContext context) throws BeansException {

SpringContextUtils.applicationContext = context;

}

// 取得静态变量中的applicationContext

public static ApplicationContext getApplicationContext() {

checkApplicationContext();

return applicationContext;

}

// 获取bean

@SuppressWarnings("unchecked")

public static <T> T getBean(String name) {

checkApplicationContext();

return (T) applicationContext.getBean(name);

}

// 获取bean

public static <T> T getBean(Class<T> requiredType) {

checkApplicationContext();

return applicationContext.getBean(requiredType);

}

private static void checkApplicationContext() {

if (applicationContext == null) {

throw new IllegalStateException("applicationContext未注入，需在xml文件配置SpringContextUtils bean或添加@Conponent注解");

}

}

}

-2）在Spring配置文件中注册该工具类

<bean id="springContextUtils" class="com.yjy.core.util.SpringContextUtils" />

--或者直接在SpringContextUtils类上添加@Component注解

附：仅添加@Component可能会报错，还需要添加@Lazy(false)注解

-3）使用

UserService userService = SpringContextUtils.getBean(UserService.class);

User user = userService.getUserByUsername(username);

## 手动获取request、response

原理：RequestContextListener实现了ServletRequestListener，在其覆盖的requestInitialized(ServletRequestEvent requestEvent)方法中，将request最终设值到RequestContextHolder中。

--1）代码：

HttpServletRequest req = ((ServletRequestAttributes)RequestContextHolder.getRequestAttributes()).getRequest();

HttpServletResponse resp = ((ServletWebRequest)RequestContextHolder.getRequestAttributes()).getResponse();

--2）在web.xml配置：

<listener>

<listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>

</listener>

## 自定义参数解析

描述：自定义HandlerMethodArgumentResolver，使得从页面传进来的user.id, user.name这些字段自动赋值到user对象中。

### HandlerMethodArgumentResolver接口

HandlerMethodArgumentResolver接口描述：

public interface HandlerMethodArgumentResolver {

boolean supportsParameter(MethodParameter parameter);

Object resolveArgument(MethodParameter parameter, ModelAndViewContainer mavContainer, NativeWebRequest webRequest, WebDataBinderFactory binderFactory) throws Exception;

}

接口说明：

-1）supportsParameter(MethodParameter parameter)：用于判定是否需要处理该参数。true为需要，并会去调用resolveArgument()方法false为忽略跳过。

-2）resolveArgument(…)：真正用于处理参数分解的方法。返回的Object就是Controller方法上的形参对象。

-3）实现自定义参数解析器需要重写两个方法。

### 使用

具体使用步骤：

-1）新建HandlerMethodArgumentResolver实现类

public class FormBeanArgumentResolver implements HandlerMethodArgumentResolver {

private static final String DEFAULT\_SEPARATOR = ".";

private String separator = DEFAULT\_SEPARATOR;

@Override

public boolean supportsParameter(MethodParameter parameter) {

return parameter.hasParameterAnnotation(FormBean.class);

}

@Override

public Object resolveArgument(MethodParameter parameter, ModelAndViewContainer mavContainer,

NativeWebRequest webRequest, WebDataBinderFactory binderFactory) throws Exception {

//例如：某方法定义为addUser(@FormBean("user") User user)

HttpServletRequest servletRequest = webRequest.getNativeRequest(HttpServletRequest.class);

FormBean formBean = parameter.getParameterAnnotation(FormBean.class);

String prefix = formBean.value();//获取@FormBean("user")中的user

//获取参数类型，返回的是com.yjy.domain.User

Class<?> paramType = parameter.getParameterType();

Object bindObject = BeanUtils.instantiateClass(paramType);

WebDataBinder binder = new WebDataBinder(bindObject, prefix);

PropertyValues pvs = new ServletRequestParameterPropertyValues(servletRequest, prefix, separator);

binder.bind(pvs);

return bindObject;

}

}

-2）新增自定义注解

@Retention(RetentionPolicy.RUNTIME)

@Target({ ElementType.PARAMETER })

@Documented

public @interface FormBean {

String value() default "";

}

-3）注入spring容器

@SpringBootApplication

public class Application extends WebMvcConfigurerAdapter{

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

@Override

public void addArgumentResolvers(List<HandlerMethodArgumentResolver> argumentResolvers) {

super.addArgumentResolvers(argumentResolvers);

argumentResolvers.add(new FormBeanArgumentResolver());

}

}

--或者自定义配置类

@Configuration

public class ApplicationConfigurer extends WebMvcConfigurerAdapter {

@Override

public void addArgumentResolvers(List<HandlerMethodArgumentResolver> argumentResolvers) {

super.addArgumentResolvers(argumentResolvers);

argumentResolvers.add(new TestArgumentResolver());

}

}

-4）测试

@Controller

@RequestMapping("/user")

public class UserController {

@RequestMapping("/addUser")

@ResponseBody

public User addUser(@FormBean("user") User user, @FormBean("role") Role role) {

System.out.println(user);

System.out.println(role);

return user;

}

}

例如：

在addUser(@FormBean("user") User user, @FormBean("role") Role role)方法中

输入：http://localhost:8080/user/addUser?user.id=111&user.name=AAA&role.id=222 &role.name=BBB

输出：User [id=111, name=AAA, type=type1] Role [id=222, name=BBB]

## 页面传参时的日期转换

描述：我们在页面传进来的String类型的日期需要转换成java.sql.Date类型

实现步骤：

-1）新建日期转换类（继承PropertyEditorSupport类，并重写setAsText方法）

public class SqlDateEditor extends PropertyEditorSupport {

@Override

public void setAsText(String text) {

if (Strings.isNotBlank(text)) {

setValue(new java.sql.Date(LocalDateTime.parse(text).toDate().getTime()));

} else {

setValue(null);

}

}

}

-2）配置日期转换类

@ControllerAdvice

public class GlobalBinderController {

@InitBinder

public void initBinder(WebDataBinder binder) {

// 统一去除前后空格Editor，如果空串时会设置为null

binder.registerCustomEditor(String.class, new StringTrimmerEditor(true));

// sqlDate的Editor

binder.registerCustomEditor(java.sql.Date.class, new SqlDateEditor());

}

}

-3）测试使用

public class LoginController {

@RequestMapping(value = "/testDate")

public String method(Date date) {

System.out.println(date);

}

}

## SpringMVC下载

-1）跳转链接：

<a href="./downloadFile/download" >下载</a>

-2）后台代码

@RequestMapping("download")

public ResponseEntity<byte[]> download() throws IOException {

String path="D:\\workspace\\test.xlsx";

File file=new File(path);

HttpHeaders headers = new HttpHeaders();

String fileName=new String("你好.xlsx".getBytes("UTF-8"),"iso-8859-1");//为了解决中文名称乱码问题

headers.setContentDispositionFormData("attachment", fileName);

headers.setContentType(MediaType.APPLICATION\_OCTET\_STREAM);

return new ResponseEntity<byte[]>(FileUtils.readFileToByteArray(file), headers, HttpStatus.CREATED);

}

## Ajax文件上传

--前提：引入jquery.form.js文件

-1）页面

<form id="importForm" action="${ctx}/infe/data/importData" method="post" enctype="multipart/form-data">

选择本地文件:<input type="file" name="sourceFile">

<button class="btn-query" id="importBtn">导入</button>

</form>

-2）js部分

$("#importBtn").click(function() {

$("#importForm").ajaxSubmit({

type: "post",

url: "${ctx}/infe/data/importData",

success: function(data) {

alert(data);

},

error: function(msg) {

alert("文件上传失败");

}

});

});

-3）后台

@RequestMapping(value = "/importData")

public void importData(@RequestParam("sourceFile") MultipartFile sourceFile,

ImportData importData, HttpServletResponse response, Model model) {

try {

PrintWriter out = response.getWriter();

String filename = sourceFile.getOriginalFilename();

if(filename.endsWith(".csv")) {

System.out.println("csv文件");

} else {

out.println("文件不符合要求！");

return;

}

out.println("上传成功！！");

} catch (IOException e) {

e.printStackTrace();

}

}

# 概念相关

## 5种作用域

Spring管理的bean有哪些模式？

singleton(单例模式)：默认，在Spring的Ioc容器中只存在一个对象实例，所有该对象的引用都共享这个实例。

prototype：每次对该bean对象的请求都会创建一个新的实例。

request：针对web应用，每次http请求都会产生新的bean实例。

session：针对web应用，在一个http session中，一个bean定义对应一个bean实例。

global session：针对web应用，在一个全局的http session中，一个bean定义对应一个bean实例。典型情况下，仅在使用portlet context的时候有效。

## Spring如何管理bean、bean的生命周期

Spring如何启动Bean？

-1）首先通过Resource加载配置文件并启动IOC容器

-2）IOC容器通过反射机制实例化Bean并建立bean之间的依赖关系。（Bean就是Spring IOC容器初始化，装配及管理的对象）

-3）然后通过getBean方法获取bean对象，就可以调用它的方法

Spring如何管理Bean？

Spring通过Spring容器来管理bean，一般由BeanFactory或者ApplicationContext充当管理角色。

BeanFactory采用工厂设计模式，负责读取bean配置文档，管理bean的加载，实例化，维护bean之间的依赖关系，负责bean的生命周期。

ApplicationContext除了提供上述BeanFactory所提供的功能之外，还提供了更完整的框架功能：国际化支持、aop、事务等。

BeanFactory在解析配置文件时并不会初始化对象，只有在使用对象getBean()才会对该对象初始化；

bean的生命周期？

--以BeanFactory来维护一个Bean的生命周期为例

1.Bean的建立，由BeanFactory读取Bean定义文件，并生成各个实例。

2.Setter注入，执行Bean的属性依赖注入

3.BeanNameAware的setBeanName()，如果实现该接口，则执行其setBeanName方法

4.BeanFactoryAware的setBeanFactory()，如果实现该接口，则执行其setBeanFactory方法

5.BeanPostProcessor的processBeforeInitialization()，如果有关联的processor，则在Bean初始化之前都会执行这个实例的processBeforeInitialization方法

6.InitializingBean的afterPropertiesSet()，如果实现了该接口，则执行其afterPropertiesSet方法

7.Bean定义文件中定义init-method

8.BeanPostProcessors的processAfterInitialization()，如果有关联的processor，则在Bean初始化之前都会执行这个实例的processAfterInitialization方法

9.DisposableBean的destroy()，在容器关闭时，如果Bean类实现了该接口，则执行它的destroy方法

10.Bean定义文件中定义destroy-method，在容器关闭时，可以在Bean定义文件中使用“destory-method”定义的方法

## Bean的初始化及销毁方法

Spring的初始化和销毁方法每个都有3种实现：1、注解方式 2、接口方式 3、xml配置

初始化（注解实现）：

@PostConstruct

public void preInit() {

System.out.println("初始化");

}

销毁：

@PreDestroy

public void preDestroy() {

System.out.println("销毁方法");

}

servlet启动顺序：

服务器加载servlet->servlet构造函数->PostConstruct->init方法->Service方法->destroy方法->PreDestroy->服务器卸载Servlet完毕

## 应用上下文

通过这些应用上下文的引用，你可以调用应用上下文的getBean()方法从Spring容器中获取Bean。

手动获取spring的bean：参见4.1

几种应用上下文实现：

-1）ClassPathXmlApplicationContext

--从类路径下的xml配置文件中加载上下文定义，把应用上下文定义当作类资源

例:ApplicationContext context=new ClassPathXmlApplication("foo.xml");

-2）FileSystemXmlApplicationContext

--读取文件路径下的XML配置文件并加载上下文定义

例:ApplicationContext context=new FileSystemXmlApplicationContext("D:foo.xml");

-3）XmlWebApplicationContext

--读取web应用下的XML配置文件并装载上下文定义

BeanFactory和ApplicationContext的区别：

## @Resource和@Autowired

-1）@Resource按照ByName注入

-2）@Autowired按照ByType注入；若想按照ByName装配，可结合@Qualifier使用

例如：

-1）service层有两个service实现类

@Service("service")

public class EmployeeServiceImpl implements EmployeeService{

...

}

@Service("service1")

public class EmployeeServiceImpl1 implements EmployeeService{

...

}

-2）controller层

@Controller

public class EmployeeInfoControl {

@Autowired

@Qualifier("service") //区分是哪个实现类

EmployeeService employeeService;

}

## Spring中的设计模式

工厂模式 BeanFactory

适配模式 HandlerAdapter(适配器)

动态代理 AOP

## 零散

参数设置默认值：@RequestParam(value = "page", defaultValue = "0") Integer page

<mvc:view-controller path="/" view-name="redirect:/main" /> --默认跳转链接

该Controller的所有方法在调用前，先执行此@ModelAttribute方法

# 问题待解决

注解相关问题

注解是一种什么样的编程思想？

为何能够直接使用@Autowired进行依赖注入？是如何工作的？

Spring 是如何通过@AutoWired 自动注入 Bean 属性和 Map，List 集合的？

@Required 是如何起到检查xml里面属性有没有被配置的？

Spring 框架是如何把标注@Component 的 Bean 注入到容器？

@Configuration，@ComponentScan,@Import，@Bean 注解是是如何工作的？

使用@PropertySource 引入配置文件，那么配置文件里面的配置是如何被注册到 Spring 环境里面的？

讲解如何通过自定义注解实现一个简单的树形文档生成？