

概述

2018年6月9日 23:03

一、Spring的宗旨

1. Spring是一个**轻量级的开源全栈框架**，目的是简化企业级应用的开发。
2. Spring是一个全栈框架，这体现在企业级应用的每一层架构都可以用Spring的相应技术来实现：
 - (1) 表示层（Web层）——Spring MVC
 - (2) 业务逻辑层（service层）——IoC
 - (3) 数据访问层（DAO层）——JdbcTemplate

二、Spring的两大核心概念

1. 依赖注入（Dependency Injection, DI）

在OOP中，大多数应用程序都是由两个或是更多的类通过彼此的合作来实现业务逻辑，这使得每个对象都需要获取与其合作的对象（也就是它所依赖的对象）的引用。如果这个获取过程要靠自身实现，那么这将导致代码高度耦合并且难以维护和调试。为了降低代码之间的耦合度（这一过程称为解耦），提出了**控制反转**（IoC）的概念。控制反转就是把获得依赖对象的过程反转：不再是主动地获取依赖对象，而是依赖对象自己找上门来。

依赖注入就是实现控制反转的一种主要手段。Class A中用到了Class B的对象b，一般情况下，需要在A的代码中显式地 new 一个 B 的对象。用依赖注入技术之后，A 的代码只需要定义一个私有的B对象，不需要直接 new 来获得这个对象，而是通过相关的容器控制程序来将 B 对象在外部 new 出来并注入到 A 类里的引用中。而具体获取的方法、对象被获取时的状态由配置文件（如 XML）来指定。因为此时 B 对象是 A 的一个属性，所以依赖注入是指向类的属性注入值。

2. 面向切面编程（aspect-oriented programming, AOP）

假设在两个类中，可能都需要在每个方法中做日志。按面向对象的设计方法，那就必须在两个类的方法中都加入日志的内容。也许他们是完全相同的，但就是因为面向对象的设计让类与类之间无法联系，而不能将这些重复的代码统一起来。

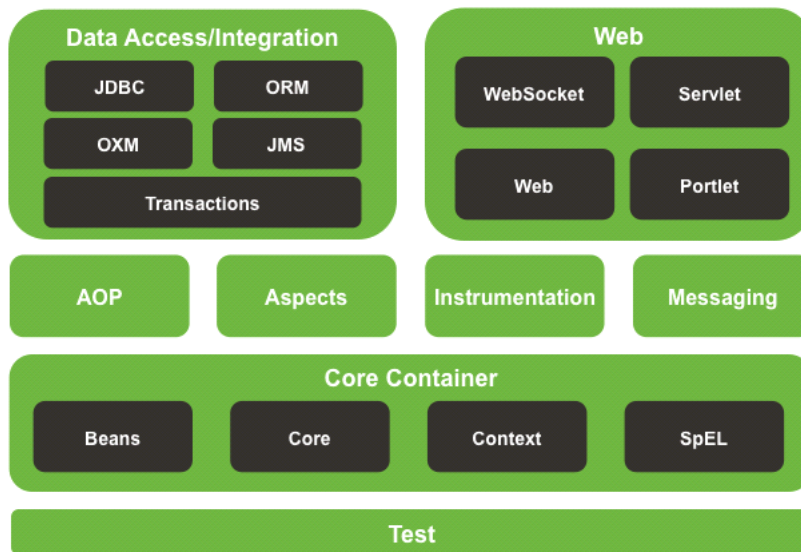
有一种解决办法是将这段代码写在一个独立的类独立的方法里，然后再在这两个类中调用。但是，这样一来，这两个类跟我们上面提到的独立的类就有耦合了，它的改变会影响这两个类。

为了解决这些问题就产生了 **AOP 思想**：在运行时动态地将代码切入到类的指定方法、指定位置上。一般而言，我们管切入到指定类指定方法的代码片段称为切面，而切入到哪些类、哪些方法则叫切入点。有了 AOP，我们就可以把几个类共有的代码，抽取到一个切片中，等到需要时再切入对象中去，从而改变其原有的行为。

三、Spring的构成



Spring Framework Runtime



如果只是利用 Spring 中的基本功能，那么只需导入核心部分的 jar 包（SpEL 是指 Expression Language），但是注意 spring-core 还要依赖于 commons-logging 来实现日志功能（如果想要看到更加详细的信息，就要加上 log4j）。

四、Spring全约束

```
<?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"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd">
</beans>
```

说明：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  //上面两个是基础IOC的约束，必备
  xmlns:context="http://www.springframework.org/schema/context"
  //上面一个是开启注解管理Bean对象的约束
  xmlns:aop="http://www.springframework.org/schema/aop"
  //aop的注解约束
  xmlns:tx="http://www.springframework.org/schema/tx"
  //事务的约束
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd
```

```
//上面一个是开启注解管理Bean对象的约束
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd
//aop的注解约束
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd">
//事务的约束
</beans>
```

IoC

2018年6月9日 23:06

简单地讲，IoC 就是把创建对象这一操作的控制权交给 Spring，让 Spring 自动地进行管理。

在 Spring 中，实现 IoC 主要有两种方法，一种是基于配置文件，另一种是基于注解。

一、IoC 底层原理

1. 实现 IoC 所需的技术：

- (1) XML 配置文件
- (2) dom4j——解析 XML 文件
- (3) 工厂设计模式
- (4) 反射

2. 底层实现的基本步骤

- (1) 创建 XML 配置文件，文件中包含要创建的对象的信息。
- (2) 构造一个工厂类，使用 dom4j 解析 XML 文件。
- (3) 在工厂类中，使用**反射**，根据类名创建对象。——forName、newInstance

二、IoC 操作基本步骤

1. 导入相关 jar 包。

2. 创建一个类。

3. 创建配置文件。

(1) Spring 配置文件的名称和路径不是固定的，但是官方建议把名称设置为“applicationContext.xml”（其实实际开发中会把配置文件命名为 Beans），把文件放在 src 文件夹下。

(2) 在配置文件中引入 schema 约束
schema 约束可以在官方文档中找到：

```
<?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.xsd ">
</beans>
```

(3) 配置对象的创建

```
<bean id="helloworld" class="com.spring.tutorialspoint.HelloWorld">
</bean>
```

一般来说，id 值就是类名的小写，关键是 class 的值不能漏写包名。

三、Spring 的 bean 管理

1. 基于配置文件（XML）

(1) bean 实例化的三种方法

★ 1) 使用类的**无参构造器**

如果类中没有无参构造器，那么就会抛出异常。

```
ApplicationContext context = new
ClassPathXmlApplicationContext("Beans.xml");
//getBean 的参数是 bean 标签的 id 值
HelloWorld obj = (HelloWorld) context.getBean("helloworld");
```

2) 使用静态工厂

a. 写一个静态工厂类

```
public class HelloWorldFactory {
    private static HelloWorld instance = new HelloWorld();

    public static HelloWorld getHelloWorld() {
        return instance;
    }
}
```

b. 进行相关配置，此时 class 属性的值是工厂类

```
<bean id="helloworld"
class="com.spring.tutorialspoint.HelloWorldFactory" factory-method
= "getHelloWorld">
</bean>
```

3) 使用实例工厂

a. 写一个工厂类，其中有个非静态方法

b. 进行相关配置

(2) <bean>标签的常用属性

- 1) id: 属性值不能包含特殊符号
- 2) class: 所要创建的对象类名全称
- 3) name: 功能和 id 属性一样，但是可以包含特殊符号。**已不常用。**
- 4) scope: 指定对象的作用范围，有多个值，常用的有以下两个
 - a. singleton: **默认值**，指定对象是单例的。这里的单例不是说试图创建多个实例时就会抛异常，而是说不管创建对象的代码写了多少次，最终创建出来的对象其实都是同一个。
 - b. prototype: 指定对象是多例的。

```
<bean id="helloworld"
class="com.spring.tutorialspoint.HelloWorldF"
scope="prototype">
</bean>
```

(3) 属性注入

属性注入是说创建实例的时候，为其属性赋值。原生 Java 中属性注入的三种方式：

- 使用 setter
- 使用有参构造器
- 使用接口注入

Spring 只支持前两种方式。

1) Spring 的属性注入

★a. 使用 setter

```
<bean id="helloworld"
class="com.spring.tutorialspoint.HelloWorld">
    <!-- property 标签，name 属性的值就是类属性名称，value 就是相应属性
    值 -->
    <property name="message" value="Hello World!"/>
</bean>
```

当要注入的属性是其他类型的对象时，用 ref 属性：

```
<!-- 注入对象类型的属性 -->
<!-- 配置 UserDao 对象 -->
<bean id = "userDaoInstance" class =
```

```

"com.spring.tutorialspoint.UserDao"></bean>
    <!-- 配置UserService对象 -->
    <bean id="userService" class =
"com.spring.tutorialspoint.UserService">
        <!-- ref的值是要注入的对象的id -->
        <property name = "userDao" ref =
"userDaoInstance"></property>
    </bean>

```

b. 使用有参构造器

```

    <!-- 使用有参构造器属性注入 -->
    <bean id = "helloworldArg" class =
"com.spring.tutorialspoint.HelloWorld">
        <!-- 为有参构造器的参数赋值 -->
        <constructor-arg name = "msg" value =
"HelloWorld"></constructor-arg>
    </bean>

```

c. 使用 p 名称空间

```

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

    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-
        beans.xsd">

    <bean id = "userDaoInstance" class =
"com.spring.tutorialspoint.UserDao"></bean>

    <bean id="userService" class =
"com.spring.tutorialspoint.UserService" p:userDao-ref =
"userDaoInstance">
    </bean>

```

2) 注入复杂类型属性

- a. 数组
- b. List
- c. Map
- d. properties

```

<!-- 注入复杂类型的属性 -->
<bean id = "userDao" class = "com.spring.tutorialspoint.UserDao">
    <!-- 数组 -->
    <property name = "arrs">
        <list>
            <value>Lily</value>
            <value>Lucy</value>
            <value>Tom</value>
        </list>
    </property>

    <!-- List -->
    <property name = "list">
        <list>
            <value>Lily</value>
            <value>Lucy</value>
            <value>Tom</value>
        </list>
    </property>

```

```

        </list>
    </property>

    <!-- Map -->
    <property name = "map">
        <map>
            <entry key = "a" value = "Lily"></entry>
            <entry key = "b" value = "Lucy"></entry>
            <entry key = "c" value = "Tom"></entry>
        </map>
    </property>

    <!-- properties -->
    <property name = "properties">
        <props>
            <prop key = "driverClass">
com.mysql.jdbc.Driver</prop>
            <prop key = "userName">root</prop>
        </props>
    </property>
</bean>

```

2. 基于注解

基于注解的方式是对基于配置文件的方式的一种改进，能使配置文件更简洁，但无法替代配置文件。基于注解的方式需要spring-aop包；除此之外还需要引入新的约束：

```

<?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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- 只会扫描属性的注解，所以一般不用 -->
    <!-- <context:annotation-config/> -->

    <!-- 开启注解扫描：扫描指定包中的所有注解 -->
    <!-- 属性值是要创建的对象所在的包，如果有多个包可以用逗号分隔，也可以只写多个包名的共同部分 -->
    <context:component-scan base-package="com.spring.annotation">
</context:component-scan>

</beans>

```

(1) 创建对象

1) 为将要创建对象的类加上注解 `@Component(value = "instanceName")`，然后就可以通过 `ApplicationContext.getBean` 方法创建对象，`getBean` 方法的参数就是 `instanceName`。

为了让代码可读性更强、程序用途更清晰，Spring 提供了 `@Component` 的三个衍生注解：

- a. `@Controller`——web层
- b. `@Service`——业务层
- c. `@Repository`——持久层

上面提到的四个注解的功能都是一样的，但是 Spring 会对后三个注解的功能进行增强。

2) 指定对象是单例还是多例
`@Scope(value = "prototype")`

(2) 注入属性

假设类 A 中有一个类 B 的对象 b，要为 b 注入值，有两种方法：

1) 为 b 添加注解 @Autowired。



2) 为 b 添加注解 @Resource(name = "ObjectName")，其中 ObjectName 是类 B @Component 注解的 value 值。这一个注解更常用，因为它明确指定了要注入的对象。

这两种方法不需要有 b 的setter。

3. 基于配置文件和基于注解方式的混合使用

用配置文件创建对象，用注解注入属性。

```
<?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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-
                           context.xsd">

    <!-- 扫描注解 -->
    <context:component-scan base-package="com.spring.xmlanno">
</context:component-scan>

    <!-- 配置对象 -->
    <bean id = "userService" class = "com.spring.xmlanno.UserService">
</bean>
    <bean id = "userDao" class = "com.spring.xmlanno.UserDao"></bean>
</beans>

public class UserDao {

    public void add()
    {
        System.out.println("UserDao add...");
    }
}

public class UserService {

    @Resource(name = "userDao")    //name的值是配置文件中相应的bean标签的id
    值
    private UserDao userDao;
    @Resource(name = "orderDao")
    private OrderDao orderDao;

    public void get()
    {
        System.out.println("UserService get..");
        this.userDao.add();
    }
}

public class testService {

    @Test
    public void test() {

        ApplicationContext context = new
        ClassPathXmlApplicationContext("Beans3.xml");
        UserService userService = (UserService)
        context.getBean("userService");
    }
}
```



```
        userService.get();  
    }  
}
```

一、底层原理

采用**动态代理**的方式

二、术语

1. Joint Point（连接点）：指定类中的哪些方法可以动态地进行修改。

★ 2. Pointcut（切入点）：类中有多个可以被动态修改的方法，在实际开发中真正发生了修改的方法称为切入点。

★ 3. Advice（通知/增强）：在类中动态地增加一个新的方法，这个新增加的方法叫做通知。通知分为五种类型（假设类中原有方法 foo，通知为 bar）：

- (1) 前置通知：在执行 foo 之前先执行 bar。
- (2) 后置通知：**正常地**执行完 foo 之后再执行 bar。
- (3) 异常通知：当 foo 出现异常时执行 bar。
- (4) 最终通知：无论 foo 是正常地执行完毕还是出现异常，bar 都会执行。
- (5) 环绕通知：bar 既是前置通知，又是后置通知。

★ 4. Aspect（切面）：把通知应用到具体的方法即切入点中。

5. Introduction（引介）：一种特殊的通知，可以在不修改类代码的前提下在运行期为类动态地添加一些方法或者域。

6. Target（目标对象）：要修改的方法所属的类。

7. Weaving（织入）：把通知应用到具体的目标对象。

8. Proxy（代理）：一个类经过织入后产生一个结果代理类。

三、操作——基于 AspectJ

AspectJ 是一个面向切面的框架，它扩展了 Java 语言，定义了 AOP 语法。Spring 2.0 以后增加了对 AspectJ 的支持，现在通常使用 AspectJ 来进行 Spring 的 AOP 操作。

在使用 AspectJ 之前需要引入其 jar 包（aspectj 和 aspectjweaver），同时还要导入 aop 约束：

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

</beans>
```

★ 1. 使用表达式配置切入点

常用的表达式是 execution 表达式：

execution(modifies-pattern? ret-type-pattern declaring-type-parttern? name--pattern(param-pattern) throws-pattern?)

后面跟着一个问号的部分是可以省略的，其余都要包含。空格不能省略。各个部分的含义是：

- modifies-pattern：指定方法的修饰符。一般都是直接省略。
- ret-type-pattern：指定方法的返回值类型，支持通配符，可以使用*通配符来匹配所有返回值类型。
- declaring-type-parttern：指定方法所属的类的全路径，支持通配符。如果指定了全路径，那么方法名前面要加 .
- name--pattern：指定匹配的方法名，支持通配符，可以使用*通配符来匹配所有方法。

- **parm-pattern**: 指定方法声明中的形参列表, 支持通配符: *和.。() 表示空参, (..) 表示0个或多个参数。(*) 表示任意类型的一个参数, (*, String) 表示第一个参数是任意类型的, 第二个参数必须是String。
- **throws-pattern**: 指定方法声明抛出的异常, 支持通配符。

以下是几个例子:

- //匹配任意public方法的执行。
execution(public * * (..))
- //匹配任意方法名以set开始的方法。
execution(* set* (..))
- //匹配AccountService里定义的任意方法的执行。
execution(* org.hb.AccountService.* (..))
- //匹配Service包中任意类的任意方法的执行。
execution(* org.hb.service.*.*(..))

2. 用XML配置文件实现aop操作

(1) 编写切入点和通知所属的类

```
public class Book {

    //切入点
    public void add()
    {
        System.out.println("Book add...");
    }
}

public class MyBook {

    //前置通知
    public void forwardAdvice()
    {
        System.out.println("Forward Advice...");
    }

    //环绕通知
    public void aroundAdvice(ProceedingJoinPoint pjp) throws Throwable
    {
        //前置通知
        System.out.println("before...");

        //执行被通知的方法
        pjp.proceed();

        //后置通知
        System.out.println("after");
    }
}
```

(2) 创建配置文件

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

<bean id = "book" class = "com.spring.aop.Book"></bean>
<bean id = "myBook" class = "com.spring.aop.MyBook"></bean>

<!-- 配置aop操作 -->
<aop:config>
  <!-- 配置切入点 -->
  <aop:pointcut expression="execution(public * com.spring.aop.Book.add(..))" id="pc1"/>
  <!-- 配置切面 -->
  <aop:aspect ref="myBook"> 这个属性的值是用作通知的方法所属的对象
    <!-- 配置通知类型 -->
    <aop:before method="forwardAdvice" pointcut-ref="pc1"/> 这个属性的值就是前面配置好的切入点的id
  </aop:aspect>
</aop:config>
</beans>
```

说明：

- ①. 切入点所属类和通知所属类都要创建对象。
- ②. 配置切入点时用表达式说明切入点是指哪个方法。
- ③. <aop:aspect> 标签 ref 属性的值是之前配置的通知所属类的对象的 id。
- ④. <aop:before> 配置的是前置通知，其他类型的通知也有相应的标签。method 属性的值是用作通知的方法名，pointcut-ref 就是切入点的 id。

★ 3. 用注解实现 aop 操作（比用配置文件更为简便）

- (1) 在配置文件中开启 aop 操作

```
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

- (2) 在配置文件中配置对象的创建
- (3) 为通知所属类添加注解

```
@Aspect
public class MyBook {

    //环绕通知
    @Around(value = "execution(public * com.spring.aop.Book.add(..))")
    public void aroundAdvice(ProceedingJoinPoint pjp) throws Throwable
    {

        //前置通知
        System.out.println("before...");

        //执行被通知的方法
        pjp.proceed();

        //后置通知
        System.out.println("after");

    }
}
```

通知方法所属类加上注解 @Aspect；通知方法加上相应的注解(比如环绕通知是 @Around)，这个注解有一个 value 属性，属性值是一个指定了切入点的表达式。

持久化技术

2018年6月9日 23:08

Spring 对多种持久化技术（DAO 层）都进行了封装，提供了相应的模板类。

ORM持久化技术	模板类
JDBC	<code>org.springframework.jdbc.core.JdbcTemplate</code>
Hibernate5.0	<code>org.springframework.orm.hibernate5.HibernateTemplate</code>
IBatis(MyBatis)	<code>org.springframework.orm.ibatis.SqlMapClientTemplate</code>
JPA	<code>org.springframework.orm.jpa.JpaTemplate</code>

进行 Spring 的数据库操作需要导入相关的jar包：spring-jdbc 和 spring-tx，以及数据库驱动的 jar 包。

在进行具体的 CRUD 操作之前，还要进行一些准备工作：

1. 建立数据库连接

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("com.mysql.jdbc.Driver");
dataSource.setUrl("jdbc:mysql://localhost:3306/studytest?useSSL = false");
dataSource.setUsername("root");
dataSource.setPassword("loveisKING");
```

2. 创建 JdbcTemplate 对象

```
JdbcTemplate jdbc = new JdbcTemplate(dataSource);
```

完成这些准备之后，就可以进行具体的 CRUD 操作。

一、增加

```
String sql = "insert into user_reg(name,password) values(?,?)";
int rows = jdbc.update(sql, "Lucy", "7890");
```

二、修改和删除

```
//修改
String sql = "update user_reg set password = ? where name = ?";
int rows = jdbc.update(sql, "123456", "Lucy");

//删除
String sql = "delete from user_reg where name = ?";
int rows = jdbc.update(sql, "Lucy");
```

三、查询

1. 查询结果为单值

```
String sql = "select count(*) from user_reg";
int count = jdbc.queryForObject(sql, Integer.class);
```

2. 查询结果为对象

(1) 用 BeanPropertyRowMapper 作为行映射器的实现

```
String sql = "select name,password from user_reg where name = ?";
User user = jdbc.queryForObject(sql,
    BeanPropertyRowMapper.newInstance(User.class), "Lucy");
```

这种方法用起来比较简洁，但是要求：

- Domain 类符合 Java Bean 规范。
- 数据库字段名符合规范，且与 Domain 中的属性名称存在一定的对应关系（如果对不上，可以为查询字段设置别名）。

(2) 自定义行映射器

```
String sql = "select name,password from user_reg where name = ?";
```

//RowMapper 接口只有一个 mapRow 方法，所以可用 lambda 表达式

//注意这是个泛型接口

```
RowMapper<User> rowMapper = (ResultSet rs, int rowNum) -> {
    String name = rs.getString("name");
    String passWord = rs.getString("password");
    User user = new User();
    user.setName(name);
    user.setPassWord(passWord);
    return user;
};
```

```
User user = jdbc.queryForObject(sql, rowMapper, "Lucy");
```

3. 查询结果为集合

```
String sql = "select name,password from user_reg";
List<User> list = jdbc.query(sql,
    BeanPropertyRowMapper.newInstance(User.class));
```

四、c3p0 连接池

1. 使用 c3p0 连接池所需的 jar 包：mchange-commons-java 和 c3p0。
2. 在配置文件中配置连接池

```
<!-- 配置 c3p0 连接池 -->
<bean id = "datasource" class =
    "com.mchange.v2.c3p0.ComboPooledDataSource">
    <!-- 注入属性值 -->
    <property name="driverClass" value = "com.mysql.jdbc.Driver">
    </property>
    <property name="jdbcUrl" value =
        "jdbc:mysql://localhost:3306/studytest?useSSL = false"></property>
    <property name="user" value = "root"></property>
    <property name="password" value = "loveisKING"></property>
</bean>
```

3. 为 JdbcTemplate 注入数据源属性

```
<bean id = "jdbc" class =
    "org.springframework.jdbc.core.JdbcTemplate">
    <!-- 将数据源对象注入JdbcTemplate -->
    <property name = "dataSource" ref = "datasource"></property>
</bean>
```

JdbcTemplate 类中有一个属性 dataSource，而 ComboPooledDataSource 正好是一个数据源对象，所以可以用配置文件的方式进行属性的注入。

五、Spring 的事务管理

进行事务管理的好处就是可以让程序自动地提交、回滚事务等，Spring 提供了两种事务管理方式：

- 编程式
- 声明式

编程式事务管理实现起来比较麻烦，所以实际开发中一般用的是声明式的事务管理。声明式事务管理可以基于 XML 配置文件实现，也可以基于注解实现。

1. 事务管理接口

Spring提供了3个主要的事务管理接口：

- PlatformTransactionManager——事务管理器
- TransactionDefinition——事务定义
- TransactionStatus——事务运行状态

在实际开发中，最常用的是 PlatformTransactionManager，因为事务管理的第一步就是配置事务管理器。Spring 为不同的持久化技术提供了相应的PlatformTransactionManager 实现，对应 Spring JDB C或 iBatis 的实现是

org.springframework.jdbc.datasource.DataSourceTransactionManager。

2. 基于 XML 配置文件的事务管理——使用 AspectJ

编写好相关的业务逻辑层和数据访问层代码之后，只需配置 XML 文件：

```
<!-- 1.配置事务管理器 -->
<!-- 注意：并不需要在程序中显示地设置一个DataSourceTransactionManager对象 -->
<bean id = "transactionManager" class =
"org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name = "dataSource" ref = "dataSource"></property>
</bean>
<!-- 2.配置事务通知 -->
<tx:advice id="transactionAdvice" transaction-
manager="transactionManager">
    <!-- 配置执行相关事务的方法 -->
    <tx:attributes>
        <!-- 可以用通配符 -->
        <tx:method name="transfer"/>
    </tx:attributes>
</tx:advice>
<!-- 3.配置切面 -->
<aop:config>
    <!-- 切入点 -->
    <!-- 这里的切入点其实就是上面配置好的执行事务的方法 -->
    <aop:pointcut expression="execution(*
com.spring.transaction.OrderService.transfer(..)" id="pointcut"/>
    <!-- 切面 -->
    <aop:advisor advice-ref="transactionAdvice" pointcut-
ref="pointcut"/>
</aop:config>
```

★ 3. 基于注解的事务管理——使用 AspectJ

这种方法要简便得多，步骤如下：

(1) 进行 XML 文件的配置。

```
<!-- 1.配置事务管理器 -->
<bean id = "transactionManager" class =
"org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name = "dataSource" ref = "dataSource"></property>
</bean>
```

```
<!-- 开启事务注解 -->
```

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

(2) 为执行相关事务的类添加注解 @Transactional。

概述

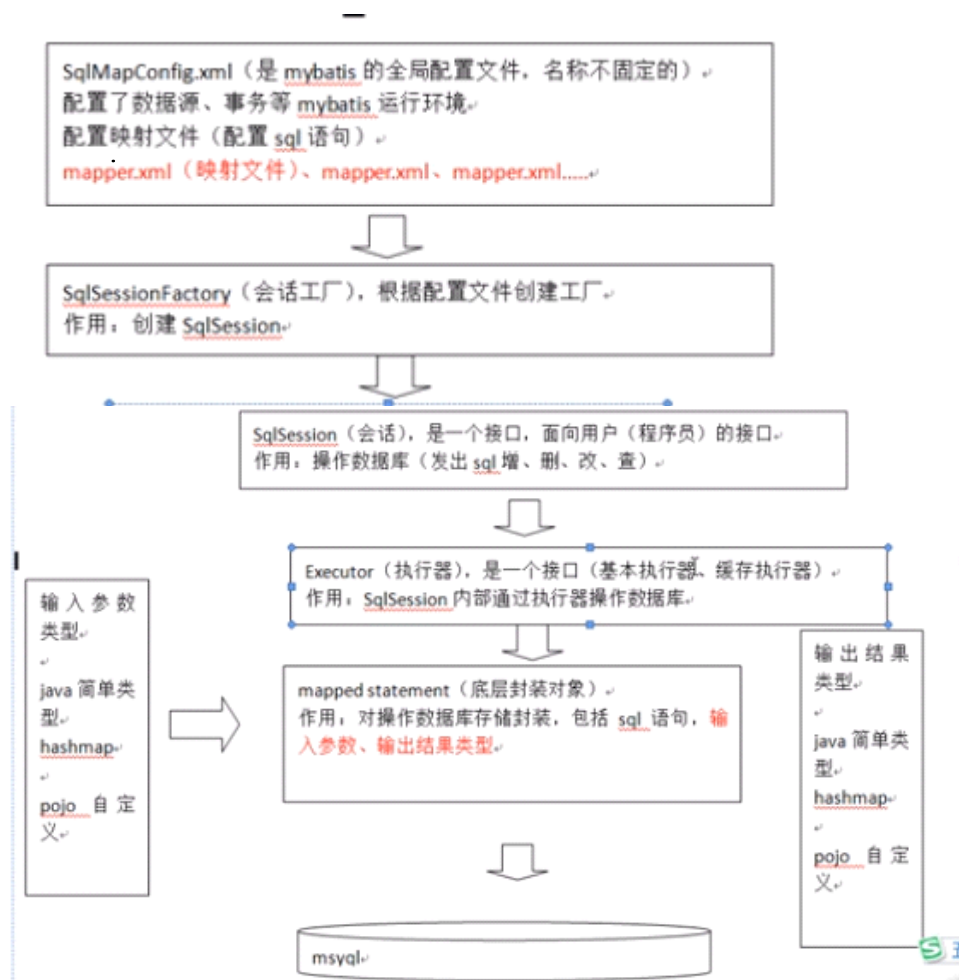
2018年6月22日 14:53

一、原生 JDBC 的问题

1. 需要手动管理数据库连接
2. SQL 语句是硬编码（“写死”）的，如果 SQL 语句变化，就需要重新编译
3. 在 ResultSet 中遍历数据时将表的字段进行硬编码

二、MyBatis 架构

MyBatis 是一个持久层框架，通过它提供的映射方式，可以半自动化地生成 SQL 语句。可以将参数自动填充到 SQL 语句中（输入映射），也可以将查询结果映射成对象（输出映射）。



MyBatis 的工程结构：

- mybatis-3.2.7.jar----mybatis的核心包
- lib----mybatis的依赖包
- mybatis-3.2.7.pdf----mybatis使用手册

三、准备工作

1. 导入核心和依赖 jar 包
2. MyBatis 默认使用 log4j 作为日志工具，在 classpath 下新建一个 properties 文件，相关配置在文档中可以找到：

```
# Global logging configuration
```

```
# DEBUG for development, ERROR or INFO for application
log4j.rootLogger=DEBUG, stdout
# MyBatis logging configuration...
log4j.logger.org.mybatis.example.BlogMapper=TRACE
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

3. 新建全局配置文件 SqlMapConfig.xml，配置的内容也可以在文档中找到：

```
<?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>
    <!-- 和spring整合后 environments配置将废除 -->
    <environments default="development">
        <environment id="development">
            <!-- 使用jdbc事务管理 -->
            <transactionManager type="JDBC" />
            <!-- 数据库连接池 -->
            <dataSource type="POOLED">
                <property name="driver"
value="com.mysql.jdbc.Driver" />
                <property name="url"
value="jdbc:mysql://localhost:3306/mybatis?
characterEncoding=utf-8" />
                <property name="username" value="root" />
                <property name="password" value="loveisKING" />
            </dataSource>
        </environment>
    </environments>

</configuration>
```

4. 新建映射配置文件（最好新建一个子目录）

映射文件用来配置 SQL 语句。映射文件的命名规则一般是 XXXMapper.xml, XXX 就是要查询的表名。也可以按照 iBatis 的命名规则直接命名为 XXX.xml。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- namespace 用来对 SQL 语句进行分类管理，简单理解为隔离 SQL 语句 -->
<mapper namespace="test">
    <!-- 配置 SQL 语句 -->
    <!-- select 标签用来写 SELECT 语句，id 用来标识映射文件中的 SQL -->
    <!-- 因为最后要将 SQL 语句封装成 mappedStatement 对象，所以也称为
statement 的 id -->
    <!-- parameterType 用来指定参数的类型 -->
    <select id="findUserById" parameterType="int"
resultType="mybatis.po.User">
        <!-- #{}是占位符，可以在{}中指定参数的名称 如果参数类型是简单类型，
那么可以指定任意的参数名称 -->
        <!-- resultType 指定了单条查询结果映射成的 Java 对象类型，SELECT
标签指定这个属性 就意味着要将单条记录映射成对象 -->
```

```
        SELECT id,username,birthday,sex,address FROM user WHERE  
        id=#{id}  
    </select>  
</mapper>
```

写完这个文件之后，还要在 SqlMapConfig.xml 中进行加载：

```
<!-- 加载映射文件 -->  
<mappers>  
    <mapper resource="/Mybatis/config/mapper/UserMapper.xml">  
    </mapper>  
</mappers>
```

5. 创建 POJO 类，用来封装查询结果。

操作数据库的基本方法

2018年6月24日 17:25

1. 查询

```
@Test
public void findUserId() throws IOException {

    // 配置文件
    String path = "SqlMapConfig.xml";
    InputStream config = Resources.getResourceAsStream(path);

    // 创建会话工厂
    SqlSessionFactory factory = new
    SqlSessionFactoryBuilder().build(config);

    // 创建 SqlSession
    SqlSession session = factory.openSession();

    // 操作数据库
    // 第一个参数是映射文件中的 SQL 语句的 id, 格式是 namespace.id
    // 第二个参数就是要填充到 SQL 语句中的参数
    User user = session.selectOne("test.findUserId", 10);
    System.out.println(user);

    // 释放资源
    session.close();
}
```

2. 增加

```
public void insertUser() {
    // 插入 user 对象
    User user = new User();
    user.setUsername("Tony");
    user.setAddress("Shanghai");
    user.setSex("1");
    user.setBirthday(new Date());

    // 插入记录
    session.insert("test.insertUser", user);

    // 提交事务
    session.commit();
}
```

相应的配置:

```
<!-- 参数是 POJO 对象的属性, MyBatis 通过 ONGL 获取对象的属性值 -->
<insert id="insertUser" parameterType="mybatis.po.User">
```

```

INSERT INTO
user(id,username,birthday,sex,address)
VALUES(#{id},#{username},#{birthday},#{sex},#{address})
</insert>

```

如果想在插入新记录之后返回主键，可以如下配置：

(1) 自增型主键

```

<!-- 参数是 POJO 对象的属性, MyBatis 通过 ONGL 获取对象的属性值 -->
<insert id="insertUser" parameterType="mybatis.po.User">
    <!-- 将新增记录的主键返回, 保存到 user 对象中 -->
    <!-- keyProperty 指定了将主键保存到对象的哪个属性中, order 指定了下面这
    条 SQL 语句在 INSERT 语句之前还是之后执行 -->
    <selectKey keyProperty="id" order="AFTER"
    resultType="java.lang.Integer">
        <!-- LAST_INSERT_ID() 只适用于自增型主键 -->
        SELECT LAST_INSERT_ID()
    </selectKey>
    INSERT INTO
    user(id,username,birthday,sex,address)
    VALUES(#{id},#{username},#{birthday},#{sex},#{address})
</insert>

```

(2) 非自增型主键

可以用 UUID 函数生成主键，但需要修改主键字段类型为 VARCHAR(35)。这个函数要在 INSERT 之前调用。

```

<!-- 参数是 POJO 对象的属性, MyBatis 通过 ONGL 获取对象的属性值 -->
<insert id="insertUser" parameterType="mybatis.po.User">
    <!-- 将新增记录的主键返回, 保存到 user 对象中 -->
    <!-- keyProperty 指定了将主键保存到对象的哪个属性中, order 指定了下面这
    条 SQL 语句在 INSERT 语句之前还是之后执行 -->
    <selectKey keyProperty="id" order="BEFORE"
    resultType="java.lang.String">
        <!-- LAST_INSERT_ID() 只适用于自增型主键 -->
        SELECT UUID()
    </selectKey>
    INSERT INTO
    user(id,username,birthday,sex,address)
    VALUES(#{id},#{username},#{birthday},#{sex},#{address})
</insert>

```

3. 删除

```

public void deleteUserById() {
    session.delete("test.deleteUserById", 39);
    session.commit();
}

```

相应的配置：

```

<!-- 删除用户 -->
<delete id="deleteUserById" parameterType="int">
    DELETE FROM user WHERE
    id=#{id}
</delete>

```

4. 更新

```

public void updateUser() {
    User user = new User();
    user.setId(10);
    user.setUsername("张三");
    user.setBirthday(new Date());
    user.setSex("1");
    user.setAddress("北京");
    session.update("test.updateUser", user);
    session.commit();
}

```

相应的配置：

```

<!-- 更新 -->
<!-- 因为要更新的是一个对象，所以可以设置参数类型为 POJO -->
<update id="updateUser" parameterType="mybatis.po.User">
    UPDATE user
    SET
        username=#{username},birthday=#{birthday},sex=#{sex},address=#{
        address}
    WHERE id=#{id}
</update>

```

说明：

1. #{} 和 \${}

(1) #{} 表示一个占位符号，通过#{ } 可以实现 preparedStatement 向占位符中设置值，自动进行 java 类型和 jdbc 类型转换，#{ } 可以有效防止 sql 注入。#{ } 可以接收简单类型值或 pojo 属性值。如果 parameterType 传输单个简单类型值，#{ } 括号中可以是 value 或其它名称。

```
concat(' ',#{value},' ')
```

(2) \${ } 表示拼接 sql 串，通过\${ } 可以将 parameterType 传入的内容拼接在 sql 中且不进行 jdbc 类型转换，\${ } 可以接收简单类型值或 pojo 属性值，如果 parameterType 传输单个简单类型值，\${ } 括号中只能是 value。

2.selectOne 和 selectList

selectOne 查询一条记录，如果使用 selectOne 查询多条记录则抛出异常。selectList 可以查询一条或多条记录。

详解 SqlSession

2018年6月24日 21:06

SqlSession 中封装了对数据库的操作，如：查询、插入、更新、删除等。通过 SqlSessionFactory 创建 SqlSession，而 SqlSessionFactory 是通过 SqlSessionFactoryBuilder 创建的。

1. SqlSessionFactoryBuilder

SqlSessionFactoryBuilder 用于创建 SqlSessionFactory，SqlSessionFactory 一旦创建完成就不需要 SqlSessionFactoryBuilder 了，因为 SqlSession 是通过 SqlSessionFactory 生产，所以可以将 SqlSessionFactoryBuilder 当成一个工具类使用，最佳使用范围是方法范围即方法体内局部变量。

2. SqlSessionFactory

SqlSessionFactory 是一个接口，接口中定义了 openSession 的不同重载方法，SqlSessionFactory 的最佳使用范围是**整个应用运行期间**，一旦创建后可以重复使用，通常以**单例模式**管理 SqlSessionFactory。

3. SqlSession

SqlSession 是一个面向用户的接口，SqlSession 中定义了数据库操作，默认使用 DefaultSqlSession 实现类。

它是**线程不安全的**，因为在它的实现类中还有数据域属性，所以最佳应用范围是方法体内**局部变量**。

注意，要**确保 SqlSession 被关闭**。保证做到这点的最佳方式是下面的工作模式：

```
SqlSession session = sqlSessionFactory.openSession();
try {
    // following 3 lines pseudocod for "doing some work"
    session.insert(...);
    session.update(...);
    session.delete(...);
    session.commit();
} finally {
    session.close();
}
```

还有，如果你正在使用jdk 1.7以上的版本还有 MyBatis 3.2 以上的版本，你可以使用 try-with-resources 语句：

```
try (SqlSession session = sqlSessionFactory.openSession()) {
    // following 3 lines pseudocode for "doing some work"
    session.insert(...);
    session.update(...);
    session.delete(...);
    session.commit();
}
```

编写 DAO

2018年6月24日 21:00

一、传统方法

编写 DAO 接口和实现类，在实现类中注入 SqlSessionFactory，用来创建 SqlSession。

1. 编写接口

```
// Dao 接口
public interface UserDao {
    public User findUserById(int id) throws Exception;
    public void insertUser(User user) throws Exception;
    public void deleteUser(int id) throws Exception;
}
```

2. 编写实现类

```
// 接口实现类
public class UserDaoImpl implements UserDao{

    private SqlSessionFactory factory;

    // 先暂时通过构造器注入
    public UserDaoImpl(SqlSessionFactory factory) {
        this.factory = factory;
    }

    @Override
    public User findUserById(int id) throws Exception {
        SqlSession session = factory.openSession();
        User user = session.selectOne("test.findUserById", id);
        session.close();
        return user;
    }

    @Override
    public void insertUser(User user) throws Exception {
        SqlSession session = factory.openSession();
        session.insert("test.insertUser", user);
        session.commit();
        session.close();
    }

    @Override
    public void deleteUser(int id) throws Exception {
        SqlSession session = factory.openSession();
        session.delete("test.deleteUserById", id);
        session.commit();
        session.close();
    }
}
```

传统方法存在的问题：接口实现类中存在大量的样板代码；操作数据库时 statementId 硬编码，而且这些方法使用泛型，传入的参数即使类型错误在编译阶段也不报错。

二、mapper 代理

首先配置映射文件；然后编写接口，这种接口一般叫 mapper 但实际上就相当于 Dao，这种接口的编写需要遵循一些规范，这样 MyBatis 就可以自动生成接口实现类代理对象。

1. 规范

- (1) 映射文件中，<mapper> 的 namespace 是接口的全名：包名.接口名
- (2) 接口的方法名和 statementId 一致
- (3) 接口方法的形参类型和 parameterType 指定的类型一致
- (4) 接口方法的返回值类型和 resultType 指定的类型一致

这样一来实际上就是自动生成了传统方法编写的实现类中对数据库操作的代码。

2. 写完配置文件和接口之后不要忘了在 SqlMapConfig.xml 中进行配置。<mappers> 下可以有多个 <mapper>。

完成以上工作之后，使用时只需通过 SqlSession 对象获取代理对象，从而操作数据库即可。

```
UserMapper mapper = session.getMapper(UserMapper.class); // 获得 mapper  
代理对象  
mapper.findUserById(10);
```

3. 代理对象细节

如果代理对象的调用方法返回单个 POJO 对象，那么实际上对象会调用 selectOne 方法；如果返回的是集合，那么调用的是 selectList 方法。

4. 接口方法的参数

mapper 接口方法的参数一般都只有一个，为了满足不同的业务需求，可以使用经过包装的 POJO 作为参数。

详解 SqlMapConfig.xml

2018年6月25日 14:34

一、避免数据库连接参数的硬编码

1. 将数据库连接参数单独保存在一个 properties 文件中，然后在 SqlMapConfig.xml 中加载这些值。

好处：

- 避免了 SqlMapConfig.xml 中硬编码参数
- 对这些参数进行统一管理，以便以后其他配置文件读取

将以下属性保存在一个单独的 .properties 文件中：

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8
jdbc.username=root
jdbc.password=loveisKING
```

然后在 SqlMapConfig.xml 中加载并读取：

```
<?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>
    <!-- 加载属性文件 -->
    <properties resource="db.properties">
    </properties>
    <!-- 和spring整合后 environments配置将废除 -->
    <environments default="development">
        <environment id="development">
            <!-- 使用jdbc事务管理 -->
            <transactionManager type="JDBC" />
            <!-- 数据库连接池 -->
            <dataSource type="POOLED">
                <!-- 这里的属性值变成了 ${} 占位符 -->
                <property name="driver" value="${jdbc.driver}" />
                <property name="url" value="${jdbc.url}" />
                <property name="username" value="${jdbc.username}" />
                <property name="password" value="${jdbc.password}" />
            </dataSource>
        </environment>
    </environments>
    <!-- 加载映射文件 -->
    <mapppers>
        <mapper resource="mapper/UserMapper.xml"></mapper>
        <mapper resource="mapper/userMap.xml"></mapper>
    </mapppers>
</configuration>
```

2. <properties> 标签的特性

(1) 这个标签可以有子标签 <property>，也就是说可以把属性键值对直接放在<properties>标签中

(2) MyBatis 读取属性的顺序：

1. <property> 中定义的属性

2. <properties> 的属性 resource 或 url 指向的属性文件
resource 和 url 的区别:

- resource 指向 CLASSPATH 中的文件
- url 指定了文件的绝对路径

3. 映射文件中 SQL 语句的参数值也会被作为的属性

后一步读取的属性将会覆盖前一步读取的同名属性。这就可能会引发一些问题，比如 .properties 文件中有一个属性名为 name，映射文件中有一条 SQL 语句 SELECT * FROM user WHERE username=#{name}，那么此时传递给 SQL 语句的参数值就会覆盖属性 name 的属性值。

建议：不要用 <property> 定义属性，把属性全部放在 .properties 文件中，而且属性名要有一定的特殊性比如加上前缀/后缀等等。

二、全局参数 <settings>

可以用这个标签来设置一些运行参数，比如开启二级缓存、延迟加载等。这个标签定义的参数是全局性的，设置不当的话会影响 MyBatis 的行为。参考官方文档：

<http://www.mybatis.org/mybatis-3/zh/configuration.html#settings>

三、类型别名 <typeAliases>

当需要在 parameterType 中指定参数类型时，原先给出的类的全名比如 java.lang.String，这就很不方便。MyBatis 允许使用别名来简化这个工作，比如 java.lang.String 的别名是 string。MyBatis 默认提供的别名有：

<http://www.mybatis.org/mybatis-3/zh/configuration.html#typeAliases>。还可以自定义别名：

```
<typeAliases>
  <typeAlias alias="Author" type="domain.blog.Author"/>
</typeAliases>
```

更为常见的用法是指定一个包名，MyBatis 会在包名下面搜索需要的 Java Bean：

```
<typeAliases>
  <package name="domain.blog"/>
</typeAliases>
```

每一个在包 domain.blog 中的 Java Bean，在没有注解的情况下，会使用 Bean 的首字母小写的非限定类名来作为它的别名。比如 domain.blog.Author 的别名为 author；若有注解，则别名为其注解值。看下面的例子：

```
@Alias("author")
public class Author {
    //...
}
```

四、类型处理器 <typeHandlers>

类型处理器就是将 Java 类型映射为数据库的数据类型。MyBatis 自带的处理器有很多，通常无需自定义。 <http://www.mybatis.org/mybatis-3/zh/configuration.html#typeHandlers>

五、映射器 <mappers>

这个标签用来告诉 MyBatis 映射文件在哪里。

- (1) 通过 resource 属性加载单个映射文件
- (2) 通过 url 属性加载单个映射文件
- (3) 通过 class 属性加载映射文件

在使用 mapper 代理开发接口时，可以用这个 class 属性来指定接口，然后 MyBatis 就会加载相应的映射文件。条件：映射文件的文件名和接口名一致，而且映射文件和接口的 .java 文件要在一个目录下。

```
<mappers>
  <mapper class="org.mybatis.builder.AuthorMapper"/>
  <mapper class="org.mybatis.builder.BlogMapper"/>
  <mapper class="org.mybatis.builder.PostMapper"/>
</mappers>
```

（4）批量加载指定包内的接口（推荐使用）
使用这种方法时也要遵循上面提到的规范。

```
<!-- 将包内的映射器接口全部注册为映射器 -->
<mappers>
  <package name="org.mybatis.builder"/>
</mappers>
```

输入输出映射

2018年6月25日 16:52

一、输入映射 parameterType

1. 传递简单类型
2. 传递 POJO
3. 传递 hashMap

```
<select id="findUserByMap" parameterType="hashMap"
      resultType="user">
    SELECT id,username,birthday,sex,address FROM user WHERE
    id=#{id} AND username LIKE concat('%',{username},'%')
</select>
```

参数是 hashMap 的 key。调用方法时：

```
Map<String,Object> map = new HashMap<>();
map.put("id", 10);
map.put("username", "张三");
System.out.println(mapper.findUserByMap(map));
```

4. 传递 POJO 包装对象

对于一些复杂的查询条件，可以使用自定义的 POJO，把需要的查询条件传递进去。

```
// 这个类用来扩展原始用户的信息，比如由于业务发展要新增是否为会员这项信息
public class UserCustom extends User {

}

// 这个对象一般都是视图层传递过来的
// 这个对象不仅可以封装用户的个人信息，还可以一并封装其他的相关信息，比如订单记录等等
public class UserQueryVo {

    private UserCustom user; // 用户的信息

    // 可以有其他的查询条件

    public UserCustom getUser() {
        return user;
    }

    public void setUser(UserCustom user) {
        this.user = user;
    }

}
```

配置映射文件：

```
<!-- 综合查询 -->
<select id="findUserList" parameterType="mybatis.po.UserQueryVo"
      resultType="mybatis.po.UserCustom">
    SELECT id,username,birthday,sex,address FROM user WHERE
```

```

        user.sex = #{user.sex} AND user.username LIKE
        concat('%',#{user.username},'%')
    </select>

```

传递的参数类型是 UserQueryVo，而这个类有一个属性 user，这个 user 有属性 id 和 username，所以在 SQL 语句中可以写成 #{user.sex} 和 #{user.username}。注意属性名不能写错。

二、输出映射 resultType 和 resultMap

1. resultType

(1) 输出 POJO

只有查询的列名和 POJO 中的属性名完全一致时，列值才能被映射到 POJO 对象中。

(2) 输出 POJO 列表

这种情况下 resultType 的值和输出 POJO 时的一样，但是接口方法的返回值就是 List。

(3) 输出简单类型

只有当查询结果中只有一行一列，才能输出简单类型。

```

<!-- 使用聚合函数 -->
<select id="findUserCount"
    parameterType="mybatis.po.UserQueryVo" resultType="int">
    SELECT count(*) FROM
    user WHERE
    user.sex = #{user.sex}
    AND user.username LIKE
    concat('%',#{user.username},'%')
</select>

```

2. resultMap

(1) 如果查询的列名和 POJO 的属性名不一致，那么可以通过 resultMap 完成映射。首先要配置一个 resultMap：

```

<!-- 配置 resultMap -->
<resultMap type="user" id="result2user">
    <!-- id 指的是查询结果集的唯一标识，column 就是查询的列名（可能是列的别名），property 就是对应的 POJO 属性 -->
    <id column="id_" property="id"></id>
    <!-- result 用来定义其他列和对应属性的映射 -->
    <result column="username_" property="username"></result>
</resultMap>
<!-- 如果 resultMap 被定义在其他映射文件中，那么就要加上 namespace 作为限定 -->
<select id="findUserById_" parameterType="int"
    resultMap="result2user">
    SELECT id AS id_,username AS username_ FROM user WHERE id=#{value}
</select>

```

然后就可以使用 findUserById_ 进行查询。

动态 SQL

2018年6月25日 19:16

MyBatis 可以通过表达式进行判断，从而灵活地拼接 SQL 语句。

一、基本用法

```
<select id="findUserList" parameterType="mybatis.po.UserQueryVo"
      resultType="mybatis.po.UserCustom">
    SELECT id,username,birthday,sex,address FROM user
    <where>
        <!-- 当两个条件都可用时，where 标签会自动去掉第一个 AND -->
        <!-- 先检验参数值的合法性，然后再决定是否使用这个参数作为条件 -->
        <!-- if 标签用于定义判断条件 -->
        <if test="user != null">
            <if test="user.sex != null and user.sex != ''">
                AND user.sex = #{user.sex}
            </if>
            <if test="user.username != null and user.username != ''">
                AND user.username LIKE
                concat('%',#{user.username},'%')
            </if>
        </if>
    </where>
</select>
```

这里的配置意味着，如果 user.sex 的值不合法，那么生成的 SQL 语句是 **SELECT id,username,birthday,sex,address FROM user WHERE user.username LIKE concat('%',?, '%')**；如果 user.username 不合法，那么生成的 SQL 语句是 **SELECT id,username,birthday,sex,address FROM user WHERE user.sex = ?**；如果两个参数值都合法，那么生成的 SQL 语句就是 **SELECT id,username,birthday,sex,address FROM user WHERE user.sex = ? AND user.username LIKE concat('%',?, '%')**。

二、定义 SQL 片段

多个 SQL 语句中可能有相同的条件，那么可以把这些条件抽取出来作为 SQL 片段供多条语句使用。

```
<!-- 定义 SQL 片段，使更多的 SQL 语句可以实现灵活拼接 -->
<!-- 一般而言，基于单表定义 SQL 片段可以提高可重用性；在 SQL 片段中也不要包含
where 标签，因为 SQL 语句可能引用多个片段，
    一个 where 标签就会生成一个 WHERE 关键字，多个 where 标签就会引起冲突 -->
<sql id="query_user_where">
    <if test="user != null">
        <if test="user.sex != null and user.sex != ''">
            AND user.sex = #{user.sex}
        </if>
        <if test="user.username != null and user.username != ''">
            AND user.username LIKE
            concat('%',#{user.username},'%')
        </if>
    </if>
</sql>

<select id="findUserList" parameterType="mybatis.po.UserQueryVo"
      resultType="mybatis.po.UserCustom">
    SELECT id,username,birthday,sex,address FROM user
```

```

<where>
    <!-- 引用 SQL 片段, 如果片段不在这个映射文件中, 那么就要在 id 前面加上
    namespace -->
    <include refid="query_user_where"></include>
</where>
</select>

```

三、<foreach> 标签

如果向 SQL 中传递数组或者 List（任何可迭代的对象都能传入，但是数组和 List 比较常用），那么就用这个标签进行解析。

```

<sql id="query_user_where">
    <if test="ids != null">
        <!-- 拼接生成 AND ( id = 1 OR id = 2 OR id = 3) -->
        <foreach collection="ids" item="id_" open="AND (" close=")"
            separator="OR">
            <!-- 每次遍历时生成的内容 -->
            id = #{id_}
        </foreach>
    </if>
</sql>

```

现在对象有一个属性 ids，这是一个 List，里面保存着多个需要查询的 id。foreach 标签中各个属性的含义：

- (1) collection: 传入的数组/List 的名字
- (2) item: 这个标识用来指代每次遍历从数组/List 取出的元素
- (3) open: 需要拼接的 SQL 语句中的开始部分
- (4) close: 需要拼接的 SQL 语句中的开始部分
- (5) separator: 每次遍历都会生成整个 SQL 语句的一小部分，这些部分之间用 (6)

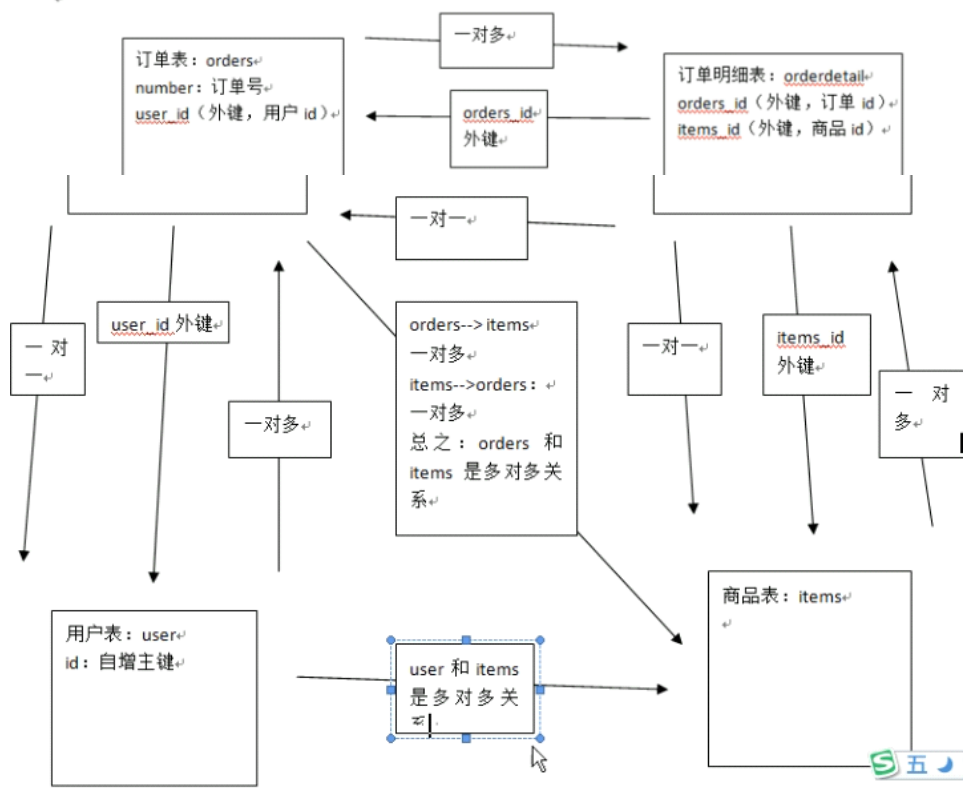
separator 指定的内容进行拼接

其实上面的 SQL 语句应该用 IN 关键字来编写。

高级映射

2018年6月25日 21:11

数据模型:



一、一对一查询

查询用户的订单信息。

1. 创建 POJO

如果最初设计的 POJO 不能满足查询要求, 那么就继承在查询结果中占有较多字段的 POJO, 设计一个扩展的 POJO。

2. 使用 resultMap

POJO :

```
package mybatis.po;

// 扩展的 Orders 类
public class OrdersCustom extends Orders {

    private String username;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    @Override
    public String toString() {
        return "order_id:" + getId() + ",username:" + username;
    }
}
```

```
}
```

映射文件:

```
<!-- 查询订单关联的用户 -->
<select id="queryOrdersCustomUser"
        resultType="mybatis.po.OrdersCustom">
    SELECT
        orders.id,orders.user_id,orders.number,orders.create_time,user.usrname
    FROM orders,user
    WHERE orders.user_id = user.id;
</select>
```

3. 使用 resultMap
POJO:

```
public class Orders {
    private Integer id;
    private Integer user_id;
    private String number;
    private Date create_time;
    private String note;
    private User user;

    //...
}
```

映射文件:

```
<!-- 把查询结果映射到原始的 Orders 中 -->
<resultMap type="mybatis.po.Orders" id="OrdersUser">
    <id column="id" property="id"></id>
    <result column="user_id" property="user_id"></result>
    <result column="number" property="number"></result>
    <result column="create_time" property="create_time"></result>
    <result column="user_id" property="user_id"></result>
    <!-- 用于映射多表查询中的关联信息 -->
    <!-- Orders 中有一个 User 类型的属性，以下配置就是把查询结果映射到的
    User 的属性中 -->
    <association property="user" javaType="mybatis.po.User">
        <id column="user_id" property="id"></id>
        <result column="username" property="username"></result>
    </association>
</resultMap>

<select id="queryOrdersUser" resultMap="OrdersUser">
    SELECT
        orders.id,orders.user_id,orders.number,orders.create_time,user.username
    FROM orders,user
    WHERE orders.user_id = user.id;
</select>
```

4. resultType 和 resultMap 的比较

(1) 使用 resultType 比较简单，如果 POJO 中没有和查询结果的列相对应的属性，就增加。一般而言，没有特殊的查询要求时使用 resultType。

(2) 如果有特殊的查询要求，比如多表查询时将关联信息映射到另一个 POJO 中，就用 resultMap。

(3) resultMap 可以实现延迟加载，但是 resultTyp 不行。

二、一对多查询

查询订单关联的用户信息和订单明细。

映射文件：

```
<resultMap type="mybatis.po.Orders" id="OrdersUserDetails"
    extends="OrdersUser">
    <!-- 继承了上面已经配置过的 resultMap, 所以就不用重复配置订单和用户的映射 -->
    <!-- 映射订单明细到 details 属性中 -->
    <!-- 一个订单有多条明细, details 是一个List, 所以用 collection 进行映射, 就是把查询结果映射到集合中 -->
    <!-- 这里的 property 指的是 List 元素的属性 -->
    <collection property="details"
        ofType="mybatis.po.OrderDetail">
        <id column="orderdetail_id" property="id"></id>
        <result column="item_id" property="item_id"></result>
        <result column="item_num" property="item_num"></result>
    </collection>
</resultMap>

<select id="queryOrdersUserDetails"
    resultMap="OrdersUserDetails">
    SELECT
    orders.id,
    orders.user_id,
    orders.number,
    orders.create_time,
    user.username,
    orderdetail.id AS orderdetail_id,
    orderdetail.item_id,
    orderdetail.item_num
    FROM orders,user,orderdetail
    WHERE orders.user_id = user.id
    AND orders.id = orderdetail.order_id
</select>
```

三、多对多查询

查询用户及用户购买的商品信息。

扩展 User :

```
public class User {

    private int id;

    private String username; // 用户姓名

    private String sex; // 性别

    private Date birthday; // 生日

    private String address; // 地址

    private List<Orders> orders;

    //...
}
```

扩展 OrderDetail:

```

public class OrderDetail {

    private Integer id;

    private Integer order_id;

    private Integer item_id;

    private Integer item_num;

    private Items item;

    //...
}

```

映射文件:

```

<resultMap type="user" id="UserItems">
    <!-- 用户 -->
    <id column="id" property="id"></id>
    <result column="username" property="username"></result>
    <!-- 订单 -->
    <collection property="orders" ofType="mybatis.po.Orders">
        <id column="order_id" property="id"></id>
        <result column="number" property="number"></result>
        <result column="create_time" property="create_time"></result>
    <!-- 订单明细 -->
    <collection property="details"
        ofType="mybatis.po.OrderDetail">
        <id column="orderdetail_id" property="id"></id>
        <result column="item_id" property="item_id"></result>
        <result column="item_num" property="item_num"></result>
        <!-- 这条明细对应的商品 -->
        <association property="item" javaType="mybatis.po.Items">
            <id column="item_id" property="id"></id>
            <result column="item_name" property="name"></result>
            <result column="price" property="price"></result>
        </association>
    </collection>
    </collection>
</resultMap>

<select id="queryUserItems" resultMap="UserItems">
    SELECT
    user.id,
    user.username,
    orders.id AS order_id,
    orders.number,
    orders.create_time,
    orderdetail.id AS orderdetail_id,
    orderdetail.item_id,
    items.name AS
    item_name,
    items.price,
    orderdetail.item_num
    FROM
    user,
    orders,
    orderdetail,
    items
    WHERE
    orders.user_id = user.id

```

```
        AND orders.id =  
        orderdetail.order_id  
        AND orderdetail.item_id = items.id;  
</select>
```

总之，有特殊的映射要求就用 resultMap。比如查询用户权限范围内的一级模块和二级模块，可以先将一级模块映射到 list 中，再把二级模块映射到一级模块的 list 中。这种情况下使用 resultMap 就比较方便，如果用 resultType 就必须自己写双重循环。
注意 collection 和 association 的用法和区别。

延迟加载

2018年6月26日 11:04

一、概念

先进行单表查询，如果有需要就再进行多表查询，这样可以提高性能。

比如说，先查询订单的信息，然后等想进一步查看该订单的用户时，就再查询该用户的信息。

二、使用 <association>

先配置查询订单的映射文件：

```
<select id="qrueyOrderThenUser" resultMap="Order2User">
    SELECT
        id,number,create_time,user_id FROM orders
</select>
```

然后再配置查询订单对应的用户的映射：

```
<resultMap id="Order2User" type="mybatis.po.Orders">
    <!-- 订单 -->
    <id column="id" property="id"></id>
    <result column="user_id" property="user_id"></result>
    <result column="number" property="number"></result>
    <result column="create_time" property="create_time"></result>
    <!-- 用户 -->
    <!-- 延迟加载查询用户信息的 SQL 语句, select 就是查询用户的 SQL, column
    是指当前查询结果中的哪一列可以和 user
    表关联起来 -->
    <association property="user" javaType="user"
        select="mybatis.mapper.UserMapper.findUserById"
        column="user_id">
    </association>
</resultMap>
```

这样配置的话，其实完整的 SQL 语句就相当于：

```
SELECT
    id,
    number,
    create_time,
    user_id,
    (SELECT
        username
    FROM
        user
    WHERE
        orders.user_id = user.id) AS username
FROM
    orders;
```

最后，还要在 SqlMapConfig.xml 中开启延迟加载：

```
<!-- 全局参数 -->
<settings>
    <!-- 延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。 -->
    <setting name="lazyLoadingEnabled" value="true" />
    <!-- 每个属性会按需加载 -->
```

```
<setting name="aggressiveLazyLoading" value="false"></setting>
</settings>
```

调用方法时：

```
public void test5() { // 先查询订单，延迟查询用户
    SqlSession session = factory.openSession();
    OrdersCustomMapper mapper =
        session.getMapper(OrdersCustomMapper.class);
    // 查询订单
    List<Orders> list = mapper.qrueyOrderThenUser();
    // 遍历订单列表
    for(Orders o : list) {
        User user = o.getUser(); // 延迟加载 user 属性
        System.out.println(user.getUsername());
    }
}
```

三、使用 <collection>
方法一样

四、自己实现延迟加载
首次查询和延迟加载的两个方法分离，按需调用延迟加载方法。

五、总结
先进行简单的查询（最典型的例子就是单表查询），然后按需进行下一步查询。

缓存（以查询为例）

2018年6月26日 12:41

MyBatis 提供了查询缓存，减轻数据库压力，提高性能。

一、概述

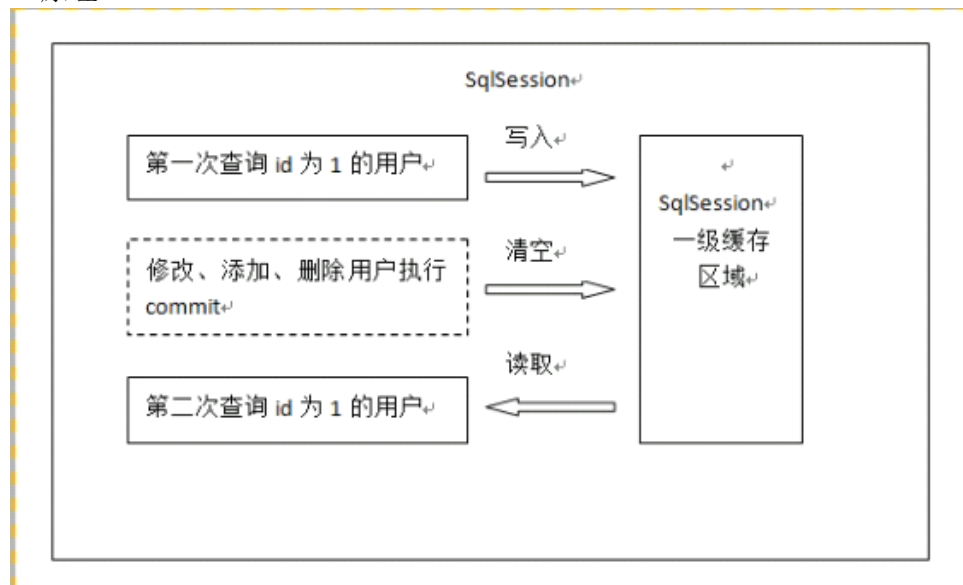
一级缓存是 SqlSession 级别的缓存，在操作数据库时需要构造 SqlSession 对象，在这个对象中有一个 hashMap，用来存储缓存。

二级缓存是 mapper 级别的。多个 SqlSession 使用同一个 mapper 的 SQL 语句，这时就是用到保存在 mapper 中的缓存。



二、一级缓存

1. 原理



图中的清空的目的是避免脏读。

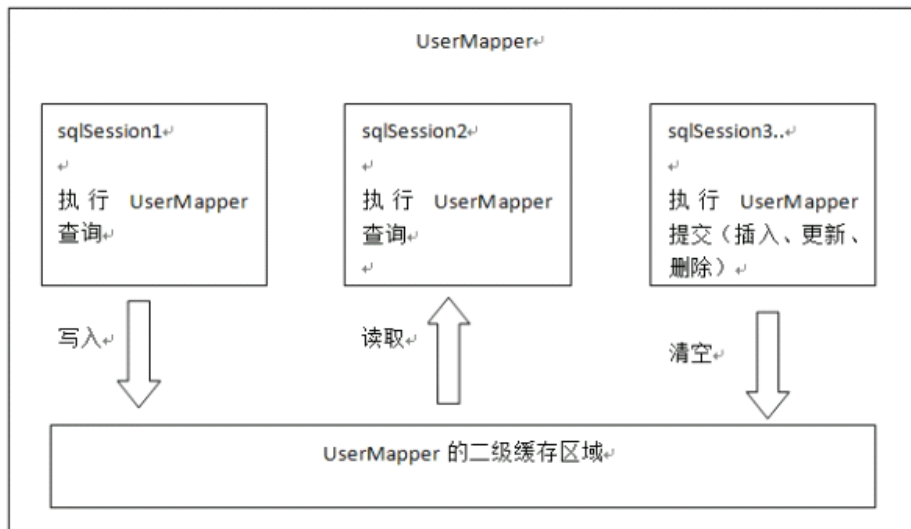
2. 实现

MyBatis 默认开启一级缓存，无需手动配置。

如果重复调用多次相同的 mapper 方法而且参数值一样，那么除了第一次调用时会生成 SQL 语句查询数据库以外，其他调用都不会再次操作数据库，除非在这期间提交了事务。

三、二级缓存

1. 原理



2. 与一级缓存的区别

(1) 二级缓存区域是按照映射文件中 mapper 的 namespace 来划分的，如果多个 mapper 有相同的 namespace，那么这些 mapper 就共享同一个二级缓存区域。

(2) 多个 SqlSession 共享同一个二级缓存区域。

(3) 与一级缓存相同的是把缓存数据保存在 hashMap 中。

3. 实现

(1) 开启二级缓存

首先在 SqlMapConfig.xml 中进行配置：

```
<setting name="cacheEnabled" value="true"></setting>
```

配置需要二级缓存的 mapper 的映射文件：

```
<cache></cache>
```

(2) 让 POJO 实现序列化

二级缓存数据的存储介质不一定在内存中，为了将数据取出就需要进行反序列化。

(3) 确保关闭 SqlSession，否则缓存数据不会被写入缓存区域。

4. 相关配置

(1) 禁用缓存

在 statement 中设置 useCache=false 可以禁用当前 select 语句的二级缓存. 默认情况是 true，即使用二级缓存。

```
<select id="findOrderListResultMap"
resultMap="ordersUserMap" useCache="false">
```

(2) 刷新缓存

在 mapper 的同一个 namespace 中，如果有其它 insert、update、delete 操作，那么执行完操作之后需要刷新缓存，如果不执行刷新缓存会出现脏读。

设置 statement 配置中的 flushCache="true" 属性，默认情况下为 true 即刷新缓存，如果改成 false 则不会刷新。使用缓存时如果手动修改数据库表中的查询数据会出现脏读。

```
<insert id="insertUser"
parameterType="cn.itcast.mybatis.po.User" flushCache="true">
```

(3) 其他参数

```
<cache
eviction="FIFO"
flushInterval="60000"
```

```
size="512"
readOnly="true"/>
```

- ①flushInterval（刷新间隔）可以被设置为任意的正整数，而且它们代表一个合理的毫秒形式的时间段。默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新。
- ②size（引用数目）可以被设置为任意正整数，要记住你缓存的对象数目和你运行环境的可用内存资源数目。默认值是1024。
- ③readOnly（只读）属性可以被设置为 true 或 false。只读的缓存会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。可读写的缓存会返回缓存对象的拷贝（通过序列化）。这会慢一些，但是安全，因此默认是 false。
- ④eviction 指定了回收策略，可用的回收策略有：
 - LRU - 最近最少使用的:移除最长时间不被使用的对象。默认
 - FIFO - 先进先出:按对象进入缓存的顺序来移除它们。
 - SOFT - 软引用:移除基于垃圾回收器状态和软引用规则的对象。
 - WEAK - 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对象。

5. 应用场景

对于访问多的查询请求且用户对查询结果实时性要求不高，此时可采用 mybatis 二级缓存技术降低数据库访问量，提高访问速度，业务场景比如：耗时较高的统计分析 sql、电话账单查询 sql 等。

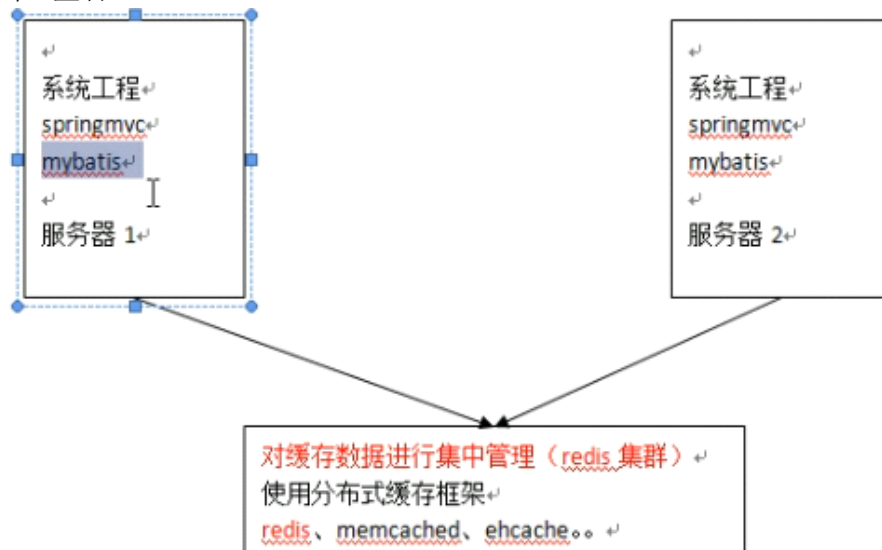
实现方法如下：通过设置刷新间隔时间，由 mybatis 每隔一段时间自动清空缓存，根据数据变化频率设置缓存刷新间隔 flushInterval，比如设置为30分钟、60分钟、24小时等，根据需求而定。

6. 局限性

mybatis 二级缓存对细粒度的数据级别的缓存实现不好。

比如如下需求：对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次都能查询最新的商品信息，此时如果使用 mybatis 的二级缓存就无法实现当一个商品变化时只刷新该商品的缓存信息而不刷新其它商品的信息，因为mybaitis 的二级缓存区域以 mapper 为单位划分，当一个商品信息变化会将所有商品信息的缓存数据全部清空。解决此类问题需要在业务层根据需求对数据有针对性缓存。

四、整合 ehcache



缓存的数据在各个服务器中单独存储，不便于管理，所以要使用分布式缓存。MyBatis 无法实现分布式缓存，所以需要和其他的分布式缓存框架进行整合。

1. 整合原理

Mybatis 提供 Cache 接口，要实现自己的缓存方式，就实现这个接口。（MyBatis 提供了这个接口的一个默认实现类 PerpetualCache。）

```

public interface Cache {
    String getId();
    int getSize();
    void putObject(Object key, Object value);
    Object getObject(Object key);
    boolean hasKey(Object key);
    Object removeObject(Object key);
    void clear();
}

```

实现了接口之后，还要在映射文件中进行配置：

```

<cache type="com.domain.something.MyCustomCache"/>

```

2. 整合方法

配置映射文件：

```

<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>

```

引入 ehcache 的配置文件：

```

<ehcache:config
  xmlns:ehcache="http://www.ehcache.org/v3"
  xmlns:jcache="http://www.ehcache.org/v3/jsr107">

  <!--
    OPTIONAL
    services to be managed and lifecycled by the CacheManager
  -->
  <ehcache:service>
    <!--
      One element in another namespace, using our JSR-107 extension as
      an example here
    -->
    <jcache:defaults>
      <jcache:cache name="invoices" template="myDefaultTemplate"/>
    </jcache:defaults>
  </ehcache:service>

  <!--
    OPTIONAL
    A <cache> element defines a cache, identified by the mandatory
    'alias' attribute, to be managed by the CacheManager
  -->
  <ehcache:cache alias="productCache">

    <!--
      OPTIONAL, defaults to java.lang.Object
      The FQCN of the type of keys K we'll use with the Cache<K, V>
    -->
    <ehcache:key-type
      copier="org.ehcache.impl.copy.SerializingCopier">
      java.lang.Long</ehcache:key-type>

    <!--
      OPTIONAL, defaults to java.lang.Object
      The FQCN of the type of values V we'll use with the Cache<K,
      V>
    -->
    <ehcache:value-type
      copier="org.ehcache.impl.copy.SerializingCopier">
      com.pany.domain.Product</ehcache:value-type>

    <!--

```

```

    OPTIONAL, defaults to no expiry
    Entries to the Cache can be made to expire after a given time
-->
<ehcache:expiry>
  <!--
    time to idle, the maximum time for an entry to remain
untouched
    Entries to the Cache can be made to expire after a given
time
    other options are:
      * <ttl>, time to live;
      * <class>, for a custom Expiry implementation; or
      * <none>, for no expiry
  -->
  <ehcache:tti unit="minutes">2</ehcache:tti>
</ehcache:expiry>

  <!--
    OPTIONAL, defaults to no advice
    An eviction advisor, which lets you control what entries
should only get evicted as last resort
    FQCN of a org.ehcache.config.EvictionAdvisor implementation
  -->
  <ehcache:eviction-advisor>
com.pany.ehcache.MyEvictionAdvisor</ehcache:eviction-advisor>

  <!--
    OPTIONAL,
    Let's you configure your cache as a "cache-through",
    i.e. a Cache that uses a CacheLoaderWriter to load on misses,
and write on mutative operations.
  -->
  <ehcache:loader-writer>
    <!--
      The FQCN implementing
org.ehcache.spi.loaderwriter.CacheLoaderWriter
    -->
    <ehcache:class>
com.pany.ehcache.integration.ProductCacheLoaderWriter</ehcache:class>
    <!-- Any further elements in another namespace -->
  </ehcache:loader-writer>

  <!--
    The maximal number of entries to be held in the Cache, prior
to eviction starting
  -->
  <ehcache:heap unit="entries">200</ehcache:heap>

  <!--
    OPTIONAL
    Any further elements in another namespace
  -->
</ehcache:cache>

  <!--
    OPTIONAL
    A <cache-template> defines a named template that can be used be
<cache> definitions in this same file
    They have all the same property as the <cache> elements above
  -->
  <ehcache:cache-template name="myDefaultTemplate">
    <ehcache:expiry>
      <ehcache:none/>
    </ehcache:expiry>
  </ehcache:cache-template>
  <!--
    OPTIONAL

```

```

        Any further elements in another namespace
    -->
</ehcache:cache-template>

<!--
    A <cache> that uses the template above by referencing the cache-
    template's name in the uses-template attribute:
    -->
<ehcache:cache alias="customerCache" uses-
template="myDefaultTemplate">
    <!--
        Adds the key and value type configuration
    -->
    <ehcache:key-type>java.lang.Long</ehcache:key-type>
    <ehcache:value-type>com.pany.domain.Customer</ehcache:value-type>

    <!--
        Overwrites the capacity limit set by the template to a new value
    -->
    <ehcache:heap unit="entries">200</ehcache:heap>
</ehcache:cache>

</ehcache:config>

```

根据需要修改参数:

```

<mapper namespace="org.acme.FooMapper">
    <cache type="org.mybatis.caches.ehcache.EhcacheCache"/>
    <property name="timeToIdleSeconds" value="3600"/><!--1 hour-->
    <property name="timeToLiveSeconds" value="3600"/><!--1 hour-->
    <property name="maxEntriesLocalHeap" value="1000"/>
    <property name="maxEntriesLocalDisk" value="10000000"/>
    <property name="memoryStoreEvictionPolicy" value="LRU"/>
</cache>
</mapper>

```

与 Spring 整合

2018年6月26日 15:04

一、思路

通过 Spring 管理 SqlSessionFactory、mapper 接口，用 SqlSessionFactory 创建 SqlSession。

二、搭建环境

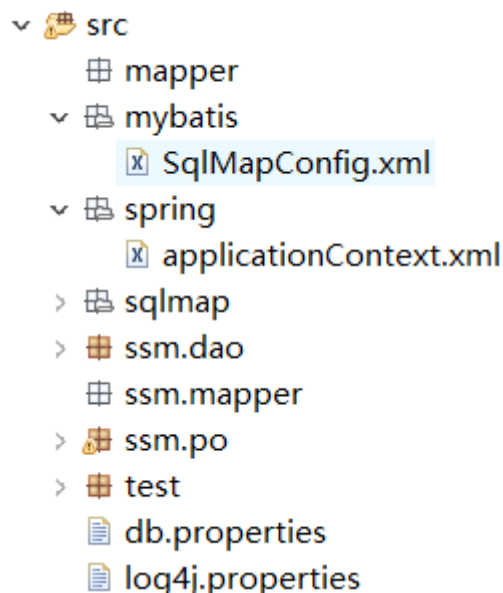
1. jar 包

- Spring
- MyBatis
- 整合包: mybatis-spring

2. 配置文件

包括 MyBatis 和 Spring 的配置文件，分别放在不同的包下。

完成后整个工程的结构为：



说明：sqlmap 下放的是用传统 Dao 方法编写的配置文件，mapper 下则是 mapper 代理方法的配置文件。

三、管理 SqlSessionFactory

在 Spring 的配置文件中：

```
<?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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-
                           context.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd">
```

```
<!-- 数据库的配置文件 -->
```

```

<context:property-placeholder
    location="classpath:db.properties" />
<!-- 配置数据源 -->
<bean id="dbcp" class="org.apache.commons.dbcp2.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="maxTotal" value="10" />
    <property name="maxIdle" value="5" />
</bean>

<!-- SqlSessionFactory 对象 -->
<bean id="factory"
    class="org.mybatis.spring.SqlSessionFactoryBean">
    <!-- 属性名不能写错 -->
    <!-- 连接池 -->
    <property name="dataSource" ref="dbcp"></property>
    <!-- 加载 MyBatis 的配置文件 -->
    <property name="configLocation"
        value="mybatis/SqlMapConfig.xml"></property>
</bean>

<!-- 配置 Dao -->
<bean id="userDao" class="ssm.dao.UserDaoImpl">
    <property name="sqlSessionFactory" ref="factory"></property>
</bean>
</beans>

```

四、管理 mapper

下面介绍三种方法：

1. 传统的 Dao

(1) 修改 Dao 接口实现类，继承 SqlSessionSupport

```

// 接口实现类
// 继承 SqlSessionDaoSupport, 这个类已经有了一个 SqlSessionFactory 属性
public class UserDaoImpl extends SqlSessionDaoSupport implements
    UserDao{

    @Override
    public User findUserById(int id) throws Exception {
        SqlSession session = this.getSqlSession();
        User user = session.selectOne("test.findUserById", id);
        //session.close(); // Spring 会自动关闭资源
        return user;
    }
}

```

(2) 创建映射文件，放在 sqlmap 目录下。并且在 SqlMapConfig.xml 中进行配置。注意此时 SqlMapConfig.xml 文件中的配置发生变化，去掉了很多配置：

```

<?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>

```

```

<!-- 定义别名 -->
<typeAliases>
    <typeAlias alias="user" type="ssm.po.User"></typeAlias>
</typeAliases>

<!-- 加载映射文件 -->
<mappers>
    <mapper resource="sqlmap/User.xml"></mapper>
</mappers>
</configuration>

```

(3) 测试

```

public class UserDaoImplTest {

    private ApplicationContext context;

    @Before
    public void setup() throws Exception {
        context = new
            ClassPathXmlApplicationContext("spring/applicationContext.xml");
    }

    @Test
    public void test() throws Exception {
        UserDao userDao = (UserDao) context.getBean("userDao");
        System.out.println(userDao.findUserById(10));
    }

}

```

2. mapper 代理

- (1) 创建接口、映射文件
- (2) 在 applicationContext.xml 中配置 mapper

```

<!-- 配置 mapper -->
<!-- 这个 bean 用来获取 UserMapper -->
<bean id="userMapper"
    class="org.mybatis.spring.mapper.MapperFactoryBean">
    <!-- 指定 mapper 接口 -->
    <property name="mapperInterface"
        value="ssm.mapper.UserMapper"></property>
    <!-- 配置 SqlSessionFactory -->
    <property name="sqlSessionFactory" ref="factory"></property>
</bean>

```

(3) 测试

```

public class UserMapperTest {

    private ApplicationContext context;

    @Before
    public void setUp() throws Exception {
        context = new
            ClassPathXmlApplicationContext("spring/applicationContext.xml");
    }

}

```



```

@Test
public void test() throws Exception {
    UserMapper mapper= (UserMapper)
        context.getBean("userMapper");
    System.out.println(mapper.findUserById(10));
}
}

```

这种方法存在的问题是，针对每一个 mapper 都要配置一次，很繁琐。

3. 批量扫描 mapper 接口（推荐使用）

在 applicationContext.xml 中配置：

```

<!-- 批量扫描包中的所有 mapper 接口，自动创建代理对象并在 Spring 容器中注册 -->
<!-- 仍然需要保持接口文件和映射文件在同一个包下而且名字相同 -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <!-- 指定需要扫描的包，如果有多个包，包与包之间用半角逗号隔开 -->
    <property name="basePackage" value="ssm.mapper"></property>
    <!-- 注意这里的属性名 -->
    <property name="sqlSessionFactoryBeanName" value="factory">
</property>
</bean>

```

现在可以将 SqlMapConfig.xml 中的以下配置去掉：

```

<!-- 加载映射文件 -->
<mappers>
    <package name="ssm.mapper"/>
</mappers>

```

通过这种方法得到的 bean 的 id 是接口名的第一个单词的首字母小写，如接口名为 UserMapper，那么扫描得到的 bean 的 id 就是 userMapper。

逆向工程

2018年6月26日 18:13

MyBatis 提供了 MyBatis Generator, 可以针对单表自动生成 MyBatis 运行所需的代码, 包括接口文件、映射文件、POJO 等等。常见的用途是由数据库的表生成 Java 代码。

1. 使用方式

- From the command prompt with an XML configuration
- As an Ant task with an XML configuration
- As a Maven Plugin
- From another Java program with an XML configuration
- From another Java program with a Java based configuration
- As an Eclipse Feature

建议通过 XML 配置文件和 Java 程序来使用这个工具, 所以要导入 jar 包 mybatis-generator-core。

2. 配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
  PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
  "http://mybatis.org/dtd/mybatis-generator-config 1.0.dtd">

<generatorConfiguration>
  <context id="testTables" targetRuntime="MyBatis3">
    <commentGenerator>
      <!-- 是否去除自动生成的注释 true: 是: false:否 -->
      <property name="suppressAllComments" value="true" />
    </commentGenerator>

    <!-- 数据库连接的信息: 驱动类、连接地址、用户名、密码 -->
    <jdbcConnection driverClass="com.mysql.jdbc.Driver"
      connectionURL="jdbc:mysql://localhost:3306/mybatis"
      userId="root"
      password="mysql">
    </jdbcConnection>
    <!-- <jdbcConnection driverClass="oracle.jdbc.OracleDriver"
      connectionURL="jdbc:oracle:thin:@127.0.0.1:1521:yycg"
      userId="yycg"
      password="yycg">
    </jdbcConnection> -->

    <!-- 默认false, 把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer, 为
    true时把JDBC DECIMAL 和
      NUMERIC 类型解析为java.math.BigDecimal -->
    <javaTypeResolver>
      <property name="forceBigDecimals" value="false" />
    </javaTypeResolver>

    <!-- targetProject:生成PO类的位置 -->
    <javaModelGenerator targetPackage="cn.itcast.ssm.po"
      targetProject=".\\src">
      <!-- enableSubPackages:是否让schema作为包的后缀 -->
      <property name="enableSubPackages" value="false" />
      <!-- 从数据库返回的值被清理前后的空格 -->
      <property name="trimStrings" value="true" />
    </javaModelGenerator>
  </context>
</generatorConfiguration>
```

```

</javaModelGenerator>
<!-- targetProject: mapper映射文件生成的位置 -->
<sqlMapGenerator targetPackage="cn.itcast.ssm.mapper"
    targetProject=".\\src">
    <!-- enableSubPackages: 是否让schema作为包的后缀 -->
    <property name="enableSubPackages" value="false" />
</sqlMapGenerator>
<!-- targetPackage: mapper接口生成的位置 -->
<javaClientGenerator type="XMLMAPPER"
    targetPackage="cn.itcast.ssm.mapper"
    targetProject=".\\src">
    <!-- enableSubPackages: 是否让schema作为包的后缀 -->
    <property name="enableSubPackages" value="false" />
</javaClientGenerator>
<!-- 指定数据库表 -->
<table tableName="items"></table>
<table tableName="orders"></table>
<table tableName="orderdetail"></table>
<table tableName="user"></table>
<!-- <table schema="" tableName="sys_user"></table>
<table schema="" tableName="sys_role"></table>
<table schema="" tableName="sys_permission"></table>
<table schema="" tableName="sys_user_role"></table>
<table schema="" tableName="sys_role_permission"></table> -->

<!-- 有些表的字段需要指定java类型
<table schema="" tableName="">
    <columnOverride column="" javaType="" />
</table> -->

</context>
</generatorConfiguration>

```

数据库表、映射文件位置、mapper 接口位置、POJO 包的位置这四项配置最为关键。

3. MBG 程序

```

List<String> warnings = new ArrayList<String>();
boolean overwrite = true;
File configFile = new File("generatorConfig.xml");
ConfigurationParser cp = new ConfigurationParser(warnings);
Configuration config = cp.parseConfiguration(configFile);
DefaultShellCallback callback = new DefaultShellCallback(overwrite);
MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config,
    callback, warnings);
myBatisGenerator.generate(null);

```

执行完这段代码之后就自动生成了相应的代码。

4. 使用生成的代码

注意，为了安全，一般生成代码的工程和实际上正在开发的工程不是同一个，生成代码之后再将这些代码拷贝到需要它们的工程中。

```

@Test
public void test() {
    // 根据主键查询
    System.out.println(mapper.selectByPrimaryKey(10));
    UserExample userExample = new UserExample();
    UserExample.Criteria criteria = userExample.createCriteria();

    // 自定义条件查询

```

```
// 通过 UserExample.Criteria 对象来构造查询条件
criteria.andAddressEqualTo("北京"); // 查询地址为北京的用户
List<User> users = mapper.selectByExample(userExample);
System.out.println(users);

// 插入记录
User user = new User();
user.setUsername("王五");
user.setAddress("河南");
user.setSex("2");
mapper.insert(user);

// 更新记录
mapper.updateByPrimaryKey(user); // 根据传入的值更新相应的字段
mapper.updateByPrimaryKeySelective(user); // 传入的值不为空才更新相应的字段
}
```

基于注解的开发

2018年6月27日 9:22

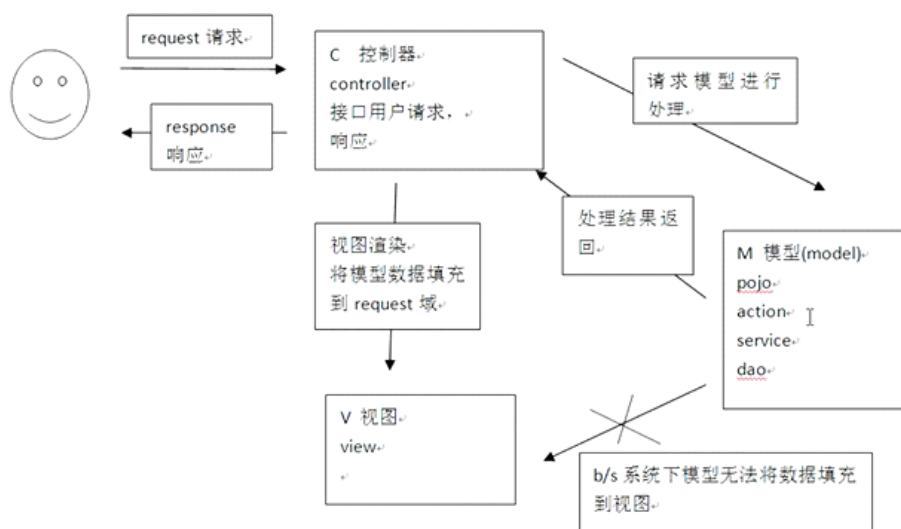
<http://www.mybatis.org/mybatis-3/zh/java-api.html#sqlSessions>

概述

2018年6月22日 14:53

Spring MVC 是 Spring 的一个框架，无需通过中间层就可以和 Spring 进行整合。

一、MVC 在 browser/server 模式中的应用



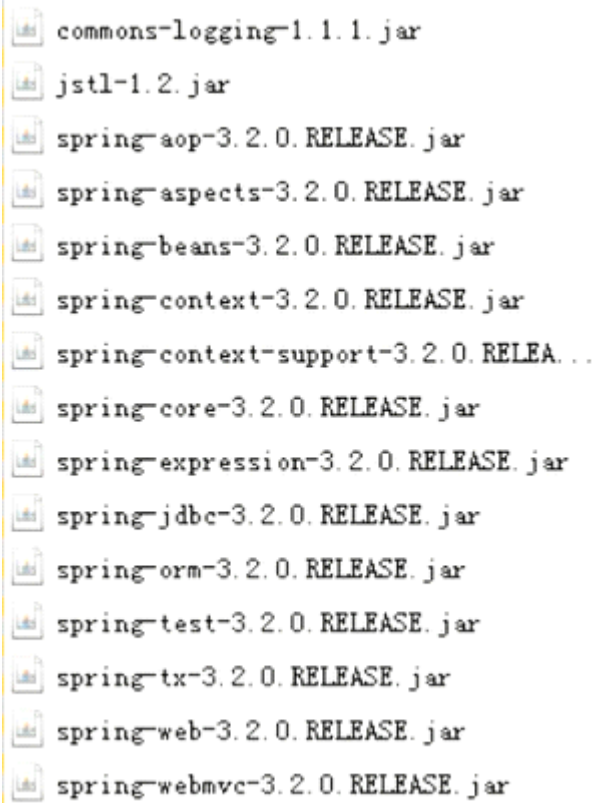
二、Spring MVC 的工作流程

1. 客户端向前端控制器 DispatcherServlet 发请求
2. 前端控制器请求处理器映射器 HandlerMapping 查找处理器 Handler
3. 处理器映射器向前端控制器返回处理器
4. 前端控制器调用处理器适配器 HandlerAdapter 执行处理器
5. 处理器向处理器适配器返回 ModelAndView，这是 Spring MVC 的一个底层对象，包括 view 和 model
6. 处理器适配器向前端控制器返回 ModelAndView
7. 前端控制器请求视图解析器解析逻辑视图，根据逻辑视图生成真正的视图
8. 视图解析器解析完成后向前端控制器返回视图，前端控制器进行视图渲染，将 ModelAndView 中的数据填充到 request 中
9. 前端控制器向客户端响应

三、Spring MVC 的组件

1. 前端控制器 DispatcherServlet
接受请求，响应结果
2. 处理器映射器 HandlerMapping
根据请求的 URL 查找 Handler，可以根据配置文件查找，也可以根据注解查找
3. 处理器适配器 HandlerAdapter
按照特定的规则执行 Handler，编写 Handler 的时候要按照 HandlerAdapter 的要求，这样适配器才可以正确执行 Handler
4. 处理器 Handler——**开发时的重点**
需要按照 HandlerAdapter 的要求编写。
5. 视图解析器 View Resolver
根据逻辑视图名解析成真正的视图，封装在 view 中
6. 视图 view
view 是一个接口，实现类支持不同类型的视图

四、需要的 jar 包



commons-logging-1.1.1.jar
jstl-1.2.jar
spring-aop-3.2.0.RELEASE.jar
spring-aspects-3.2.0.RELEASE.jar
spring-beans-3.2.0.RELEASE.jar
spring-context-3.2.0.RELEASE.jar
spring-context-support-3.2.0.RELEASE.jar
spring-core-3.2.0.RELEASE.jar
spring-expression-3.2.0.RELEASE.jar
spring-jdbc-3.2.0.RELEASE.jar
spring-orm-3.2.0.RELEASE.jar
spring-test-3.2.0.RELEASE.jar
spring-tx-3.2.0.RELEASE.jar
spring-web-3.2.0.RELEASE.jar
spring-webmvc-3.2.0.RELEASE.jar

jstl 是写 JSP 用的

基础

2018年6月23日 16:05

一、配置前端控制器

前端控制器其实就是一个 servlet，在 `web.xml` 中配置：

```
<!-- 配置前端控制器 -->
<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 配置初始化参数 -->
    <init-param>
        <!-- 这是配置文件，配置处理器映射器、适配器等 -->
        <!-- 如果不配置这个参数，那么程序会默认加载 /WEB-INF/servletname-
            servlet.xml，在这里也就是指 springmvc-servlet.xml -->
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:springmvc.xml</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <!-- 三种配置方法 -->
    <!-- 1 *.action 所有访问以 .action 结尾的资源都会由 DispatcherServlet
        解析 -->
    <!-- 2 / 所有访问都由 DispatcherServlet 解析，但是注意对静态资源的访问不
        能由 DispatcherServlet
            解析，这种方法能实现 RESTful 风格的 URL -->
    <url-pattern>*.action</url-pattern>
</servlet-mapping>
```

在 `classpath` 下的 `springmvc.xml` 中进行以下配置。如果没有专门进行配置，那么 Spring 会在 `/org/springframework/web/servlet/DispatcherServlet.properties` 文件中找到默认的配置。

使用 Spring MVC 之后的 Spring 全约束：

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


二、配置处理器适配器

所有的处理器适配器都实现 HandlerAdapter 接口。

(1) SimpleControllerHandlerAdapter 适配器

```
<!-- 配置处理器适配器 -->
<bean
    class="org.springframework.web.servlet.mvc.SimpleControllerHandler
Adapter"></bean>
```

在 SimpleControllerHandlerAdapter 源码中看到有一个 supports 方法，

```
public boolean supports(Object handler) {
    return (handler instanceof Controller);
}
```

说明这个适配器能执行实现了 Controller 接口的 Handler。

```
@FunctionalInterface
public interface Controller {

    @Nullable
    ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws Exception;
}
```

(2) HttpRequestHandlerAdapter 适配器

这个适配器能执行实现了 HttpRequestHandler 接口的处理器。

```
<bean
    class="org.springframework.web.servlet.mvc.HttpRequestHandler
Adapter"></bean>
```

实现这个接口的处理器可以用来返回 JSON 等格式的数据。

三、编写 Handler

有两种方法：

(1) 实现 Controller 接口

```
public class ItemsController1 implements Controller{

    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        // 调用 service 查找数据库库，查询商品列表
        List<Items> itemsList = new ArrayList<>();
        // 这里就用静态数据进行模拟
        Items items_1 = new Items();
        items_1.setName("联想笔记本");
        items_1.setPrice(6000f);
        items_1.setDetail("ThinkPad T430 联想笔记本电脑! ");

        Items items_2 = new Items();
        items_2.setName("苹果手机");
        items_2.setPrice(5000f);
        items_2.setDetail("iphone6苹果手机! ");

        itemsList.add(items_1);
        itemsList.add(items_2);
    }
}
```

```

        ModelAndView mav = new ModelAndView(); // 返回 ModelAndView
        mav.addObject("itemsList", itemsList); // 相当于
        request.setAttribute

        mav.setViewName("/WEB-INF/jsp/items/itemsList.jsp"); // 指定
        视图

        return mav;
    }
}

```

编写完之后配置处理器：

```

<!-- 配置处理器 -->
<bean name="/queryItems.aciton"
      class="ssm.controller.ItemsController1"></bean>

```

(2) 实现 `HttpRequestHandler` 接口

```

public class ItemsController2 implements HttpRequestHandler {

    @Override
    public void handleRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

        // 调用 service 查找数据库库, 查询商品列表
        List<Items> itemsList = new ArrayList<>();
        // 这里就用静态数据进行模拟
        Items items_1 = new Items();
        items_1.setName("联想笔记本");
        items_1.setPrice(6000f);
        items_1.setDetail("ThinkPad T430 联想笔记本电脑! ");

        Items items_2 = new Items();
        items_2.setName("苹果手机");
        items_2.setPrice(5000f);
        items_2.setDetail("iphone6苹果手机! ");

        itemsList.add(items_1);
        itemsList.add(items_2);

        request.setAttribute("itemsList", itemsList); // 设置模型数据
        request.getRequestDispatcher("/WEB-
        INF/jsp/items/itemsList.jsp"); // 设置转发的视图
    }
}

```

写完之后同样需要进行 xml 配置。

四、配置处理器映射器

处理器映射器可以有多个，而且配置方式也不唯一。所有映射器都实现了 `HandlerMapping` 接

口。

```
<!-- 配置处理器映射器 -->
<!-- 处理器的 beanname 就是 URL, 映射器就会按照 URL 进行查找 -->
<bean
    class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
<!-- 简单 URL 映射 -->
<!-- prop 标签的值就是之前配置处理器时赋予的 id -->
<bean
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/queryItems1.action">itemsController1</prop>
            <prop key="/queryItems2.action">itemsController1</prop>
        </props>
    </property>
</bean>
```

五、配置视图解析器

```
<!-- 配置视图解析器 -->
<bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <!-- 前缀 -->
    <property name="prefix" value="/WEB-INF/jsp/"></property>
    <!-- 后缀 -->
    <property name="suffix" value=".jsp"></property>
</bean>
```

配置前缀和后缀之后, 在 Controller 中指定视图时就无需写完整的路径了, 因为此时已经指定了前缀和后缀。

```
mav.setViewName("items/itemsList");
```

默认使用 jstl 标签解析 JSP 页面, 所以如果要用 JSP 页面, 就要注意在 classpath 下包含 jstl 包。

六、常见错误 404

如果页面上只显示了404, 说明配置给处理器映射器的 URL 本身就错了。如果404后面还跟着一个 URL, 那么一般就是 JSP 的 URL 设置错了。

使用注解（实际开发中首选）

2018年6月23日 19:39

注意：如果使用注解的话，那么处理器、处理器映射器、处理器适配器都要统一使用注解。而且处理器映射和适配器要成对出现。
前端控制器和视图解析器还是要在 xml 中配置。

一、处理器映射器

在 Spring 3.1 之前使用

org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping, 在 3.1 之后使用

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping
。

二、处理器适配器

在 3.1 之前使用

org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter, 在 3.1 之后使用

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter
。

三、配置方法

在 springmvc.xml 中进行配置：

```
<!-- 注解处理器映射器 -->
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMa
ppingHandlerMapping"></bean>
<!-- 注解处理器适配器 -->
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMa
ppingHandlerAdapter"></bean>
```

但是，在实际开发中会使用另一个更简便的方法：

```
<!-- 开启注解驱动，可以直接代替上面的配置 -->
<mvc:annotation-driven></mvc:annotation-driven>
```

这个标签会默认加载很多的参数绑定方法，比如 JSON 转义解析器。

四、编写 Handler

使用注解之后，Handler 类只需加上注解 **@Controller** 而无需再去实现接口。同时也需要给方法加上注解 **@RequestMapping**，这样一个 URL 就对应一个方法。为了便于维护，通常 URL 就设置为方法名。

```
@Controller
public class ItemsController3{

    // 这个 URL 不加 .action 也行
    @RequestMapping("/queryItems")
    public ModelAndView queryItems() {

        // 调用 service 查找数据库库，查询商品列表
```

```

List<Items> itemsList = new ArrayList<>();
// 这里就用静态数据进行模拟
Items items_1 = new Items();
items_1.setName("联想笔记本");
items_1.setPrice(6000f);
items_1.setDetail("ThinkPad T430 联想笔记本电脑! ");

Items items_2 = new Items();
items_2.setName("苹果手机");
items_2.setPrice(5000f);
items_2.setDetail("iphone6苹果手机! ");

itemsList.add(items_1);
itemsList.add(items_2);

ModelAndView mav = new ModelAndView(); // 返回 ModelAndView
mav.addObject("itemsList", itemsList); // 相当于
request.setAttribute

mav.setViewName("/WEB-INF/jsp/items/itemsList.jsp"); // 指定
视图

return mav;
}
}

```

由于现在是基于注解的开发，所以可以直接开启注解扫描，而无需再手工配置 Handler。在 springmvc.xml 中加上标签：

```

<!-- 开启注解扫描 -->
<context:component-scan base-package="ssm.controller">
</context:component-scan>

```

注解开发详解

2018年6月27日 16:23

一、@RequestMapping

1. 通常这个方法用在 controller 的方法上，用来设置该方法应该处理哪个 URL。

2. 窄化请求映射

这个注解也可用在 controller 类上，相当于增加了一个根路径，比如：

```
@Controller
@RequestMapping("/items")
public class ItemsController {

    @Resource(name="itemsService")
    private ItemsService service;

    // 商品查询页面展示
    @RequestMapping("/queryItems")
    public ModelAndView queryItems() throws Exception {
        //..
    }
}
```

这样设置之后 queryItems 处理的就是对 .../items/queryItems 的访问。这种设置叫做窄化请求映射，便于对 URL 进行分类管理。

3. 限制 HTTP 请求方法

比如限制只能用 POST，那么用 GET 方法时服务器的响应就是 405。

```
@RequestMapping(value="/editItems",method= {RequestMethod.POST})
```

method 属性接受的是一个数组，也就意味着可以允许有多种请求方法。

二、controller 方法的返回值

1. 返回 ModelAndView

2. 返回 String

(1) 如果返回的是 String，那么这个 String 表示的是逻辑视图名，视图（页面）的真正路径是前缀+逻辑视图名+后缀。此时方法的参数值要加上一个 Model。

```
public String editItems1(Model model) throws Exception{
    ItemsCustom itemsCustom = service.findItemsById(1);

    // 相当于 ModelAndView.addObject 方法，将数据传到页面
    model.addAttribute("itemsCustom", itemsCustom);

    return "items/editItems";
}
```

(2) 第二种含义，表示重定向的 URL。

重定向时地址栏的 URL 发生改变，原先保存在 request 中的数据无法传到重定向后的资源中。

```
@RequestMapping("/editItems2")
public String editItems2() throws Exception{
    return "redirect:queryItems.action"; // 重定向的 URL
}
```

```
}
```

(3) 第三种含义，表示转发的 URL。

转发时地址栏的 URL 不变，request 中携带的数据可以跟着一起转发。

```
@RequestMapping("/editItems3")
public String editItems3() throws Exception{
    return "forward:queryItems.action"; // 转发的 URL
}
```

3. 返回 void

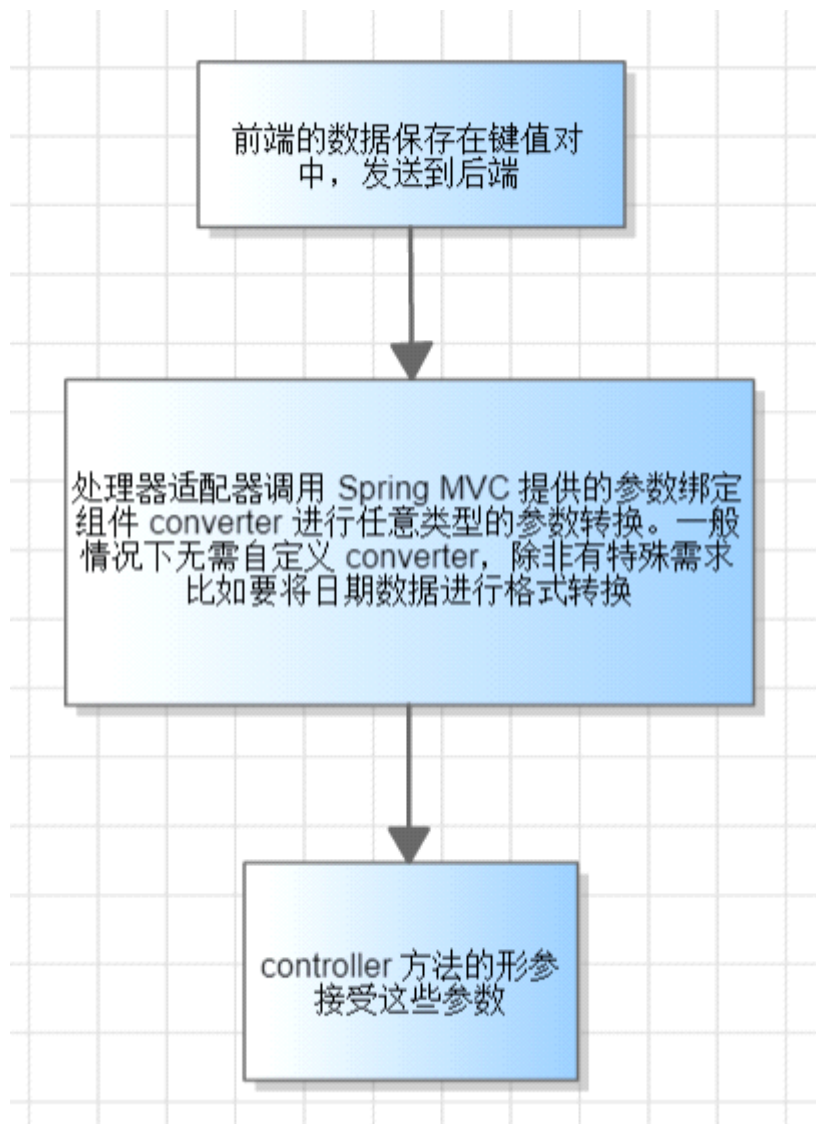
controller 方法的形参可以为 request 和 response，然后使用这两个对象指定响应的结果。

- (1) 使用 request 进行页面转发
- (2) 使用 response 重定向
- (3) 用 response 响应指定的数据，比如 JSON:

```
response.setCharacterEncoding("UTF-8");
response.setContentType("application/json;charset=UTF-8");
response.getWriter().write("some JSON data...");
```

三、参数绑定

在 Spring MVC 中，页面提交的数据要传到 controller 方法的形参中。



1. controller 方法默认支持的形参类型

- (1) HttpServletRequest
- (2) HttpServletResponse
- (3) HttpSession
- (4) Model/ModelMap: Model 是接口, ModelMap 是实现类, 作用都是将数据填充到 request 域。
- (5) 简单类型

在参数绑定时, 如果形参是这些类型, 那么 Spring MVC 就直接进行参数绑定。

2. 简单类型形参

- (1) 直接绑定

HTTP 请求报文中携带了一个名为 id 的参数, 类型是整数, 那么 controller 方法的形参就定义为 Integer id。注意, 形参名和请求报文中的参数名必须一致。

- (2) 使用 @RequestParam

```
@RequestMapping(value="/editItems",method={RequestMethod.POST,RequestMethod.GET})
public ModelAndView
editItems(@RequestParam(value="id",required=true,defaultValue="2")
Integer id_) throws Exception {

    // 调用 service 查询商品信息
    ItemsCustom itemsCustom = service.findItemsById(id_);

    // 返回 ModelAndView
    ModelAndView mav = new ModelAndView();

    // 将商品信息填充到 model 中
    mav.addObject("itemsCustom",itemsCustom);

    // 设定商品修改页面
    mav.setViewName("items/editItems");

    return mav;
}
```

这个注解用在方法的形参上, 三个属性的含义:

- value: 请求报文中携带的参数的名字
- required: 该参数是否必需, 如果请求报文中没有传入这个参数, 那么就会报错
- defaultValue: 参数的默认值

3. POJO 形参

- (1) 直接绑定

如果页面中 <input name=""> name 的值和 POJO 对象的属性名一致, 那么就可以实现绑定。

```
// 提交修改过后的商品信息
@RequestMapping(value="/submitItems",method=RequestMethod.POST)
// 这里使用了 ItemsQueryVo, 那么传入的参数的名称就应该是 itemsCustom.id 这类的, 因为 ItemsQueryVo 中有一个 itemsCustom 属性
public ModelAndView submitItems(ItemsQueryVo vo) throws Exception{

    // 调用 service 更新商品信息
    service.updateItems(vo.getItemsCustom().getId(),
vo.getItemsCustom());

    // 返回 ModelAndView
    ModelAndView mav = new ModelAndView();
}
```



```

        mav.setViewName("items/success");

        return mav;
    }

```

可能出现的无法绑定问题:

设置 <form> 的属性 enctype=" multipart/form-data" 时会导致参数绑定失败, 要改为 enctype=" x-www-form-urlencoded" 。原因不详。

(2) 自定义参数绑定组件

比如说进行日期格式的转换。先实现一个转换类, 目标类型和 POJO 中的属性类型一致。

```

// 自定义参数绑定组件, Converter<S, T>, S 是原始类型, T 是目标类型
public class CustomDateConverter implements Converter<String, Date> {

    @Override
    public Date convert(String source) {
        // 将日期转换为 yyyy-MM-dd HH:mm:ss
        SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
        try {
            return format.parse(source);
        } catch (ParseException e) {
            e.printStackTrace();
        }
        return null;
    }
}

```

然后在 springmvc.xml 中进行配置, 本质上是注入到处理器适配器中:

```

<!-- 开启自定义参数绑定 -->
<mvc:annotation-driven
    conversion-service="conversionService"></mvc:annotation-driven>

<!-- 配置参数绑定组件 -->
<bean id="conversionService"
    class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
    <!-- 自定义参数绑定组件 -->
    <property name="converters">
        <list>
            <bean
                class="ssm.controller.converter.CustomDateConverter">
            </bean>
        </list>
    </property>
</bean>

```

4. 数组形参

当页面穿过来的数据有多个时, 可以用数组接受, 参数名和形参名对应, 就可直接绑定

```

@RequestMapping("/deleteItems")
public ModelAndView deleteItems(Integer[] ids) throws Exception {

    //List<ItemsCustom> itemsList = service.deleteItemsByIds(ids);
    List<ItemsCustom> itemsList = service.queryItemsList(null);
    ModelAndView mav = new ModelAndView(); // 返回 ModelAndView
    mav.addObject("itemsList", itemsList); // 相当于
    request.setAttribute

```

```

        mav.setViewName("items/itemsList"); // 指定视图

        return mav;
    }
}

```

前端页面的定义:

```

<td><input type="checkbox" name="ids" value="${item.id}"/></td>

```

5. List 形参

仍然需要注意前端参数名和形参名的对应。比如列表为 `List<ItemsCustom> itemsList`, 那么传过来的参数名就应该是 `itemsList[0].name` 之类的。

```

// 批量修改商品信息后提交, 提交的信息保存在 List 中, 此时 ItemsQueryVo 中有一个
// 属性 itemsList, 这是 List<ItemsCustom> 类型的
@RequestMapping("/editItemsSubmit")
public ModelAndView editItemsSubmit(ItemsQueryVo vo) throws Exception
{

    ModelAndView mav = new ModelAndView(); // 返回 ModelAndView
    // 相关业务逻辑
    return mav;
}

```

页面定义:

```

<c:forEach items="${itemsList}" var="item" varStatus="status">
<tr>

    <!-- 把参数值保存在 List 中 -->
    <td><input name="itemsList[${status.index}].name"
value="${item.name}"/></td>
    <td><input name="itemsList[${status.index}].price"
value="${item.price}"/></td>
    <td><input name="itemsList[${status.index}].createTime"
value="<fmt:formatDate value="${item.createTime}" pattern="yyyy-MM-dd HH:mm:ss"/>"/></td>
    <td><input name="itemsList[${status.index}].detail"
value="${item.detail}"/></td>

</tr>
</c:forEach>

```

6. Map 形参

在包装类中定义 Map 对象，并添加 get/set 方法，action 使用包装对象接收。

包装类中定义 Map 对象如下：

```
Public class QueryVo {  
private Map<String, Object> itemInfo = new HashMap<String, Object>();  
    //get/set 方法...  
}
```

页面定义如下：

```
<tr>  
<td>学生信息: </td>  
<td>  
姓名: <input type="text" name="itemInfo['name']"/>  
年龄: <input type="text" name="itemInfo['price']"/>  
.. .. .  
</td>  
</tr>
```

参数检验

2018年6月28日 10:41

在实际开发中，通常要对参数进行检验，一般而言前端会进行检验，但是如果要求较高的安全性就要在后端进行检验。

- 表示层：用 controller 检验页面请求参数的合法性，不区分客户端类型（浏览器端、移动端、远程调用）
- 业务层：检验用在 service 接口中的关键参数，这是最常进行的检验
- 持久层：一般而言不进行检验

使用 hibernate-validator 这一第三方库进行检验。这个 jar 包中已经提供了依赖包。

一、配置方法（在 springmvc.xml 中进行配置）

1. 配置校验器

```
<!-- 参数校验器 -->
<bean id="validator"
      class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
    <!-- hibernate 校验器 -->
    <property name="providerClass"
              value="org.hibernate.validator.HibernateValidator" />
    <!-- 指定校验使用的资源文件，在文件中配置校验错误信息，如果不指定则默认使用
    classpath 下的 ValidationMessages.properties -->
    <property name="validationMessageSource" ref="messageSource" />
</bean>

<!-- 校验错误信息配置文件 -->
<bean id="messageSource"
      class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <!-- 资源文件名 -->
    <property name="basenames">
        <list>
            <value>classpath:CustomValidationMessages</value>
        </list>
    </property>
    <!-- 资源文件编码格式 -->
    <property name="fileEncodings" value="utf-8" />
    <!-- 对资源文件内容缓存时间，单位秒 -->
    <property name="cacheSeconds" value="120" />
</bean>
```

2. 将校验器注入到处理器适配器中

```
<!-- 开启注解驱动，配置处理器适配器和映射器 -->
<!-- 注入自定义参数绑定组件 -->
<!-- 注入校验器 -->
<mvc:annotation-driven
    conversion-service="conversionService" validator="validator">
</mvc:annotation-driven>
```

二、在 POJO 中添加校验规则

```
public class Items {
```

```

    private Integer id;

    // 名称长度为1-30个字符
    @Size(min=1,max=30,message="{items.name.length.error}")
    private String name;

    private Float price;

    // 非空检验
    @NotNull(message="{items.createTime.isNull}")
    private Date createTime;

    //...
}

// 这个对象是提供给表现层的
public class ItemsQueryVo {

    // 原始商品信息
    private Items items;

    // 扩展后的商品信息
    @Valid
    private ItemsCustom itemsCustom;

    //...
}

```

要进行校验的方法的参数类型是 `ItemsQueryVo`，而页面传过来的类型是 `ItemsCustom`，`ItemsCustom` 继承自 `Items` 所以也受到同样的校验条件的约束，因此这里只需给 `itemsCustom` 属性加上 `@Valid`。

同时在 `ValidationMessages.properties` 文件中配置提示信息：

```

# validation error information
items.name.length.error=the length of name must be more than 1
character and less than 30 characters
items.createTime.isNull=the create time must be not null

```

三、进行校验

在控制器类中，作如下修改：

```

@RequestMapping(value = "/submitItems", method = RequestMethod.POST)
public ModelAndView submitItems(@Valid ItemsQueryVo vo, BindingResult
bindingResult) throws Exception {

    List<ObjectError> errors = bindingResult.getAllErrors(); // 获取错
    误信息

    ModelAndView mav = new ModelAndView();
    if (bindingResult.hasErrors()) {
        mav.addObject("errors", errors);
        mav.addObject("itemsCustom", vo.getItemsCustom());
        mav.setViewName("items/editItems");
    } else {
        // 调用 service 更新商品信息
        service.updateItems(vo.getItemsCustom().getId(),
            vo.getItemsCustom());

        // 返回 ModelAndView
        List<ItemsCustom> itemsList = service.queryItemsList(vo);
    }
}

```

```

        mav.addObject("itemsList", itemsList); // 相当于
        request.setAttribute
        mav.setViewName("items/itemsList"); // 指定视图
    }

    return mav;
}

```

注意，添加了 @Valid 的参数的后面必须要跟着一个 BindingResult 类型的参数，这个参数用来接受错误信息。

四、分组检验

一个 POJO 中可能有多个属性需要检验，但是不同的方法可能只需要检验个别属性，这时需要进行分组校验。

1. 定义校验分组接口

```

// 不需要有任何方法
// 这个分组只检验名称
public interface ValidateName {

}

// 这个分组只校验日期时间
public interface ValidateDateTime {

}

```

2. 在 POJO 的注解中加上分组归属

```

// 名称长度为1-30个字符
@Size(min=1,max=30,message="{items.name.length.error}",groups=
{ValidateName.class})
private String name;

// 非空检验
@NotNull(message="{items.createTime.isNull}",groups=
{ValidateDateTime.class})
private Date createTime;

```

3. 指定 controller 方法使用哪个分组

注意此时使用的注解是 @Validated，这个注解是 Spring 对 @Valid 的封装，它才能实现校验分组。

```

@Validated(value= {ValidateName.class,ValidateDateTime.class})

```

四、数据回显

1. POJO 数据

数据回显是指未经过检验的数据重新显示到表单中。Spring 默认将 POJO 的数据保存在 request 域中，key 为类名第一个单词的首字母小写，因此在页面中可以用下面的代码顺利地取出数据：

```

<input type="hidden" name="itemsCustom.id"
value="${itemsCustom.id }"/>

```

但是如果 `value="${itemsCustom.id}"` 中改变了变量名，比如改成 `items.id`，那么这个时候就需要在 `controller` 方法中指定数据回显时保存在 `request` 中的 `key`。

```
public ModelAndView submitItems(@ModelAttribute("items")
    @Validated(value = { ValidateName.class,
        ValidateDateTime.class }) ItemsQueryVo vo, BindingResult
        bindingResult) throws Exception {
```

注解 `@ModelAttribute` 还可以用在整个方法上，这样就可以把方法的返回值也保存在 `request` 域中，`key` 就是这个注解指定的值。

```
// 查询商品类型
@ModelAttribute("itemsTypes")
public Map<String,String> getItemsTypes(){
    // 这里先用静态数据进行模拟
    Map<String,String> map = new HashMap<>();
    map.put("1", "笔记本");
    map.put("2", "笔记本");
    map.put("3", "手机");
    map.put("4", "手机");
    return map;
}
```

在页面中就可以获取这些键值对：

```
<select name="itemtype">
    <c:forEach items="${itemsTypes}" var="itemtype">
        <option value="${itemtype.key}">${itemtype.value}
        </option>
    </c:forEach>
</select>
```

其实，不用注解也能实现回显：

```
if (bindingResult.hasErrors()) {
    mav.addObject("errors", errors);
    mav.addObject("itemsCustom", vo.getItemsCustom());
    mav.setViewName("items/editItems");
}
```

2. 简单数据

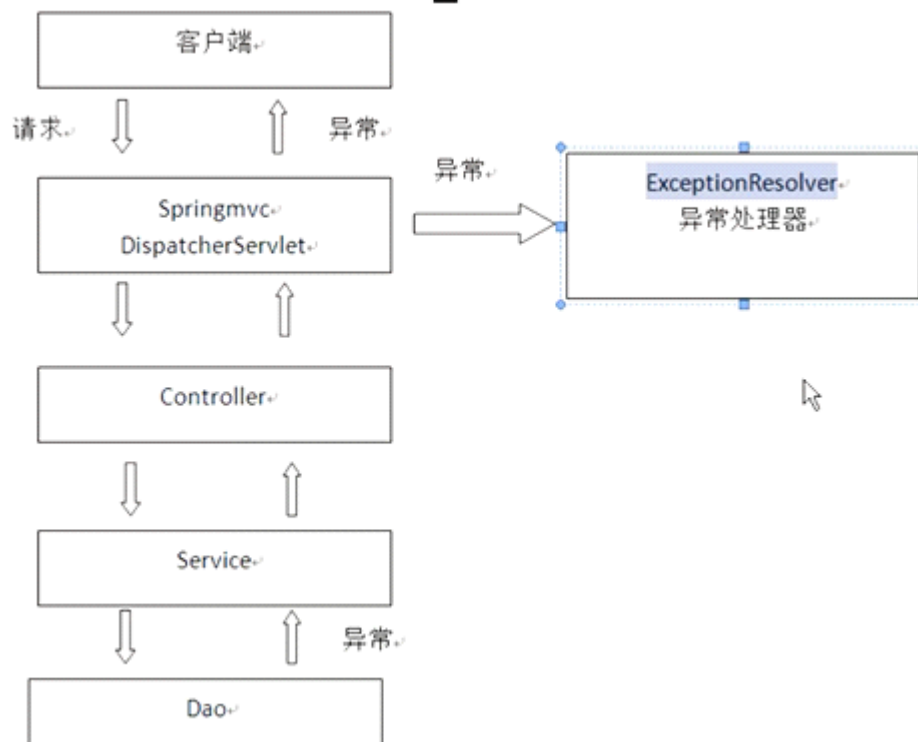
只能通过把数据加到 `ModelAndView` 或者 `Model` 中进行回显：

```
mav.addObject("id", id);
```

异常处理

2018年6月28日 16:08

一、思路



1. Spring 提供了一个全局异常处理进行统一异常处理，一个系统中有一个。
2. 尽量不要把异常抛给客户端

二、全局异常处理器

发生异常后手动抛出，从下往上抛，抛给前端控制器之后交给异常处理器处理。异常处理器的处理流程是：

- (1) 解析异常类型
- (2) 如果该异常是系统定义的，直接获取异常信息并显示在页面上。
- (3) 如果该异常不是系统定义的，构造一个自定义的异常类型，显示信息为未知错误。

1. 编写异常处理器

实现接口 `HandlerExceptionResolver` 之后就是全局异常处理器。

```
public class CustomExceptionHandler implements
HandlerExceptionResolver {

    @Override
    public ModelAndView resolveException(HttpServletRequest request,
        HttpServletResponse response, Object obj,
        Exception ex) {

        CustomException customEx = null;
        if(ex instanceof CustomException) { // 系统自定义异常
            customEx = (CustomException) ex;
        } else { // 不是系统自定义异常
            customEx = new CustomException("unknown");
        }

        String message = customEx.getMessage();
        ModelAndView mav = new ModelAndView();
```



```

        mav.addObject("message", message);
        mav.setViewName("error"); // 定向到错误页面

        return mav;
    }
}

```

2. 在 springmvc.xml 中配置

```
<bean class="ssm.exception.CustomExceptionResolver"></bean>
```

三、测试

如果是手动抛出的异常，那么就说明是自定义的异常；否则是运行时异常。

可以在 controller 方法中抛出：

```

@RequestMapping(value = "/editItems", method = { RequestMethod.POST,
RequestMethod.GET })
public ModelAndView editItems(@RequestParam(value = "id", required =
true, defaultValue = "2") Integer id_)
    throws Exception {

    // 调用 service 查询商品信息
    ItemsCustom itemsCustom = service.findItemsById(id_);

    if(itemsCustom == null) { // 没有查询到相应商品
        throw new CustomException("无此商品");
    }

    // 返回 ModelAndView
    ModelAndView mav = new ModelAndView();
    // 将商品信息填充到 model 中
    mav.addObject("itemsCustom", itemsCustom);
    // 设定商品修改页面
    mav.setViewName("items/editItems");

    return mav;
}

```

也可以在 service 方法中抛出：

```

public ItemsCustom findItemsById(Integer id) throws Exception {

    Items items = mapper.selectByPrimaryKey(id);
    ItemsCustom custom = null;
    if(items != null) {
        // 这里可能还要进行其他的业务处理
        custom = new ItemsCustom();
        BeanUtils.copyProperties(custom, items);
        return custom; // 最终返回的是扩展过的 Items
    } else {
        throw new CustomException("无此商品");
    }
}

```

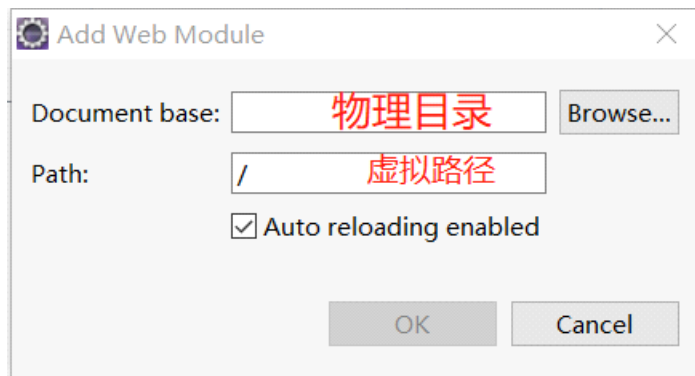
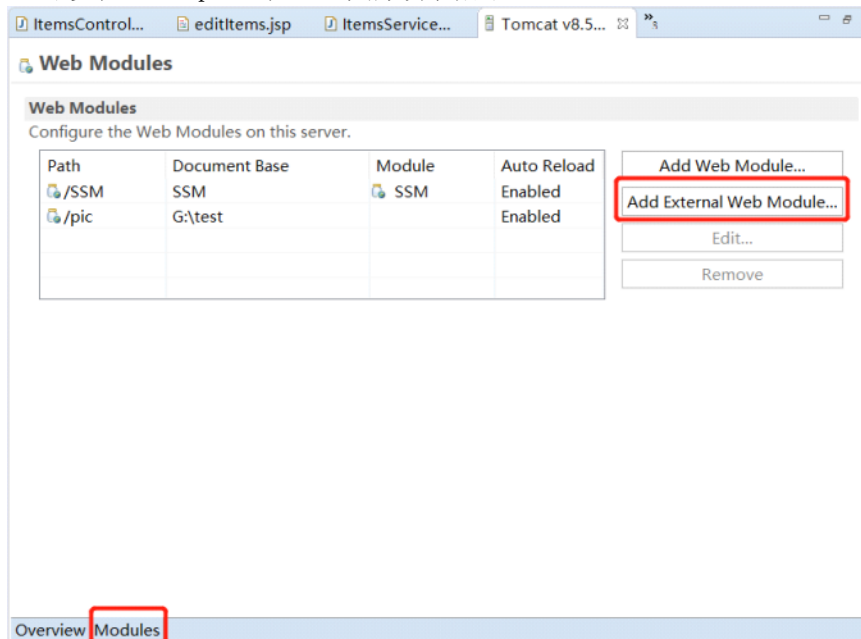
与业务功能相关的异常在 `service` 中抛出，否则在 `controller` 中抛出。因此，上面的异常应该在 `service` 中抛出。抛出异常的方法要有 `throws` 声明。

上传图片

2018年6月29日 17:35

一、配置保存图片的目录

1. 可以在 Eclipse 中通过图形界面配置



2. 也可以在 Tomcat 的 conf/server.xml 中配置

```
<Context docBase="G:\test" path="/pic" reloadable="false">
```

访问的时候就直接是 localhost:8080/pic/

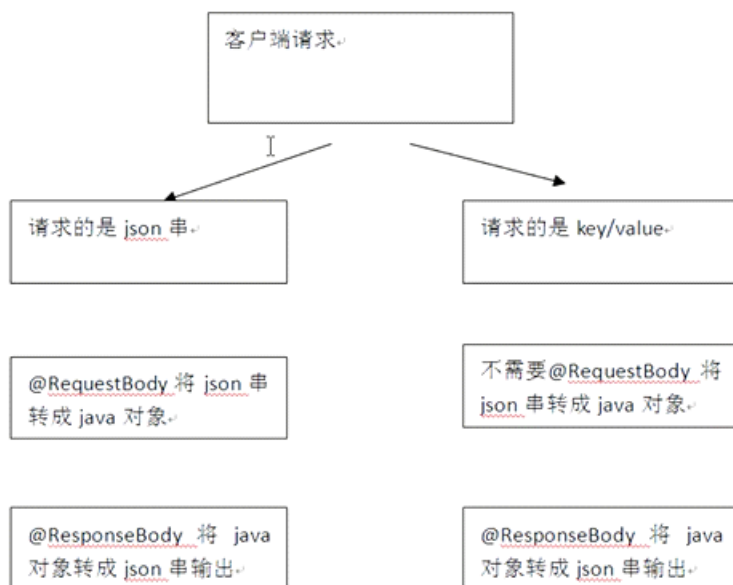
在实际开发中，通常是单独建立一个图片服务器，而且在设置虚拟目录时要分级，以提高 I/O 性能，这种分级通常是按照年/月/日的级别来分级。

二、

JSON 传输

2018年6月29日 18:08

一、概述



如果要求前端在发送请求时把数据写成 JSON，那么对前端而言会很不方便，所以 Spring MVC 常采用的模式是前端请求键值对，后端响应 JSON。

二、配置

1. 使用 jackson-mapper-asl 和 jackson-core-asl 进行 JSON 转换。
2. 在 springmvc.xml 中进行配置

```
<!-- 注解适配器 -->
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"
>
    <property name="messageConverters">
        <list>
            <bean
class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"></bean>
        </list>
    </property>
</bean>
```

如果已经写了 <mvc:annotation-driven/>，那么可以不用这样配置。

三、请求 JSON，响应 JSON

```
// 传过来的是 JSON，参数绑定到 itemsCustom 之后直接返回就可以了
@RequestMapping("/requestJson")
public @ResponseBody ItemsCustom requestJson(@RequestBody ItemsCustom itemsCustom) {

    return itemsCustom;
}
```

参数加 @RequestBody，返回值加 @ResponseBody

四、请求键值对，响应 JSON

```
// 传过来的是键值对
@RequestMapping("/requestKeyValue")
public @ResponseBody ItemsCustom requestKeyValue(ItemsCustom
itemsCustom) {

    return itemsCustom;
}
```

只需要在返回值加 @ResponseBody。

RESTful

2018年6月29日 20:22

一、概述

REST, 即Representational State Transfer的缩写, 表现层状态转化。如果一个架构符合 REST 原则, 就称它为 RESTful 架构。

- (1) 每一个URI代表一种资源;
- (2) 客户端和服务端之间, 传递这种资源的某种表现层;
- (3) 客户端通过四个 HTTP 动词 (GET 用来获取资源, POST 用来新建资源也可以用于更新资源, PUT 用来更新资源, DELETE 用来删除资源), 对服务器端资源进行操作, 实现“表现层状态转化”。

二、RESTful 架构的具体规范

1. URL 格式

URL 必须简洁, 在 URL 中不能出现表示客户端意愿的动词, 客户端想对资源进行的操作必须用 HTTP 的方法来表达。

- 非 REST 的 URL: /posts/show/1
- REST 的 URL: /posts/1, 然后用GET方法表示show

2. HTTP 方法

不管是获取、新建、更新、删除资源, 使用的 URL 都是一样的, 因此在 HTTP 请求报文中添加方法参数来区别。这就需要 controller 进行判断, controller 方法编写起来就比较复杂。

3. 请求报文中的 ContentType

请求报文必须明确指定这个字段。

可以看到, 如果完全遵守 REST 原则, 那么开发起来实际上会比较繁琐, 所以大部分情况都只实现了 REST 的 URL 风格。

三、实现 RESTful URL

在 web.xml 中配置一个 REST 前端控制器:

```
<!-- RESTful 前端控制器 -->
<servlet>
    <servlet-name>springmvc-rest</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 配置初始化参数 -->
    <init-param>
        <!-- 这是配置文件, 配置处理器映射器、适配器等 -->
        <!-- 如果不配置这个参数, 那么程序会默认加载 /WEB-INF/servletname-
            servlet.xml, 在这里也就是指 springmvc-servlet.xml -->
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring/springmvc.xml</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>springmvc-rest</servlet-name>
    <!-- 三种配置方法 -->
    <!-- 1 *.action 所有访问以 .action 结尾的资源都会由 DispatcherServlet
        解析 -->
    <!-- 2 / 所有访问都由 DispatcherServlet 解析, 但是注意对静态资源的访问不
        能由 DispatcherServlet
```

```
    解析, 这种方法能实现 RESTful 风格的 URL -->
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

编写 controller 方法:

```
// 查询商品信息, 响应 JSON
@RequestMapping("/itemsView/{id}")
public @ResponseBody ItemsCustom itemsView(@PathVariable("id") Integer
id) throws Exception{
    ItemsCustom itemsCustom = service.findItemsById(id);
    return itemsCustom;
}
```

@PathVariable 用来指定将 URL 中的参数绑定到 controller 方法的哪个参数中。

四、解析静态资源

但是配置了 RESTful 前端控制器之后, 对静态资源的请求也会交给前端控制器处理, 这是不允许的, 所以在 springmvc.xml 中进行配置:

```
<mvc:resource location="/js/" mapping="/js/**">
```

这就意味着 URL 为 /js/ 时, 就会访问到 /js 下的所有资源。

拦截器

2018年6月29日 21:13

一、概述

拦截器类要实现 `HandlerInterceptor` 接口，接口的三个方法作用如下：

- (1) `preHandle`：在进入 `Handler` 之前执行，常用于身份验证、授权等。返回 `true` 表示放行，否则表示拦截
- (2) `postHandle`：进入 `Handler` 之后、返回 `ModelAndView` 之前执行。这个方法的作用主要体现于 `ModelAndView`，公用的 `model` 数据可以从这里转发给视图，也可以在这里指定统一的视图
- (3) `afterCompletion`：完成 `Handler` 之后执行，在这里可以执行统一的异常处理、日志处理

二、配置

Spring MVC 可以将拦截器注入到每个处理器映射器，凡是 URL 映射成功的请求都要先经过拦截器。在 `sprimgvc.xml` 中进行配置：

```
<!-- 配置拦截器 -->
<mvc:interceptors>
    <!--如果配置多个拦截器，那么按配置顺序执行 -->
    <mvc:interceptor>
        <!-- 拦截所有的 URL 包括其子 URL -->
        <mvc:mapping path="/*" />
        <bean class="ssm.interceptor.MyHandlerInterceptor" />
    </mvc:interceptor>
    <mvc:interceptor>
        <!-- 拦截所有的 URL 包括其子 URL -->
        <mvc:mapping path="/*" />
        <bean class="ssm.interceptor.MyHandlerInterceptor2" />
    </mvc:interceptor>
</mvc:interceptors>
```

三、拦截器方法执行时机

1. 两个拦截器都放行

`preHandle`方法按顺序执行，`postHandle` 和 `afterCompletion`按拦截器配置的逆向顺序执行。

2. 拦截器1放行，拦截器2不放行

- 拦截器1放行，拦截器2 `preHandle` 才会执行。
- 拦截器2 `preHandle` 不放行，拦截器2 `postHandle`和 `afterCompletion` 不会执行。
- 只要有一个拦截器不放行，`postHandle` 不会执行

3. 两个拦截器都不放

- 拦截器1 `preHandle` 不放行，`postHandle` 和 `afterCompletion` 不会执行。
- 拦截器1 `preHandle` 不放行，拦截器2不执行。

4. 拦截器1不放行，拦截器2放行

相当于两个都不放行

四、应用

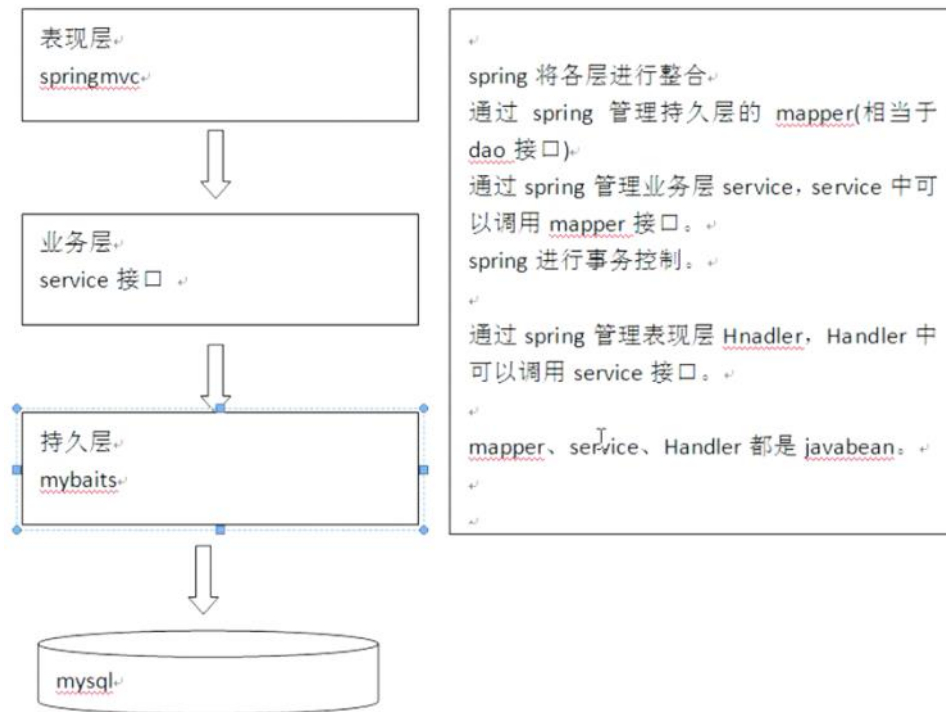
1. 统一日志处理拦截器的 `preHandle` 一定要放行，且将它放在拦截器链接中第一个位置。
2. 登陆认证拦截器，放在拦截器链接中第一个位置。权限校验拦截器，放在登陆认证拦截器之后。（因为登陆通过后才校验权限，当然登录认证拦截器要放在统一日志处理拦截器后面）

基于 Eclipse

2018年6月24日 15:20

一、架构

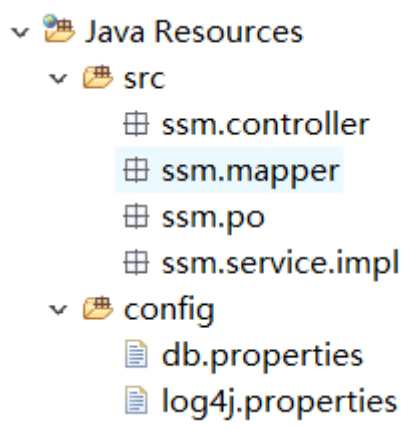
如果是自己负责全栈，那么一般是从后往前开发。



所需的 jar 包:

- mybatis 的 jar 包
- mybatis 和 spring 整合包
- log4j 包
- dbcp 数据库连接池包
- spring3.2 所有 jar 包
- jstl 包

所需的配置文件: 数据库连接属性、log4j 配置



二、整合持久层

1. 思路

MyBatis 和 Spring 整合, 通过 Spring 管理 `mapper` 接口。具体方法是使用 `mapper` 的扫描

器自动扫描 mapper 接口并在 Spring 中进行注册。

2. 步骤

(1) 配置 MyBatis, 创建配置文件 SqlMapConfig.xml

```
<!-- 全局参数 settings, 按需配置 -->

<!-- 定义别名 -->
<typeAliases>
    <!-- 批量扫描包别名 -->
    <package name="ssm.po" />
</typeAliases>
```

(2) 创建整合的配置文件, applicationContext-dao.xml

```
<!-- 数据库的配置文件 -->
<context:property-placeholder
    location="classpath:db.properties" />
<!-- 配置数据源 -->
<bean id="dbcp" class="org.apache.commons.dbcp2.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="maxTotal" value="30" />
    <property name="maxIdle" value="5" />
</bean>

<!-- SqlSessionFactory 对象 -->
<bean id="factory"
    class="org.mybatis.spring.SqlSessionFactoryBean">
    <!-- 连接池 -->
    <property name="dataSource" ref="dbcp"></property>
    <!-- 加载 MyBatis 的配置文件 -->
    <property name="configLocation"
        value="classpath:mybatis/SqlMapConfig.xml"></property>
</bean>

<!-- 配置 mapper -->
<!-- 批量扫描包中的所有 mapper 接口, 自动创建代理对象并在 Spring 容器中注册 -->
<!-- 指定需要扫描的包, 如果有多个包, 包与包之间用半角逗号隔开 -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <!-- 注意这里的属性名 -->
    <property name="basePackage" value="ssm.mapper"></property>
    <property name="sqlSessionFactoryBeanName" value="factory">
</property>
</bean>
```

(3) 用 MBG 生成 POJO 和 mapper 接口 (针对单表的)。

(4) 手动编写复杂查询 (比如多表) 的 mapper 接口。

```
<mapper namespace="ssm.mapper.ItemsMapperCustom">

    <!-- SQL 片段 -->
    <sql id="query_items_where">
        <!-- 动态拼接 -->
```

```

        <!-- 如果查询条件通过 ItemsCustom 对象传递 -->
        <if test="itemsCustom != null">
            <if test="itemsCustom.name!=null and itemsCustom.name!
            =''">
                items.name LIKE concat('%',{itemsCustom.name},'%')
            </if>
        </if>
    </sql>

    <!-- 如果是复杂的查询，传入的参数最好是包装过的 POJO -->
    <select id="queryItemsList" parameterType="ssm.po.ItemsQueryVo"
    resultType="ssm.po.ItemsCustom">
        SELECT items.id,items,name,items.price,items.create_time FROM
        items
        <where>
            <include refid="query_items_where"></include>
        </where>
    </select>

</mapper>

// 这个对象是提供给表现层的
public class ItemsQueryVo {

    // 原始商品信息
    private Items items;

    // 扩展后的商品信息
    private ItemsCustom itemsCustom;

    public Items getItems() {
        return items;
    }

    public void setItems(Items items) {
        this.items = items;
    }

    public ItemsCustom getItemsCustom() {
        return itemsCustom;
    }

    public void setItemsCustom(ItemsCustom itemsCustom) {
        this.itemsCustom = itemsCustom;
    }

}

// 这个对象用在后端，作为原始 pojo 的扩展
public class ItemsCustom extends ItemsWithBOLBs {

    // 新增的商品信息
}

```

三、整合业务层

1. 思路

通过 Spring 管理 service 接口。具体方法是将 service 接口配置在 Spring 的配置文件中（基于注解）。

2. 步骤

(1) 定义 service 接口

```
public interface ItemsService {

    public List<ItemsCustom> queryItemsList(ItemsQueryVo vo) throws
    Exception;

}

public class ItemsServiceImpl implements ItemsService {

    @Autowired
    private ItemsMapperCustom mapper; // 需要有一个 mapper 代理对象

    @Override
    public List<ItemsCustom> queryItemsList(ItemsQueryVo vo) throws
    Exception {
        return mapper.queryItemsList(vo);
    }

}
```

(2) 在 Spring 容器中配置 service。创建 applicationContext-service.xml 来配置。

```
<!-- 配置 service 对象 -->
<bean id="itemsService" class="ssm.service.impl.ItemsServiceImpl">
</bean>
```

(3) 新建一个配置文件 applicationContext-transaction.xml 进行事务管理的有关配置 (这里以配置方式为例)

```
<!-- 配置事务管理 -->
<!-- Spring 使用 JDBC 的事务控制类来对持久层框架进行事务管理 -->
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransact
    ionManager">
    <!-- 配置数据源 -->
    <!-- 这个数据源是在 appliacationContext-dao.xml 中配置的 -->
    <!-- 据说 Spring 在运行的时候把多个配置文件合为一个，所以可以跨配置文件
    引用 bean -->
    <property name="dataSource" ref="dbcp"></property>
</bean>

<!-- 配置事务的通知 -->
<tx:advice id="txAdvice"
    transaction-manager="transactionManager">
    <!-- 以下配置的是执行事务的方法 -->
    <tx:attributes>
        <!-- propagation 指定事务的传播行为 -->
        <!-- REQUIRED 是在调用 service 方法的时候开启一个事务，如果这个
        方法还调用了其他的 service 方法，那么如果当前方法产生了事务就用当
        前方法产生的事务，
        否则就创建一个新的事务 -->
        <!-- SUPPORTS : 如果当前没有创建事务，那么就一直以非事务的方式执
```

```

行 -->
<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="read" 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>

<!-- 配置事务的 AOP -->
<aop:config>
    <aop:advisor advice-ref="txAdvice"
        pointcut="execution(* ssm.service.impl.*.*(..))" />
</aop:config>

</beans>

```

四、整合表现层

1. 思路

Spring MVC 是 Spring 的一个模块，无需专门整合。

2. 步骤

(1) 创建 springmvc.xml 文件，配置处理器映射器、处理器适配器、视图解析器。

```

<!-- 注解扫描，扫描处理器的包 -->
<context:component-scan
    base-package="ssm.controller"></context:component-scan>

<!-- 开启注解驱动，配置处理器适配器和映射器 -->
<mvc:annotation-driven></mvc:annotation-driven>

<!-- 配置视图解析器 -->
<bean
    class="org.springframework.web.servlet.view.InternalResourceViewRe
solver">
    <!-- 前缀 -->
    <property name="prefix" value="/WEB-INF/jsp/"></property>
    <!-- 后缀 -->
    <property name="suffix" value=".jsp"></property>
</bean>

```

(2) 在 web.xml 中配置前端控制器。

```

<!-- 配置前端控制器 -->
<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <!-- 配置初始化参数 -->
    <init-param>
        <!-- 这是配置文件，配置处理器映射器、适配器等 -->
        <!-- 如果不配置这个参数，那么程序会默认加载 /WEB-
INF/servletname-servlet.xml，在这里也就是指 springmvc-

```

```

        servlet.xml -->
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring/springmvc.xml</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <!-- 三种配置方法 -->
    <!-- 1 *.action 所有访问以 .action 结尾的资源都会由
    DispatcherServlet 解析 -->
    <!-- 2 / 所有访问都由 DispatcherServlet 解析，但是注意对静态资源的访
    问不能由 DispatcherServlet
    解析，这种方法能实现 RESTful 风格的 URL -->
    <url-pattern>*.action</url-pattern>
</servlet-mapping>

```

(3) 基于注解开发处理器（实际开发中常称为 Controller）

```

// 商品的处理器
@Controller
public class ItemsController {

    @Autowired
    private ItemsService service;

    // 商品查询
    @RequestMapping("/queryItems")
    public ModelAndView queryItems() throws Exception {

        // 调用 service 查找数据库库，查询商品列表
        List<ItemsCustom> itemList = service.queryItemsList(null);

        ModelAndView mav = new ModelAndView(); // 返回 ModelAndView
        mav.addObject("itemsList", itemList); // 相当于
        request.setAttribute

        mav.setViewName("itemsList"); // 指定视图

        return mav;
    }
}

```

(4) 写页面

五、加载 Spring 容器

整合的过程中用 Spring 管理所有的相关组件，这些组件都配置在 applicationContext-*.xml 文件中。在 web.xml 中添加 Spring 容器的监听器：

```

<!-- 加载 Spring 容器 -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <!-- 这里使用了通配符，方便加载所有的相关配置文件 -->
    <param-value>/WEB-INF/classes/spring/applicationContext-*.xml</param-value>
</context-param>

```

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

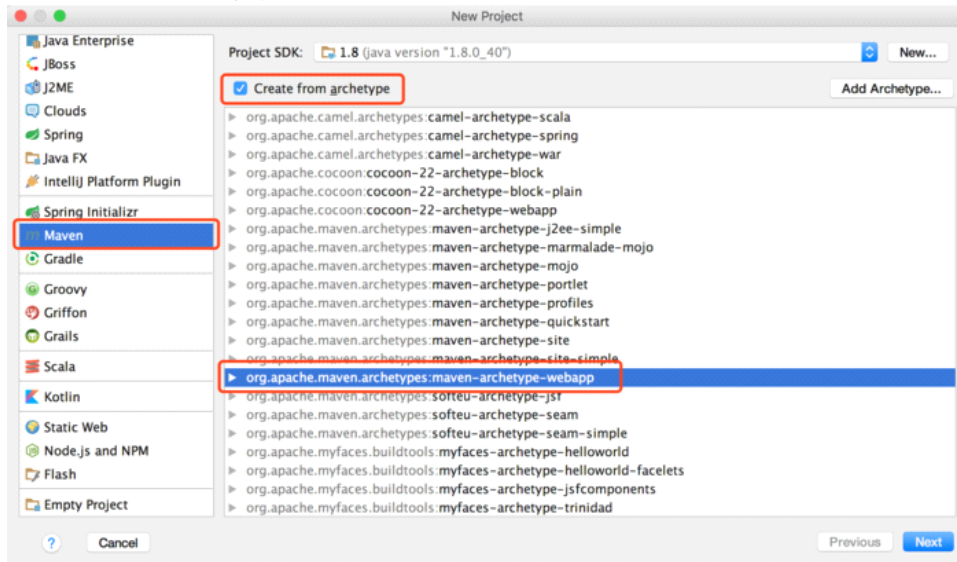
基于 IDEA 和 Maven

2018年6月30日 14:54

IDEA 版本: 2018.1

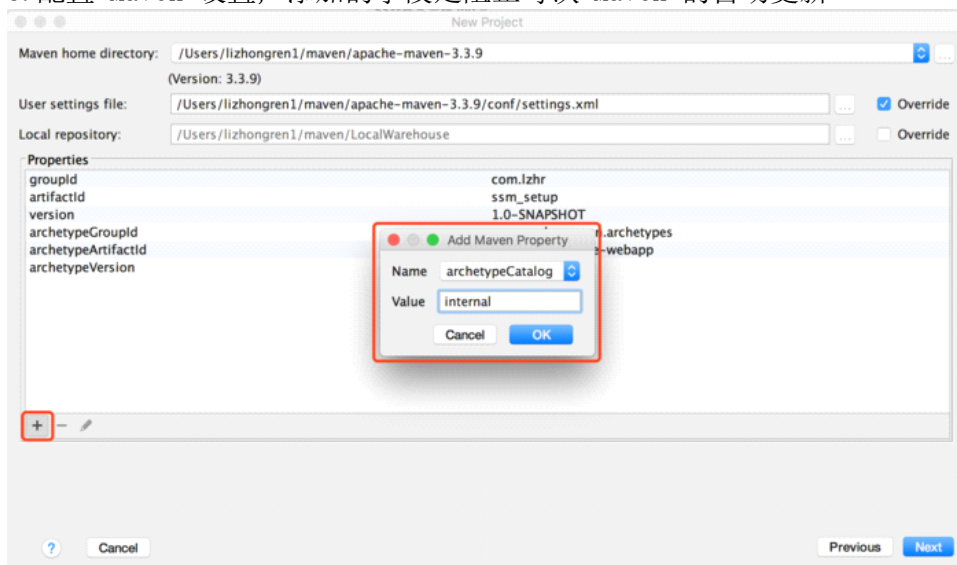
一、新建 Project

1. 选择 Maven 的模板进行创建



2. 填写 groupId 和 ArtifactId , 这里尽量按照 Maven 的命名规范来。一个 ArtifactId 就对应一个 Web 项目。

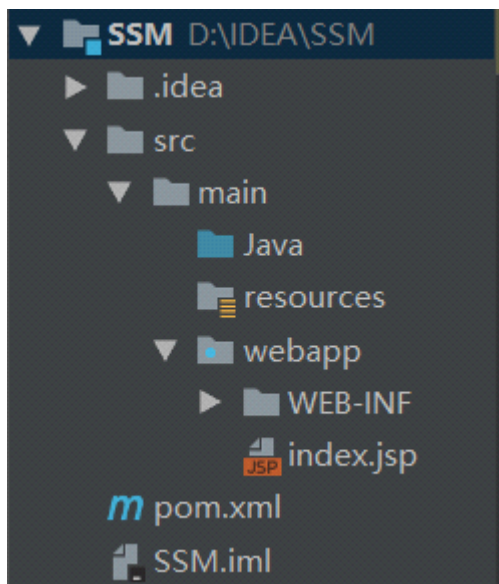
3. 配置 Maven 设置, 添加的字段是阻止每次 Maven 的自动更新



4. 起项目名

5. 完成, 等待 IDEA 建立索引

二、完善项目结构



最终的基本结构如图，Java 和 resources 目录可能要手动创建。创建完之后右键 Java 目录选择 Make Directory as/Sources Root，右键 resources 目录选择 Make Directory as/Resources Root
Java 目录放的是 .java 文件，resources 目录主要是配置文件。

三、配置 Maven

在 pom.xml 文件中添加需要的配置。大概是下面这些依赖，运行时发现缺了什么就再加：

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

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>edu.whu</groupId>
  <artifactId>SSM</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>

  <name>SSM Maven Webapp</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>
UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
    <!-- 设置 spring 的版本 -->
    <spring.version>5.0.4.RELEASE</spring.version>
    <!-- 配置 jackson 的版本 -->
    <jackson.version>2.9.4</jackson.version>
  </properties>

  <dependencies>

    <!-- servlet 的版本和 Tomcat 的版本要对应 -->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.1.0</version>
    </dependency>
  </dependencies>
</project>
```

```

        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>

    <!-- 配置 spring -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-tx</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>${spring.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>${spring.version}</version>
    </dependency>

    <!-- 使用 SpringMVC 需配置 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>

    <!-- aop -->
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.9.0</version>
    </dependency>

    <!-- Spring 和 MyBatis -->
    <dependency>
        <groupId>org.mybatis</groupId>

```

```

        <artifactId>mybatis-spring</artifactId>
        <version>1.3.2</version>
    </dependency>

    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.46</version>
    </dependency>

    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.4.6</version>
    </dependency>

    <dependency>
        <groupId>org.mybatis.generator</groupId>
        <artifactId>mybatis-generator-core</artifactId>
        <version>1.3.6</version>
    </dependency>

    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-dbcp2</artifactId>
        <version>2.1.1</version>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-annotations</artifactId>
        <version>${jackson.version}</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>${jackson.version}</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>${jackson.version}</version>
    </dependency>

    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>

    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>

    <dependency>
        <groupId>commons-beanutils</groupId>

```

```

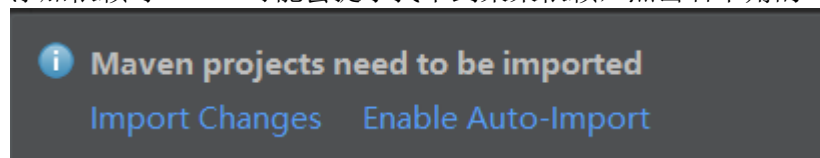
        <artifactId>commons-beanutils</artifactId>
        <version>1.9.3</version>
    </dependency>

</dependencies>

<build>
    <finalName>SSM</finalName>
    <pluginManagement><!-- lock down plugins versions to avoid
using Maven defaults (may be moved to parent pom) -->
        <plugins>
            <plugin>
                <artifactId>maven-clean-plugin</artifactId>
                <version>3.0.0</version>
            </plugin>
            <!-- see http://maven.apache.org/ref/current/maven-core/default-bindings.html#Plugin bindings for war packaging -->
            <plugin>
                <artifactId>maven-resources-plugin</artifactId>
                <version>3.0.2</version>
            </plugin>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.7.0</version>
            </plugin>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.20.1</version>
            </plugin>
            <plugin>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.2.0</version>
            </plugin>
            <plugin>
                <artifactId>maven-install-plugin</artifactId>
                <version>2.5.2</version>
            </plugin>
            <plugin>
                <artifactId>maven-deploy-plugin</artifactId>
                <version>2.8.2</version>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
</project>

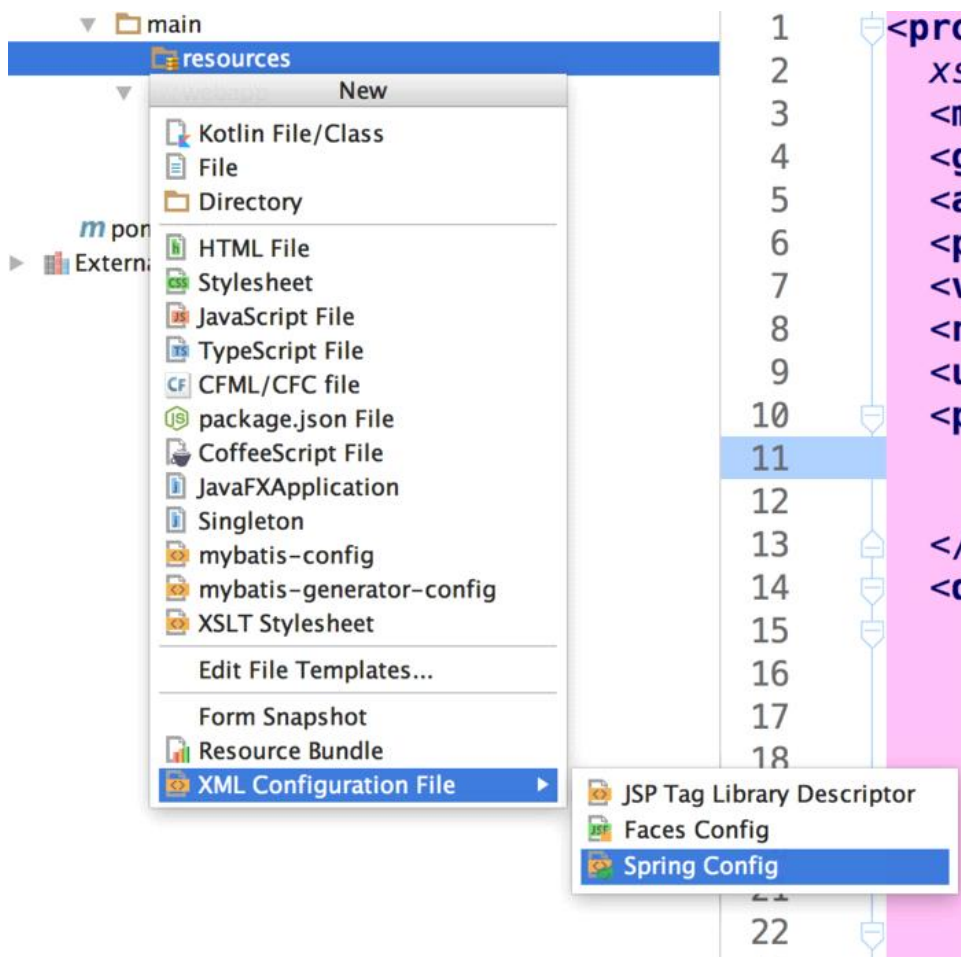
```

添加依赖时 IDEA 可能会提示找不到某某依赖，点击右下角的 Import Changes 即可。



四、配置 Spring MVC

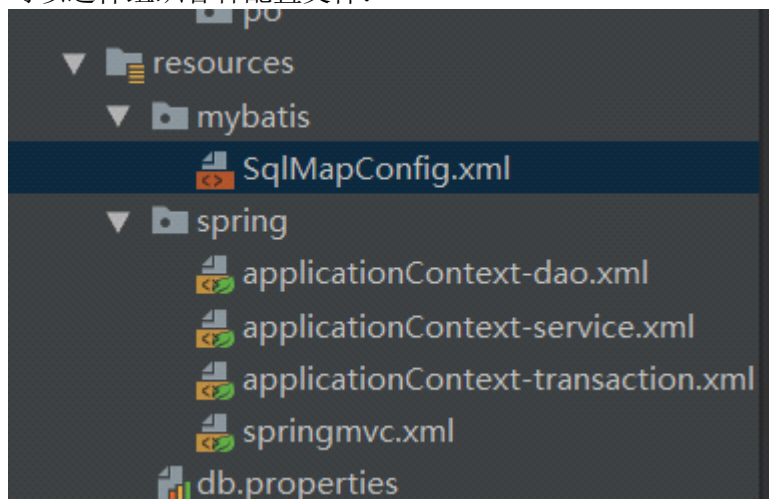
在 resources 文件夹中新建一个 XML 文件进行配置：



四、配置 MyBatis

1. 在 resources 目录下创建数据库连接池的配置文件 .properties
2. 在 resources 目录下创建 log4j 的配置文件 .properties
3. 在 resources 目录下创建 MyBatis 的配置文件 .xml

可以这样组织各种配置文件：



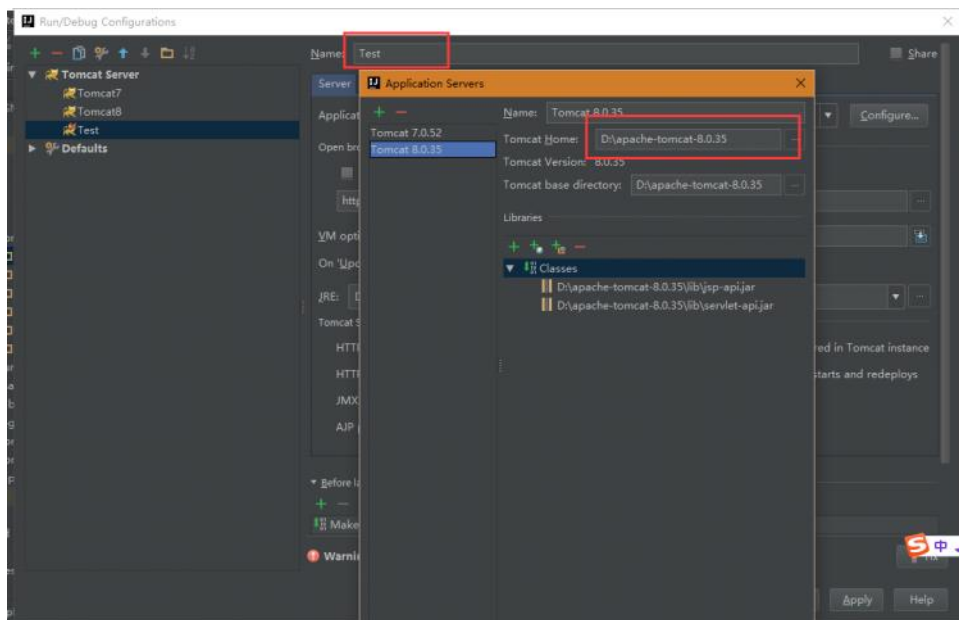
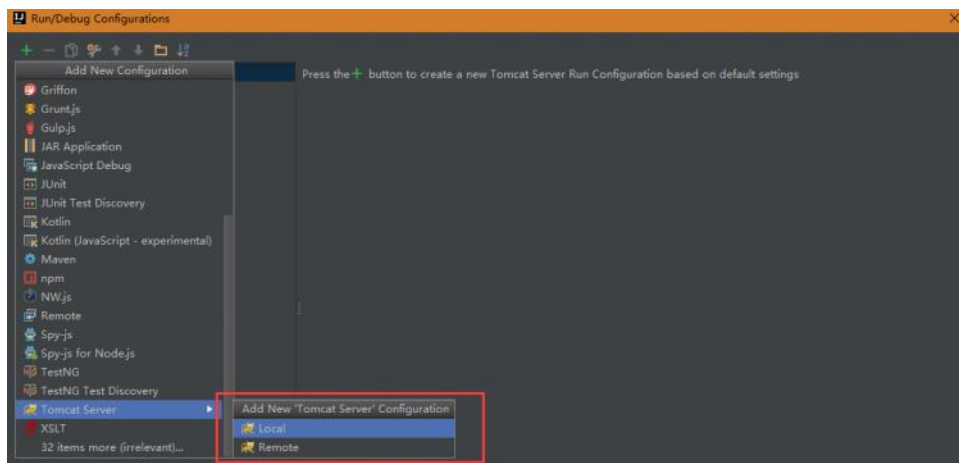
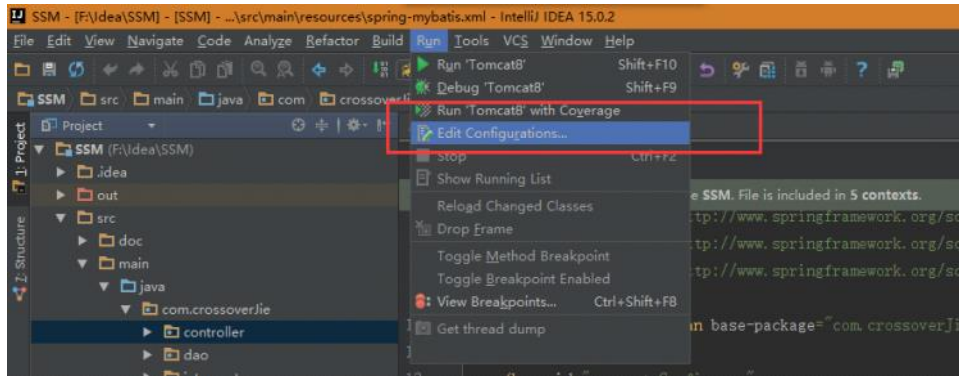
五、配置 web.xml

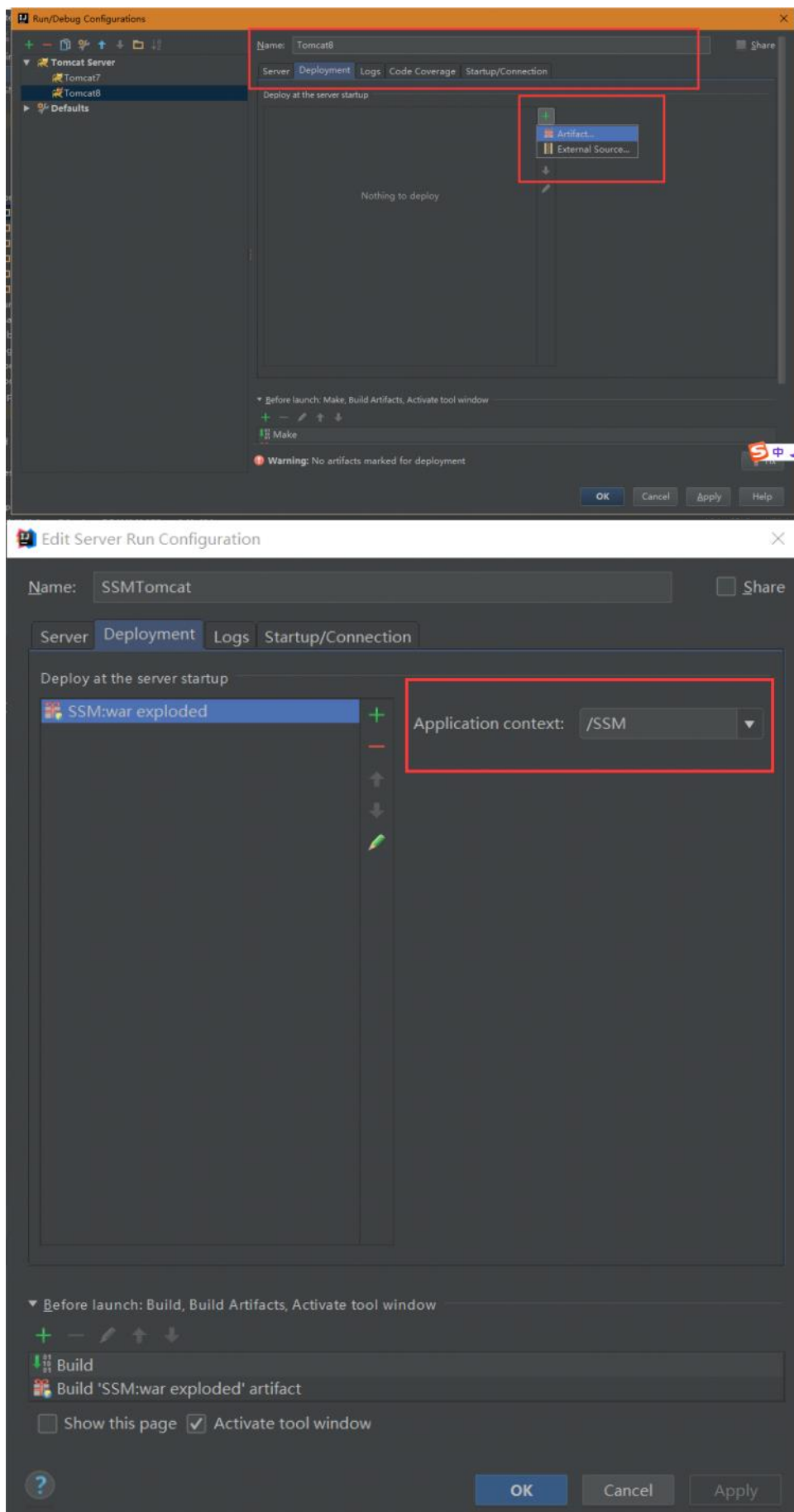
有些地方的写法和在 Eclipse 中的不同：

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <!-- 这里使用了通配符，方便加载所有的相关配置文件 -->
  <param-value>classpath:/spring/applicationContext-*.xml</param-
value>
</context-param>
<listener>
  <listener-class>
```

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

六、配置 Tomcat

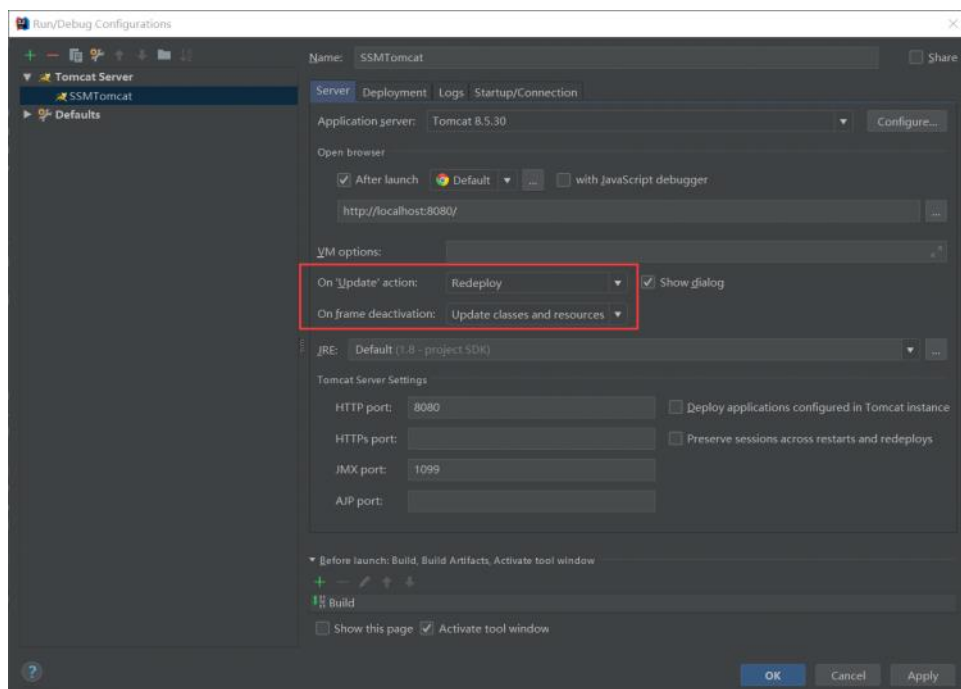




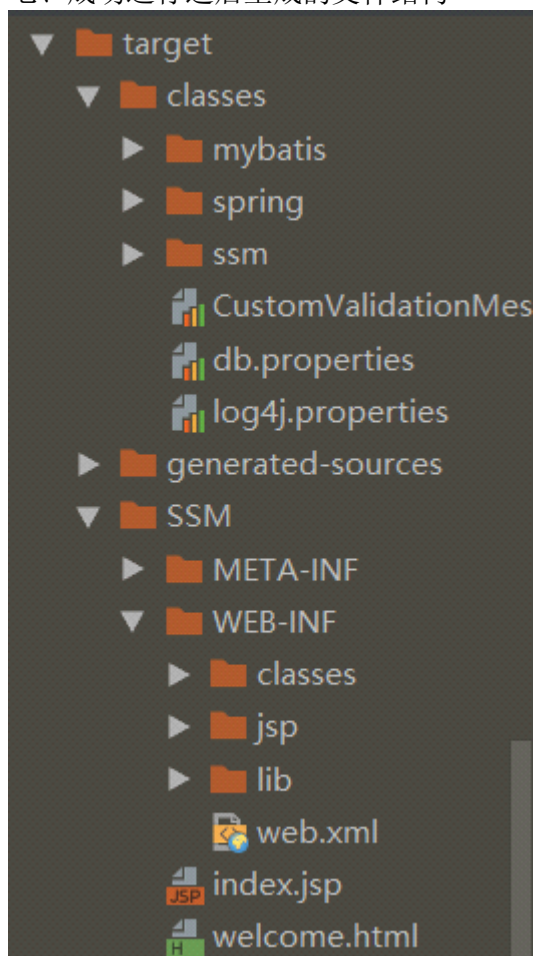
上面这张图红框的地方配的是这个 Web 应用的名称，IDEA 默认把它配成了 /，这意味着如果在浏览器中输入 localhost:8080，当应用在线上时就直接访问到这个应用；如果应用不在线上那么就访问不到任何东西，Tomcat 的主页也不能，因为这时 URL 冲突了。所以这个地方一定要记得配置。配置完成之后访问这个应用下的资源时 URL 就是 localhost:8080/SSM/

注意一个地方，在 Artifact 那一步选择的是 war 这种配置时，回到下面这个页面时可以配置两个选项：

- On update action: 当发现更新时的操作，可以选重新部署
- On frame deactivation: 当IDEA 切换时的操作（比如打开网页等），选择Update classes and resources，这样在修改页面时不用重启服务器，实现热部署

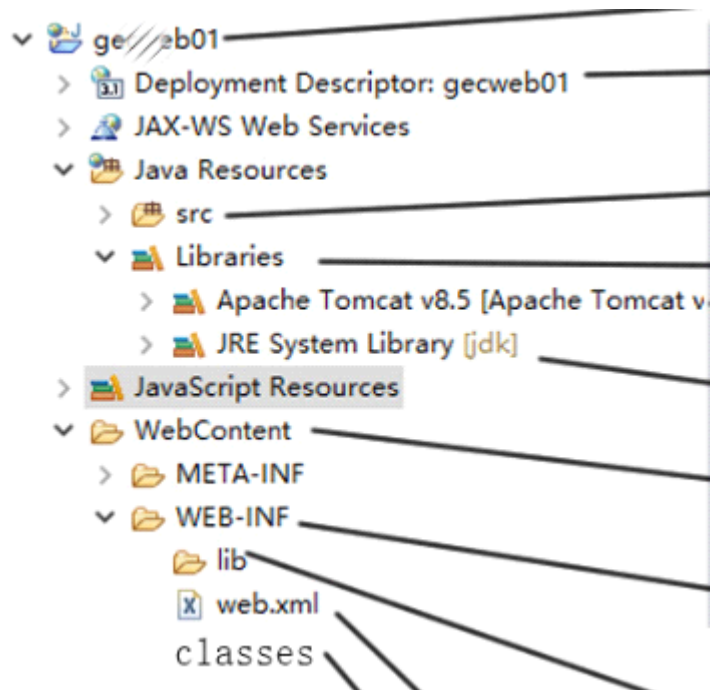


七、成功运行之后生成的文件结构



成功部署之后，Project 的根目录下生成了一个 target 文件夹。可以看到下面的 SSM 文件

夹其实与 Eclipse 中 WebContent 文件夹结构一致。



IDEA 的部署路径

2018年6月30日 19:30

一、war exploded 模式

项目会部署到 C:\Users\DELL\.IntelliJ IDEA2018.1\system\tomcat 下的对应目录中，目录的名称是由项目名也就是 Maven 中的 ArtifactId 转化来的。

目录下有三个文件夹：

- conf
- logs
- work

conf 就是一些配置，logs 是日志，work 中是页面等资源。/conf/Catalina/localhost 下有一个 XML 文件，文件的名称是之前配置的 Application Context，也就是所谓的 Web 应用的名称，里面有配置：

```
<Context path="/SSM" docBase="D:\IDEA\SSM\target\SSM" />
```

这其实就是配置了 Web 应用的根 URL。

二、war 模式

当好好配置 Application Context 之后（也就是说这里不能用默认的 /），IDEA 会将项目部署到 Tomcat 的 webapps 目录下。文件夹的名称就是 Application Context。