

SSM

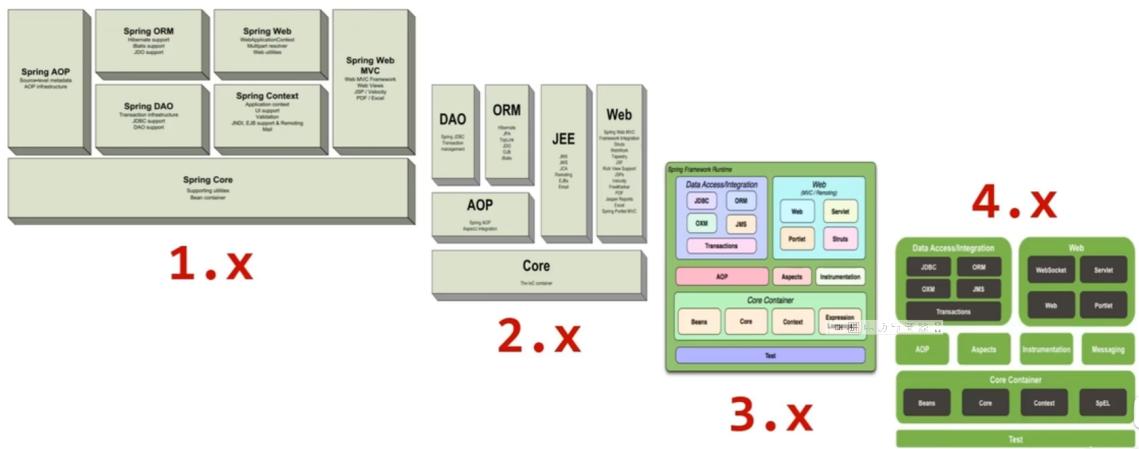
一、Spring

1、系统架构

Spring Framework系统架构

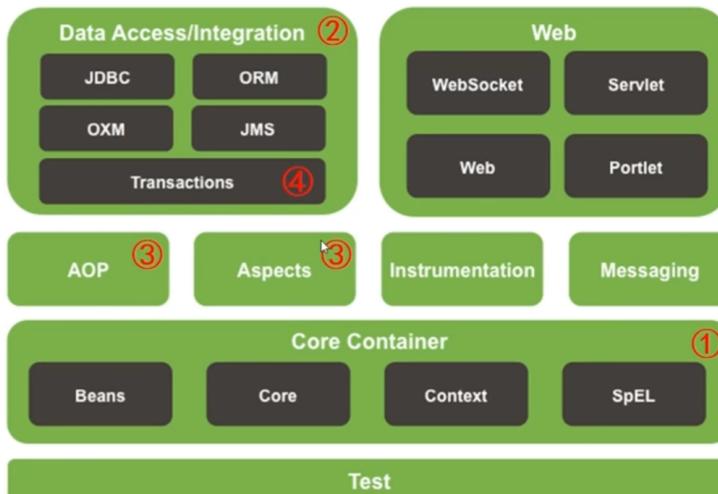
黑马程序员
www.itheima.com

- Spring Framework是Spring生态圈中最基础的项目，是其他项目的根基



Spring Framework系统架构

黑马程序员
www.itheima.com



- Data Access: 数据访问
- Data Integration: 数据集成
- Web: Web开发
- AOP: 面向切面编程
- Aspects: AOP思想实现
- Core Container: 核心容器
- Test: 单元测试与集成测试

2、核心概念

关注 黑马程序员 www.itheima.com

核心概念

- 代码书写现状
 - 耦合度偏高
- 解决方案
 - 使用对象时，在程序中不要主动使用new产生对象，转换为由外部提供对象
- IoC (Inversion of Control) 控制反转**
 - 对象的创建控制权由程序转移到外部，这种思想称为控制反转

关闭弹幕

业务层实现

```
public class BookServiceImpl implements BookService {  
    private BookDao bookDao;  
  
    public void save() {  
        bookDao.save();  
    }  
}
```

数据层实现

```
public class BookDaoImpl implements BookDao {  
    public void save() {  
        System.out.println("book dao save ...");  
    }  
}  
  
public class BookDaoImpl2 implements BookDao {  
    public void save() {  
        System.out.println("book dao save ...2");  
    }  
}
```

核心概念

黑马程序员 www.itheima.com

- IoC (Inversion of Control) 控制反转**
 - 使用对象时，由主动new产生对象转换为由外部提供对象，此过程中对象创建控制权由程序转移到外部，此思想称为控制反转
- Spring技术对IoC思想进行了实现
 - Spring提供了一个容器，称为**IoC容器**，用来充当IoC思想中的外部
 - IoC容器负责对象的创建、初始化等一系列工作，被创建或被管理的对象在IoC容器中统称为**Bean**
- DI (Dependency Injection) 依赖注入**
 - 在容器中建立bean与bean之间的依赖关系的整个过程，称为依赖注入

业务层实现

```
public class BookServiceImpl implements BookService {  
    private BookDao bookDao;  
  
    public void save() {  
        bookDao.save();  
    }  
}
```

数据层实现

```
public class BookDaoImpl implements BookDao {  
    public void save() {  
        System.out.println("book dao save ...");  
    }  
}
```

依赖dao对象运行

IoC容器

```
graph LR; service[service] -- 依赖 --> dao[dao]
```

核心概念

- 目标：充分解耦
 - 使用IoC容器管理bean（IoC）
 - 在IoC容器内将有依赖关系的bean进行关系绑定（DI）
- 最终效果
 - 使用对象时不仅可以直接从IoC容器中获取，并且获取到的bean已经绑定了所有的依赖关系

3、IoC入门案例

IoC入门案例思路分析

1. 管理什么？（Service与Dao）
2. 如何将被管理的对象告知IoC容器？（配置）
3. 被管理的对象交给IoC容器，如何获取到IoC容器？（接口）
4. IoC容器得到后，如何从容器中获取bean？（接口方法）
5. 使用Spring导入哪些坐标？（pom.xml）

IoC入门案例



步骤 IoC入门案例（XML版）

②：定义Spring管理的类（接口）

```
public interface BookService {  
    public void save();  
}
```

```
public class BookServiceImpl implements BookService {  
  
    private BookDao bookDao = new BookDaoImpl();  
  
    public void save() {  
        bookDao.save();  
    }  
}
```

步骤 Ioc入门案例 (XML版)

①：导入Spring坐标

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.10.RELEASE</version>
</dependency>
```

IoC入门案例

步骤 Ioc入门案例 (XML版)

②：定义Spring管理的类（接口）

```
public interface BookService {
    public void save();
}
```

```
public class BookServiceImpl implements BookService {
    private BookDao bookDao = new BookDaoImpl();

    public void save() {
        bookDao.save();
    }
}
```

IoC入门案例

步骤 Ioc入门案例 (XML版)

③：创建Spring配置文件，配置对应类作为Spring管理的bean

```
<?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
                           https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="bookService" class="com.itheima.service.impl.BookServiceImpl"></bean>
</beans>
```

注意事项

bean定义时id属性在同一个上下文中不能重复

步骤 IoC入门案例 (XML版)

④：初始化IoC容器（Spring核心容器/Spring容器），通过容器获取bean

```
public class App {  
    public static void main(String[] args) {  
        //加载配置文件得到上下文对象，也就是容器对象  
        ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");  
        //获取资源  
        BookService bookService = (BookService) ctx.getBean("bookService");  
        bookService.save();  
    }  
}
```

4. DI入门案例

DI入门案例思路分析

1. 基于IoC管理bean
2. Service中使用new形式创建的Dao对象是否保留？(否)
3. Service中需要的Dao对象如何进入到Service中？(提供方法)
4. Service与Dao间的关系如何描述？(配置)

①：删除使用new的形式创建对象的代码

```
public class BookServiceImpl implements BookService {  
    private BookDao bookDao = new BookDaoImpl();  
  
    public void save() {  
        bookDao.save();  
    }  
}
```

```

public class BookServiceImpl implements BookService {
    //5.删除业务层中使用new的方式创建的dao对象
    private BookDao bookDao;

    public void save() {
        System.out.println("book service save ...");
        bookDao.save();
    }
    //6.提供对应的set方法
    public void setBookDao(BookDao bookDao) {
        this.bookDao = bookDao;
    }
}

```

<!--1.导入spring的坐标spring-context，对应版本是5.2.10.RELEASE-->

```

<!--2.配置bean-->
<!--bean标签表示配置bean
id属性表示给bean起名字
class属性表示给bean定义类型-->
<bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl"/>

<bean id="bookService" class="com.itheima.service.impl.BookServiceImpl">
    <!--7.配置server与dao的关系-->
    <!--property标签表示配置当前bean的属性
name属性表示配置哪一个具体的属性
ref属性表示参照哪一个bean-->
    <property name="bookDao" ref="bookDao"/>
</bean>

```

步骤 DI入门案例（XML版）

③：配置service与dao之间的关系

```

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

    <bean id="bookService" class="com.itheima.di.
        <property name='bookDao' ref='bookDao'/>
    </bean>

    <bean id="bookDao" class="com.itheima.di.dao.impl.BookDaoImpl"/>
</beans>

```

```

public class BookServiceImpl implements BookService {

    private BookDao bookDao;

    public void save() {
        bookDao.save();
    }

    public void setBookDao(BookDao bookDao) {
        this.bookDao = bookDao;
    }
}

```

5、bean基础配置

类别	描述
名称	bean
类型	标签
所属	beans标签
功能	定义Spring核心容器管理的对象
格式	<pre><beans> <bean/> <bean></bean> </beans></pre>
属性列表	<code>id</code> : bean的id，使用容器可以通过id值获取对应的bean，在一个容器中id值唯一 <code>class</code> : bean的类型，即配置的bean的全路径类名
范例	<pre><bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl"/> <bean id="bookService" class="com.itheima.service.impl.BookServiceImpl"></bean></pre>

类别	描述
名称	name
类型	属性
所属	bean标签
功能	定义bean的别名，可定义多个，使用逗号(,)分号(;)空格()分隔
范例	<pre><bean id="bookDao" name="dao bookDaoImpl" class="com.itheima.dao.impl.BookDaoImpl"/> <bean name="service,bookServiceImpl" class="com.itheima.service.impl.BookServiceImpl"/></pre>

注意事项

获取bean无论是通过id还是name获取，如果无法获取到，将抛出异常NoSuchBeanDefinitionException
NoSuchBeanDefinitionException: No bean named 'bookServiceImpl' available

类别	描述
名称	scope
类型	属性
所属	bean标签
功能	定义bean的作用范围，可选范围如下 ● singleton: 单例 (默认) ● prototype: 非单例
范例	<pre><bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl" scope="prototype" /></pre>

bean作用范围说明

- 为什么bean默认为单例？
- 适合交给容器进行管理的bean
 - 表现层对象
 - 业务层对象
 - 数据层对象
 - 工具对象
- 不适合交给容器进行管理的bean
 - 封装实体的域对象

```
<!--name:为bean指定别名，别名可以有多个，使用逗号，分号，空格进行分隔-->
<bean id="bookService" name="service service4 bookEbi"
class="com.itheima.service.impl.BookServiceImpl">
    <property name="bookDao" ref="bookDao"/>
</bean>

<!--scope: 为bean设置作用范围，可选值为单例singleton，非单例prototype-->
<bean id="bookDao" name="dao" class="com.itheima.dao.impl.BookDaoImpl"
scope="prototype"/>
</beans>
```

6、bean的实例化

bean本质上就是对象，创建bean使用构造方法完成

构造方法

- 提供可访问的构造方法

```
public class BookDaoImpl implements BookDao {  
    public BookDaoImpl() {  
        System.out.println("book constructor is running ...");  
    }  
    public void save() {  
        System.out.println("book dao save ...");  
    }  
}
```

- 配置

```
<bean  
    id="bookDao"  
    class="com.itheima.dao.impl.BookDaoImpl"  
/>
```

- 无参构造方法如果不存在，将抛出异常BeanCreationException

静态工厂

- 静态工厂

```
public class OrderDaoFactory {  
    public static OrderDao getOrderDao(){  
        return new OrderDaoImpl();  
    }  
}
```

- 配置

```
<bean  
    id="orderDao"  
    factory-method="getOrderDao"  
    class="com.itheima.factory.OrderDaoFactory"  
/>
```

实例工厂与FactoryBean

- 实例工厂

```
public class UserDaoFactory {  
    public UserDao getUserDao(){  
        return new UserDaoImpl();  
    }  
}
```

- 配置

```
<bean id="userDaoFactory" class="com.itheima.factory.UserDaoFactory"/>  
  
<bean id="userDao" factory-method="getUserDao" factory-bean="userDaoFactory"  
      />
```

配合使用
实际无意义
方法名不固定
每次需要配置

实例化bean的第四种方式——FactoryBean

- FactoryBean

```
public class UserDaoFactoryBean implements FactoryBean<UserDao> {  
    public UserDao getObject() throws Exception {  
        return new UserDaoImpl();  
    }  
    public Class<?> getObjectType() {  
        return UserDao.class;  
    }  
}
```

- 配置

```
<bean id="userDao" class="com.itheima.factory.UserDaoFactoryBean" />
```

补充：

实例化bean的三种方式

- 构造方法 (常用)
- 静态工厂 (了解)
- 实例工厂 (了解)
 - FactoryBean (实用)

7、bean的生命周期

+ 关注 **n** 生命周期控制



黑马君
www.itheima.com

- 提供生命周期控制方法

```
public class BookDaoImpl implements BookDao {  
    public void save() {  
        System.out.println("book dao save ...");  
    }  
    public void init(){  
        System.out.println("book init ...");  
    }  
    public void destroy(){  
        System.out.println("book destroy ...");  
    }  
}
```

- 配置生命周期控制方法

```
<bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl" init-method="init" destroy-method="destroy"/>
```

init:开始bean, destroy:销毁bean

- 实现 InitializingBean, DisposableBean 接口

```
public class BookServiceImpl implements BookService, InitializingBean, DisposableBean {  
    public void save() {  
        System.out.println("book service save ...");  
    }  
    public void afterPropertiesSet() throws Exception {  
        System.out.println("afterPropertiesSet");  
    }  
    public void destroy() throws Exception {  
        System.out.println("destroy");  
    }  
}
```

bean 生命周期

- 初始化容器
 - 创建对象（内存分配）
 - 执行构造方法
 - 执行属性注入（set 操作）
 - 执行 bean 初始化方法
- 使用 bean
 - 执行业务操作
- 关闭/销毁容器
 - 执行 bean 销毁方法

set在init的前面

8、setter注入

依赖注入方式



- 思考：向一个类中传递数据的方式有几种？
 - 普通方法（set方法）
 - 构造方法
- 思考：依赖注入描述了在容器中建立bean与bean之间依赖关系的过程，如果bean运行需要的是数字或字符串呢？
 - 引用类型
 - 简单类型（基本数据类型与String）
- 依赖注入方式
 - setter注入
 - ◆ 简单类型
 - ◆ 引用类型
 - 构造器注入
 - 简单类型
 - 引用类型

引用类型：一个方法

简单类型：基本数据类型与String

```
public class BookDaoImpl implements BookDao {  
  
    private String databaseName;  
    private int connectionNum;  
    //setter注入需要提供要注入对象的set方法  
    public void setConnectionNum(int connectionNum) {  
        this.connectionNum = connectionNum;  
    }  
    //setter注入需要提供要注入对象的set方法  
    public void setDatabaseName(String databaseName) {  
        this.databaseName = databaseName;  
    }  
  
    public void save() {  
        System.out.println("book dao save ..."+databaseName+","+connectionNum);  
    }  
}
```

上面databaseName、connectionNum的值由xml文件赋

```

<!--注入简单类型-->
<bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl">
    <!--property标签：设置注入属性-->
    <!--name属性：设置注入的属性名，实际是set方法对应的名称-->
    <!--value属性：设置注入简单类型数据值-->
    <property name="connectionNum" value="10"/>
    <property name="databaseName" value="mysql"/>
</bean>

<bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"/>

<!--注入引用类型-->
<bean id="bookService" class="com.itheima.service.impl.BooksServiceImpl">
    <!--property标签：设置注入属性-->
    <!--name属性：设置注入的属性名，实际是set方法对应的名称-->
    <!--ref属性：设置注入引用类型bean的id或name-->
    <property name="bookDao" ref="bookDao"/>
    <property name="userDao" ref="userDao"/>
</bean>

```

9、构造器注入

上代码

```

public class BookDaoImpl implements BookDao {
    private String databaseName;
    private int connectionNum;

    public BookDaoImpl(String databaseName, int connectionNum) {
        this.databaseName = databaseName;
        this.connectionNum = connectionNum;
    }

    public void save() {
        System.out.println("book dao save ..." + databaseName + "," + connectionNum);
    }
}

```

就是setter该为构造方法

```

<bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl">
    根据构造方法参数名称注入
    <constructor-arg name="connectionNum" value="10"/>
    <constructor-arg name="databaseName" value="mysql"/>
</bean>
<bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"/>

<bean id="bookService" class="com.itheima.service.impl.BookServiceImpl">
    <constructor-arg name="userDao" ref="userDao"/>
    <constructor-arg name="bookDao" ref="bookDao"/>
</bean>

```

name指的是传入构造方法的形参

```

<!--解决形参名称的问题，与形参名不耦合--&gt;
&lt;bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl"&gt;
    &lt;!--根据构造方法参数类型注入--&gt;
    &lt;constructor-arg type="int" value="10"/&gt;
    &lt;constructor-arg type="java.lang.String" value="mysql"/&gt;
&lt;/bean&gt;
&lt;bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"/&gt;

&lt;bean id="bookService" class="com.itheima.service.impl.BookServiceImpl"&gt;
    &lt;constructor-arg name="userDao" ref="userDao"/&gt;
    &lt;constructor-arg name="bookDao" ref="bookDao"/&gt;
&lt;/bean&gt;
</pre>

```

依赖注入方式选择

1. 强制依赖使用构造器进行，使用setter注入有概率不进行注入导致null对象出现
2. 可选依赖使用setter注入进行，灵活性强
3. Spring框架倡导使用构造器，第三方框架内部大多数采用构造器注入的形式进行数据初始化，相对严谨
4. 如果有必要可以两者同时使用，使用构造器注入完成强制依赖的注入，使用setter注入完成可选依赖的注入
5. 实际开发过程中还要根据实际情况分析，如果受控对象没有提供setter方法就必须使用构造器注入
6. 自己开发的模块推荐使用setter注入

10、自动装配

依赖自动装配

- IoC容器根据bean所依赖的资源在容器中自动查找并注入到bean中的过程称为自动装配
- 自动装配方式
 - 按类型(常用)
 - 按名称
 - 按构造方法
 - 不启用自动装配

主要是按名称和按类型

```
public class BookServiceImpl implements BookService{  
    private BookDao bookDao;  
  
    public void setBookDao(BookDao bookDao) {  
        this.bookDao = bookDao;  
    }  
  
    public void save() {  
        System.out.println("book service save ...");  
        bookDao.save();  
    }  
}
```

set必须要

```
<bean class="com.itheima.dao.impl.BookDaoImpl"/>  
    <!--autowire属性：开启自动装配，通常使用按类型装配-->  
    <bean id="bookService" class="com.itheima.service.impl.BookServiceImp"  
    autowire="byType"/>
```

名称看bookService中声明的名称

```
<?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">
    <bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl"/>
    <bean id="bookDao2" class="com.itheima.dao.impl.BookDaoImpl"/>

    <bean id="bookService" class="com.itheima.service.impl.BookServiceImpl" autowire="byName"/>

</beans>
```

依赖自动装配特征

- 自动装配用于引用类型依赖注入，不能对简单类型进行操作
- 使用按类型装配时（`byType`）必须保障容器中相同类型的bean唯一，推荐使用
- 使用按名称装配时（`byName`）必须保障容器中具有指定名称的bean，因变量名与配置耦合，不推荐使用
- 自动装配优先级低于setter注入与构造器注入，同时出现时自动装配配置失效

11、集合注入

```
public class BookDaoImpl implements BookDao {

    private int[] array;

    private List<String> list;

    private Set<String> set;

    private Map<String, String> map;

    private Properties properties;

    public void setArray(int[] array) {
        this.array = array;
    }

    public void setList(List<String> list) {
        this.list = list;
    }

    public void setSet(Set<String> set) {
        this.set = set;
    }
```

```
public void setMap(Map<String, String> map) {
    this.map = map;
}

public void setProperties(Properties properties) {
    this.properties = properties;
}

public void save() {
    System.out.println("book dao save ...");

    System.out.println("遍历数组：" + Arrays.toString(array));

    System.out.println("遍历List" + list);

    System.out.println("遍历Set" + set);

    System.out.println("遍历Map" + map);

    System.out.println("遍历Properties" + properties);
}
```

```
<bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl">
    <!--数组注入-->
    <property name="array">
        <array>
            <value>100</value>
            <value>200</value>
            <value>300</value>
        </array>
    </property>
    <!--list集合注入-->
    <property name="list">
        <list>
            <value>itcast</value>
            <value>itheima</value>
            <value>boxuegu</value>
            <value>chuanzhihui</value>
        </list>
    </property>
    <!--set集合注入-->
    <property name="set">
        <set>
            <value>itcast</value>
            <value>itheima</value>
            <value>boxuegu</value>
            <value>boxuegu</value>
        </set>
    </property>
    <!--map集合注入-->
```

```

<property name="map">
    <map>
        <entry key="country" value="china"/>
        <entry key="province" value="henan"/>
        <entry key="city" value="kaifeng"/>
    </map>
</property>
<!--Properties注入-->
<property name="properties">
    <props>
        <prop key="country">china</prop>
        <prop key="province">henan</prop>
        <prop key="city">kaifeng</prop>
    </props>
</property>
</bean>

```

name指的是名称

set重复了会自动过滤

集合中也可以写<ref bean=""beanId/> (但不常见)

12、数据源对象管理

1、先在pom配数据源

```

<dependency>
    <groupId>c3p0</groupId>
    <artifactId>c3p0</artifactId>
    <version>0.9.1.2</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
</dependency>

```

如果是数据库也要配mysql

2、再注入

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/spring_db"/>
    <property name="user" value="root"/>
    <property name="password" value="root"/>
    <property name="maxPoolSize" value="1000"/>
</bean>
```

3、输出

```
public class App {
    public static void main(String[] args) {
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
        DataSource dataSource = (DataSource) ctx.getBean("dataSource");
        System.out.println(dataSource);
    }
}
```

13、加载properties文件

1、先加context依赖

```
<?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
       ">
```

```
<!-- 1.开启context命名空间-->
<!-- 2.使用context空间加载properties文件-->
<!--   <context:property-placeholder location="jdbc.properties" system-
properties-mode="NEVER"/>-->
<!--   <context:property-placeholder location="jdbc.properties,jdbc2.properties"
system-properties-mode="NEVER"/>-->
<!--   classpath:*.properties :   设置加载当前工程类路径中的所有properties文件-->
<!--   system-properties-mode属性: 是否加载系统属性-->
<!--   <context:property-placeholder location="*.properties" system-
properties-mode="NEVER"/>-->
```

```

<!--classpath*:*.properties : 设置加载当前工程类路径和当前工程所依赖的所有jar包中的所有properties文件-->
<context:property-placeholder location="classpath*:*.properties" system-properties-mode="NEVER"/>

<!-- 3. 使用属性占位符${}读取properties文件中的属性-->
<!-- 说明：idea自动识别${}加载的属性值，需要手工点击才可以查阅原始书写格式-->
<bean class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="${jdbc.driver}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

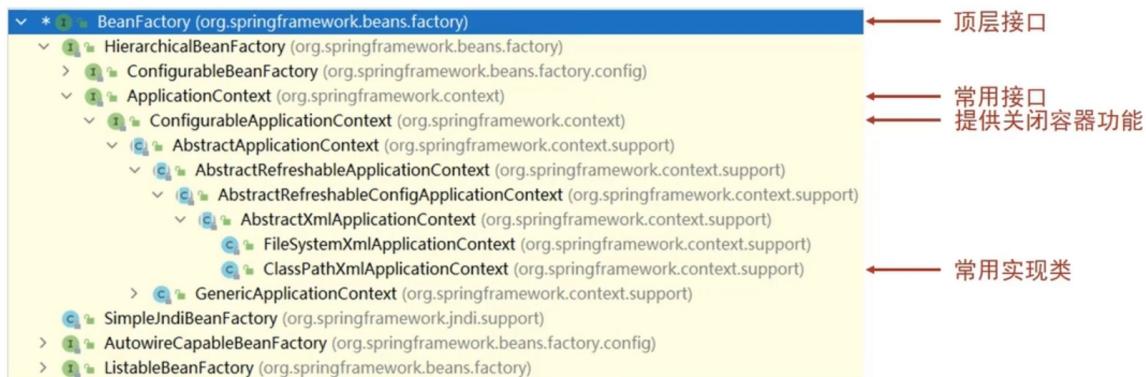
```

补充：system-properties-mode指的是系统properties模块，有些系统properties是先于程序的properties的

想要加载多个properties文件可以在location后的文件名中加逗号，但最标准的还是classpath*:*.properties

3中的是为了打印properties文件

14、容器



- 类路径加载配置文件

```
Resource resources = new ClassPathResource("applicationContext.xml");
BeanFactory bf = new XmlBeanFactory(resources);
BookDao bookDao = bf.getBean("bookDao", BookDao.class);
bookDao.save();
```

- BeanFactory创建完毕后，所有的bean均为延迟加载



15、容器总结

- BeanFactory是IoC容器的顶层接口，初始化BeanFactory对象时，加载的bean延迟加载
- ApplicationContext接口是Spring容器的核心接口，初始化时bean立即加载
- ApplicationContext接口提供基础的bean操作相关方法，通过其他接口扩展其功能
- ApplicationContext接口常用初始化类
 - ClassPathXmlApplicationContext
 - FileSystemXmlApplicationContext



<bean	
id="bookDao"	bean的Id
name="dao bookDaoImpl daoImpl"	bean别名
class="com.itheima.dao.impl.BookDaoImpl"	bean类型，静态工厂类，FactoryBean类
scope="singleton"	控制bean的实例数量
init-method="init"	生命周期初始化方法
destroy-method="destory"	生命周期销毁方法
autowire="byType"	自动装配类型
factory-method="getInstance"	bean工厂方法，应用于静态工厂或实例工厂
factory-bean="com.itheima.factory.BookDaoFactory"	实例工厂bean
lazy-init="true"	控制bean延迟加载
/	

```

<bean id="bookService" class="com.itheima.service.impl.BookServiceImpl">
    <constructor-arg name="bookDao" ref="bookDao"/>                                构造器注入引用类型
    <constructor-arg name="userDao" ref="userDao"/>
    <constructor-arg name="msg" value="WARN"/>
    <constructor-arg type="java.lang.String" index="3" value="WARN"/>                构造器注入简单类型
    <property name="bookDao" ref="bookDao"/>
    <property name="userDao" ref="userDao"/>
    <property name="msg" value="WARN"/>                                              类型匹配与索引匹配
    <property name="names">
        <list>
            <value>itcast</value>
            <ref bean="dataSource"/>
        </list>
    </property>
</bean>

```

16、注解开发定义bean

```

//@Component定义bean
@Component("bookDao")
public class BookDaoImpl implements BookDao {
    public void save() {
        System.out.println("book dao save ...");
    }
}

```

xml中配置的文件不用写<bean了

应改为

```
<context:component-scan base-package="com.itheima"/>
```

Component中也可以不写名

如：

```
@Component
public class BookServiceImpl implements BookService {
    private BookDao bookDao;

    public void setBookDao(BookDao bookDao) {
        this.bookDao = bookDao;
    }

    public void save() {
        System.out.println("book service save ...");
        bookDao.save();
    }
}
```

但启动程序中要改为：

```
BookService bookService = ctx.getBean(BookService.class);
```

17、纯注解开发

1、配置文件都可以不用写，但是要创一个配置类。

```
//声明当前类为Spring配置类
@Configuration
//设置bean扫描路径，多个路径书写为字符串数组格式
@ComponentScan({"com.itheima.service","com.itheima.dao"})
public class SpringConfig {
```

@ComponentScan 括号中的要么是String，要么是数组

2、然后改一下启动程序

```
public class AppForAnnotation {
    public static void main(String[] args) {
        //AnnotationConfigApplicationContext加载Spring配置类初始化Spring容器
        ApplicationContext ctx = new AnnotationConfigApplicationContext(SpringConfig.class);
        BookDao bookDao = (BookDao) ctx.getBean("bookDao");
        System.out.println(bookDao);
        //按类型获取bean
        BookService bookService = ctx.getBean(BookService.class);
        System.out.println(bookService);
    }
}
```

补充：

```
//@Component定义bean
//@Component("bookDao")
//@Repository: @Component衍生注解
@Repository("bookDao")
public class BookDaoImpl implements BookDao {
    public void save() {
        System.out.println("book dao save ...");
    }
}
```

```
//@Component定义bean
//@Component
//@Service: @Component衍生注解
@Service
public class BookServiceImpl implements BookService {
    private BookDao bookDao;

    public void setBookDao(BookDao bookDao) {
        this.bookDao = bookDao;
    }

    public void save() {
        System.out.println("book service save ...");
        bookDao.save();
    }
}
```

18、注解开发bean与生命周期

```
//@Scope设置bean的作用范围  
//@Scope("singleton")//单例  
//@Scope("prototype") 非单例  
public class BookDaoImpl implements BookDao {  
  
    public void save() {  
        System.out.println("book dao save ...");  
    }  
  
}
```

然后是生命周期

```
@PostConstruct//开始  
public void init() {  
    System.out.println("init ...");  
}  
//@PreDestroy设置bean的销毁方法  
@PreDestroy//结束  
public void destroy() {  
    System.out.println("destroy ...");
```

最后记得close

```
public static void main(String[] args) {  
    AnnotationConfigApplicationContext ctx = new  
    AnnotationConfigApplicationContext(SpringConfig.class);  
    BookDao bookDao1 = ctx.getBean(BookDao.class);  
    BookDao bookDao2 = ctx.getBean(BookDao.class);  
    System.out.println(bookDao1);  
    System.out.println(bookDao2);  
    ctx.close();  
}
```

补充：

@PostConstruct和 @PreDestroy注解位于 java.xml.ws.annotation包是Java EE的模块的一部分。
J2EE已经在Java 9中被弃用，并且计划在Java 11中删除它。

解决办法：

为pom.xml或build.gradle添加必要的依赖项

(Java 9+中的Spring @PostConstruct和@PreDestroy替代品)

```
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```

19、注解开发依赖注入

只有自动装配，因为注解是便于开发的，所以阉割了部分内容。

```
public class BookServiceImpl implements BookService {
    // @Autowired: 注入引用类型，自动装配模式，默认按类型装配
    @Autowired
    private BookDao bookDao;

    public void save() {
        System.out.println("book service save ...");
        bookDao.save();
    }
}
```

如果有不同类型的bean,要加 @Qualifier(" ") , @Autowired 必须要加

```
@Service
public class BookServiceImpl implements BookService {
    // @Autowired: 注入引用类型，自动装配模式，默认按类型装配
    @Autowired
    // @Qualifier: 自动装配bean时按bean名称装配
    @Qualifier("bookDao")
    private BookDao bookDao;

    public void save() {
        System.out.println("book service save ...");
        bookDao.save();
    }
}
```

- 注意：自动装配基于反射设计创建对象并暴力反射对应属性为私有属性初始化数据，因此无需提供setter方法
- 注意：自动装配建议使用无参构造方法创建对象（默认），如果不提供对应构造方法，请提供唯一的构造方法

如果要注入值

```
@Repository("bookDao")
public class BookDaoImpl implements BookDao {
    // @Value: 注入简单类型（无需提供set方法）
    @Value("name")
    private String name;

    public void save() {
        System.out.println("book dao save ..." + name);
    }
}
```

如果要读取properties文件

```
@Repository("bookDao")
public class BookDaoImpl implements BookDao {
    // @Value: 注入简单类型（无需提供set方法）
    @Value("${name}")
    private String name;

    public void save() {
        System.out.println("book dao save ..." + name);
    }
}
```

然后配置文件要加

```
@Configuration
@ComponentScan("com.itheima")
// @PropertySource加载properties配置文件
@PropertySource({"jdbc.properties"})
public class SpringConfig {
```

20、注解开发管理第三方bean

如果要管理第三方bean，要专门创一个配置文件JdbcConfig

```
//@Configuration  
public class JdbcConfig {  
    //1.定义一个方法获得要管理的对象  
    //2.添加@Bean，表示当前方法的返回值是一个bean  
    //对@Bean修饰的方法，形参根据类型自动装配  
    @Bean  
    public DataSource dataSource(){  
        DruidDataSource ds = new DruidDataSource();  
        ds.setDriverClassName(driver);  
        ds.setUrl(url);  
        ds.setUsername(userName);  
        ds.setPassword(password);  
        return ds;  
    }  
}
```

然后在SpringConfig开Import注解

```
@Configuration  
@ComponentScan("com.itheima")  
//@Import:导入配置信息  
@Import({JdbcConfig.class})  
public class SpringConfig {  
}
```

21、注解开发实现为第三方bean注入资源

先注入简单类型

```
//@Configuration  
public class JdbcConfig {  
    //1.定义一个方法获得要管理的对象  
    @Value("com.mysql.jdbc.Driver")  
    private String driver;  
    @Value("jdbc:mysql://localhost:3306/spring_db")  
    private String url;  
    @Value("root")  
    private String userName;
```

```

@Value("root")
private String password;
//2.添加@Bean，表示当前方法的返回值是一个bean
//@Bean修饰的方法，形参根据类型自动装配
@Bean
public DataSource dataSource(BookDao bookDao){
    System.out.println(bookDao);
    DruidDataSource ds = new DruidDataSource();
    ds.setDriverClassName(driver);
    ds.setUrl(url);
    ds.setUsername(userName);
    ds.setPassword(password);
    return ds;
}
}

```

注意：

传入的bookDao是被自动装配过的，所以可以打印。（按类型装配）

22、注解开发总结

XML配置比对注解配置



功能	XML配置	注解
定义bean	bean标签 ● id属性 ● class属性	@Component ● @Controller ● @Service ● @Repository @ComponentScan
设置依赖注入	setter注入(set方法) ● 引用/简单 构造器注入(构造方法) ● 引用/简单 自动装配	@Autowired ● @Qualifier @Value
配置第三方bean	bean标签 静态工厂、实例工厂、FactoryBean	@Bean
作用范围	● scope属性	@Scope
生命周期	标准接口 ● init-method ● destroy-method	@PostConstructor @PreDestroy

23、Spring整合Mybatis思路分析

- MyBatis程序核心对象分析

```
// 1. 创建SqlSessionFactoryBuilder对象
SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();
// 2. 加载SqlMapConfig.xml配置文件
InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
// 3. 创建SqlSessionFactory对象
SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);
// 4. 获得SqlSession
SqlSession sqlSession = sqlSessionFactory.openSession();
// 5. 执行SqlSession对象执行查询，获取结果User
AccountDao accountDao = sqlSession.getMapper(AccountDao.class);
Account ac = accountDao.findById(2);
System.out.println(ac);
// 6. 释放资源
sqlSession.close();
```

初始化SqlSessionFactory
获取连接，获取实现
获取数据层接口
关闭连接

- 整合MyBatis

```
<configuration>
    <properties resource="jdbc.properties"></properties>
    <typeAliases>
        <package name="com.itheima.domain"/>
    </typeAliases>
    <environments default="mysql">
        <environment id="mysql">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"></property>
                <property name="url" value="${jdbc.url}"></property>
                <property name="username" value="${jdbc.username}"></property>
                <property name="password" value="${jdbc.password}"></property>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <package name="com.itheima.dao"></package>
    </mappers>
</configuration>
```

初始化属性数据
初始化类型别名
初始化dataSource
初始化映射配置

24、Spring整合MyBatis

1、配置pom文件

```
<?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>com.itheima</groupId>
    <artifactId>spring_15_spring_mybatis</artifactId>
    <version>1.0-SNAPSHOT</version>
```

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.10.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.16</version>
    </dependency>

    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.6</version>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.2.10.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>1.3.0</version>
    </dependency>

</dependencies>
</project>
```

2、配置SpringConfig

```
package com.itheima.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.PropertySource;

@Configuration
@ComponentScan("com.itheima")
//@PropertySource: 加载类路径jdbc.properties文件
```

```
@PropertySource("classpath:jdbc.properties")
@Import({JdbcConfig.class,MybatisConfig.class})
//要么在JdbcConfig类中加@Configuration
public class SpringConfig {
}
```

3、配置JdbcConfig类

```
package com.itheima.config;

import com.alibaba.druid.pool.DruidDataSource;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;

public class JdbcConfig {
    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String userName;
    @Value("${jdbc.password}")
    private String password;

    @Bean
    public DataSource dataSource(){
        DruidDataSource ds = new DruidDataSource();
        ds.setDriverClassName(driver);
        ds.setUrl(url);
        ds.setUsername(userName);
        ds.setPassword(password);
        return ds;
    }
}
```

3、创建MybatisConfig类

```
package com.itheima.config;

import org.mybatis.spring.SqlSessionFactoryBean;
import org.mybatis.spring.mapper.MappersScannerConfigurer;
import org.springframework.context.annotation.Bean;

import javax.sql.DataSource;
```

```
public class MybatisConfig {
    //定义bean, SqlSessionFactoryBean, 用于产生SqlSessionFactory对象
    @Bean
    public SqlSessionFactoryBean sqlSessionFactory(DataSource dataSource){
        SqlSessionFactoryBean ssfb = new SqlSessionFactoryBean();
        ssfb.setTypeAliasesPackage("com.itheima.domain");
        ssfb.setDataSource(dataSource);
        return ssfb;
    }
    //定义bean, 返回MapperScannerConfigurer对象
    @Bean
    public MapperScannerConfigurer mapperScannerConfigurer(){
        MapperScannerConfigurer msc = new MapperScannerConfigurer();
        msc.setBasePackage("com.itheima.dao");
        return msc;
    }
}
```

4、配置启动类

```
import com.itheima.config.SpringConfig;
import com.itheima.domain.Account;
import com.itheima.service.AccountService;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App2 {
    public static void main(String[] args) {
        ApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);

        AccountService accountService = ctx.getBean(AccountService.class);

        Account ac = accountService.findById(1);
        System.out.println(ac);
    }
}
```

- 整合MyBatis

```

<configuration>
    <properties resource="jdbc.properties"></properties>
    <typeAliases>
        <package name="com.itheima.domain"/>
    </typeAliases>
    <environments default="mysql">
        <environment id="mysql">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"></property>
                <property name="url" value="${jdbc.url}"></property>
                <property name="username" value="${jdbc.username}"></property>
                <property name="password" value="${jdbc.password}"></property>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <package name="com.itheima.domain" />
    </mappers>
</configuration>

```

@Bean
public SqlSessionFactoryBean sqlSessionFactory(DataSource dataSource){
 SqlSessionFactoryBean ssfb = new SqlSessionFactoryBean();
 ssfb.setTypeAliasesPackage("com.itheima.domain");
 ssfb.setDataSource(dataSource);
 return ssfb;
}

- 整合MyBatis

```

<configuration>
    <properties resource="jdbc.properties"></properties>
    <typeAliases>
        <package name="com.itheima.domain"/>
    </typeAliases>
    <environments default="mysql">
        <environment id="mysql">
            <transactionManager type="JDBC">
                <dataSource type="POOLED">
                    <property name="driver" value="com.mysql.jdbc.Driver" />
                    <property name="url" value="jdbc:mysql://127.0.0.1:3306/test" />
                    <property name="username" value="root" />
                    <property name="password" value="123456" />
                </dataSource>
            </environment>
        </environments>
        <mappers>
            <package name="com.itheima.dao" />
        </mappers>
    </configuration>

```

@Bean
public MapperScannerConfigurer mapperScannerConfigurer(){
 MapperScannerConfigurer msc = new MapperScannerConfigurer();
 msc.setBasePackage("com.itheima.dao");
 return msc;
}

25、JUnit整合

JUnit介绍：测试框架

1、导包

```

<?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>com.itheima</groupId>
    <artifactId>spring_16_spring_junit</artifactId>

```

```
<version>1.0-SNAPSHOT</version>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.10.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.16</version>
    </dependency>

    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.6</version>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.2.10.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>1.3.0</version>
    </dependency>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.2.10.RELEASE</version>
    </dependency>

</dependencies>
</project>
```

2、配置运行器

```
package com.itheima.service;

import com.itheima.config.SpringConfig;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
//设置类运行器
@RunWith(SpringJUnit4ClassRunner.class)
//设置Spring环境对应的配置类
@ContextConfiguration(classes = SpringConfig.class)
public class AccountServiceTest {
    //支持自动装配注入bean(测谁写谁)
    @Autowired
    private AccountService accountService;

    @Test
    public void testFindById(){
        System.out.println(accountService.findById(1));
    }

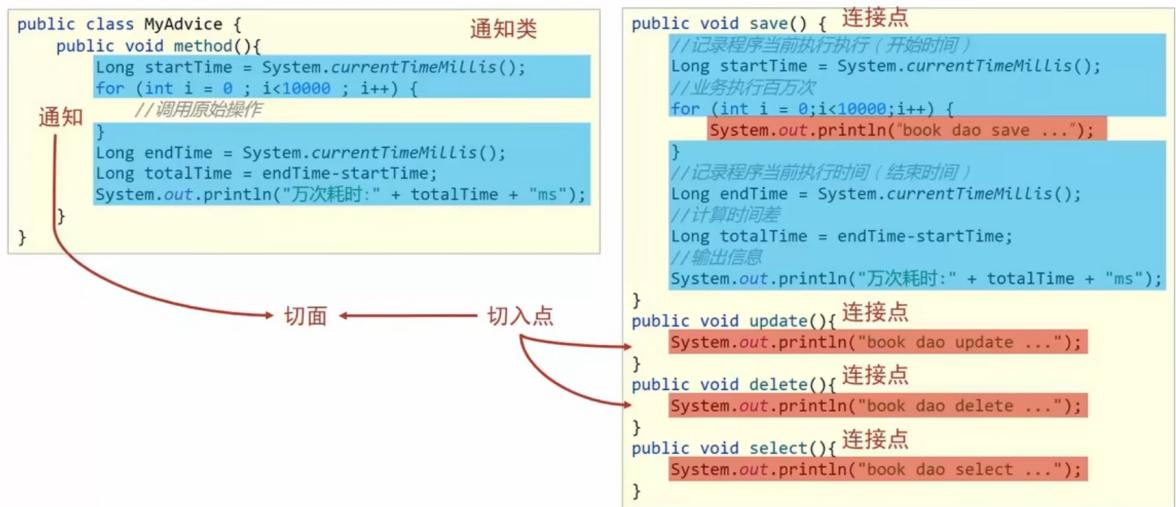
    @Test
    public void testfindAll(){
        System.out.println(accountService.findAll());
    }

}
```

26、AOP简介

AOP简介

- AOP(Aspect Oriented Programming)面向切面编程，一种编程范式，指导开发者如何组织程序结构
 - OOP(Object Oriented Programming)面向对象编程
- 作用：在不惊动原始设计的基础上对其进行功能增强
- Spring理念：无入侵式/无侵入式



- 连接点 (JoinPoint) : 程序执行过程中的任意位置 , 粒度为执行方法、抛出异常、设置变量等
 - 在SpringAOP中 , 理解为方法的执行
- 切入点 (Pointcut) : 匹配连接点的式子
 - 在SpringAOP中 , 一个切入点可以只描述一个具体方法 , 也可以匹配多个方法
 - ◆ 一个具体方法: com.itheima.dao包下的BookDao接口中的无形参无返回值的save方法
 - ◆ 匹配多个方法: 所有的save方法 , 所有的get开头的方法 , 所有以Dao结尾的接口中的任意方法 , 所有带有一个参数的方法
- 通知 (Advice) : 在切入点处执行的操作 , 也就是共性功能
 - 在SpringAOP中 , 功能最终以方法的形式呈现
- 通知类 : 定义通知的类
- 切面 (Aspect) : 描述通知与切入点的对应关系

27、AOP入门案例

1、配置pom文件

```

<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.4</version>
</dependency>

```

2、制作通知

```
@Component
```

```
//设置当前类为切面类
@Aspect
//声明AOP
public class MyAdvice {
    //设置切入点，要求配置在方法上方
    @Pointcut("execution(void com.itheima.dao.BookDao.update())")
    private void pt(){}//方法要无参数，无返回值，无实际逻辑

    //设置在切入点pt()的前面运行当前操作（前置通知）
    // @Before("pt()")
    public void method(){
        System.out.println(System.currentTimeMillis());
    }
}
```

3、配置配置类

```
@Configuration
@ComponentScan("com.itheima")
//开启注解开发AOP功能
@EnableAspectJAutoProxy
public class SpringConfig {
```

28、AOP工作流程

AOP入门案例思路分析

案例设定：测定接口执行效率

简化设定：在接口执行前输出当前系统时间

开发模式：XML or **注解**

思路分析：

1. 导入坐标 (pom.xml)
2. 制作连接点方法 (原始操作，Dao接口与实现类)
3. 制作共性功能 (通知类与通知)
4. 定义切入点
5. 绑定切入点与通知关系 (切面)

AOP工作流程

1. Spring容器启动
2. 读取所有切面配置中的切入点
3. 初始化bean，判定bean对应的类中的方法是否匹配到任意切入点
 - 匹配失败，创建对象
 - 匹配成功，创建原始对象（**目标对象**）的**代理**对象
4. 获取bean执行方法
 - 获取bean，调用方法并执行，完成操作
 - 获取的bean是代理对象时，根据代理对象的运行模式运行原始方法与增强的内容，完成操作

AOP核心概念

- 目标对象（Target）：原始功能去掉共性功能对应的类产生的对象，这种对象是无法直接完成最终工作的
- 代理（Proxy）：目标对象无法直接完成工作，需要对其进行功能回填，通过原始对象的代理对象实现

29、AOP切入点表达式

AOP切入点表达式



- 切入点：要进行增强的方法
- 切入点表达式：要进行增强的方法的描述方式

```
package com.itheima.dao;
public interface BookDao {
    public void update();
}
```

```
public class BookDaoImpl implements BookDao {
    public void update(){
        System.out.println("book dao update ...");
    }
}
```

描述方式一：执行com.itheima.dao包下的BookDao接口中的无参数update方法
`execution(void com.itheima.dao.BookDao.update())`

描述方式二：执行com.itheima.dao.impl包下的BookDaoImpl类中的无参数update方法
`execution(void com.itheima.dao.impl.BookDaoImpl.update())`

AOP切入点表达式

- 切入点表达式标准格式：动作关键字（访问修饰符 返回值 包名.类/接口名.方法名（参数）异常名）

```
execution (public User com.itheima.service.UserService.findById (int) )
```

- 动作关键字：描述切入点的行为动作，例如execution表示执行到指定切入点
- 访问修饰符：public , private等，可以省略
- 返回值
- 包名
- 类/接口名
- 方法名
- 参数
- 异常名：方法定义中抛出指定异常，可以省略

AOP切入点表达式

- 可以使用通配符描述切入点，快速描述

- * : 单个独立的任意符号，可以独立出现，也可以作为前缀或者后缀的匹配符出现

```
execution (public * com.itheima.*.UserService.find* (*) )
```

匹配com.itheima包下的任意包中的UserService类或接口中所有find开头的带有一个参数的方法

- .. : 多个连续的任意符号，可以独立出现，常用于简化包名与参数的书写

```
execution (public User com..UserService.findById(..) )
```

匹配com包下的任意包中的UserService类或接口中所有名称为findById的方法

- + : 专用于匹配子类类型

```
execution(* *..*Service+.*(..))
```

```
public class MyAdvice {  
    //切入点表达式:  
    // @Pointcut("execution(void com.itheima.dao.BookDao.update())")  
    // @Pointcut("execution(void com.itheima.dao.impl.BookDaoImpl.update())")  
    // @Pointcut("execution(* com.itheima.dao.impl.BookDaoImpl.update(*))")  
    //no (这个不行的原因是原本BookDaoImpl.update()是没参数的，但*号强制要有参数)  
    // @Pointcut("execution(void com.*.*.*.update())")  
    // @Pointcut("execution(* *..*(..))")  
    // @Pointcut("execution(* *..*e(..))")  
    // @Pointcut("execution(void com..*(..))")  
    // @Pointcut("execution(* com.itheima.*.*Service.find*(..))")  
    //执行com.itheima包下的任意包下的名称以Service结尾的类或接口中的save方法，参数任意，返回值任意  
    @Pointcut("execution(* com.itheima.*.*Service.save(..))")
```

```

private void pt() {}

@Before("pt()")
public void method(){
    System.out.println(System.currentTimeMillis());
}
}

```

- 书写技巧

- 所有代码按照标准规范开发，否则以下技巧全部失效
- 描述切入点通常描述接口，而不描述实现类
- 访问控制修饰符针对接口开发均采用public描述（可省略访问控制修饰符描述）
- 返回值类型对于增删改类使用精准类型加速匹配，对于查询类使用*通配快速描述
- 包名书写尽量不使用*匹配，效率过低，常用*做单个包描述匹配，或精准匹配
- 接口名/类名书写名称与模块相关的采用*匹配，例如UserService书写成*Service，绑定业务层接口名
- 方法名书写以动词进行精准匹配，名词采用*匹配，例如getById书写成getBy*,selectAll书写成selectAll
- 参数规则较为复杂，根据业务方法灵活调整
- 通常不使用异常作为匹配规则

30、AOP通知类型

- AOP通知描述了抽取的共性功能，根据共性功能抽取的位置不同，最终运行代码时要将其加入到合理的位置
- AOP通知共分为5种类型
 - 前置通知
 - 后置通知
 - 环绕通知（重点）
 - 返回后通知（了解）
 - 抛出异常后通知（了解）

```

public class MyAdvice {
    @Pointcut("execution(void com.itheima.dao.BookDao.update())")
    private void pt(){}
    @Pointcut("execution(int com.itheima.dao.BookDao.select())")
    private void pt2(){}

    // @Before: 前置通知，在原始方法运行之前执行
    // @Before("pt()")
    public void before() {
        System.out.println("before advice ...");
    }
}

```

```

// @After: 后置通知，在原始方法运行之后执行
//    @After("pt2()")
public void after() {
    System.out.println("after advice ...");
}

// @Around: 环绕通知，在原始方法运行的前后执行
//    @Around("pt()")
public Object around(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("around before advice ...");
    // 表示对原始操作的调用，如果原来的函数有返回值，那这里也有
    Object ret = pjp.proceed();
    System.out.println("around after advice ...");
    return ret;
}

//    @Around("pt2()")
public Object aroundSelect(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("around before advice ...");
    // 表示对原始操作的调用
    Integer ret = (Integer) pjp.proceed();
    System.out.println("around after advice ...");
    return ret;
}

// @AfterReturning: 返回后通知，在原始方法执行完毕后运行，且原始方法执行过程中未出现异常现象
//    @AfterReturning("pt2()")
public void afterReturning() {
    System.out.println("afterReturning advice ...");
}

// @AfterThrowing: 抛出异常后通知，在原始方法执行过程中出现异常后运行
@AfterThrowing("pt2()")
public void afterThrowing() {
    System.out.println("afterThrowing advice ...");
}

```

● @Around注意事项

1. 环绕通知必须依赖形参ProceedingJoinPoint才能实现对原始方法的调用，进而实现原始方法调用前后同时添加通知
2. 通知中如果未使用ProceedingJoinPoint对原始方法进行调用将跳过原始方法的执行
3. 对原始方法的调用可以不接收返回值，通知方法设置成void即可，如果接收返回值，必须设定为Object类型
4. 原始方法的返回值如果是void类型，通知方法的返回值类型可以设置成void，也可以设置成Object
5. 由于无法预知原始方法运行后是否会抛出异常，因此环绕通知方法必须抛出Throwable对象

```

@Around("pt()")
public Object around(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("around before advice ...");
    Object ret = pjp.proceed();
    System.out.println("around after advice ...");
    return ret;
}

```

- 名称：@AfterThrowing（了解）
- 类型：**方法注解**
- 位置：通知方法定义上方
- 作用：设置当前通知方法与切入点之间的绑定关系，当前通知方法在原始切入点方法运行抛出异常后执行
- 范例：

```
@AfterThrowing("pt()")
public void afterThrowing() {
    System.out.println("afterThrowing advice ...");
}
```

- 相关属性：value（默认）：切入点方法名，格式为类名.方法名()

31、案例-业务层接口执行效率

拿方法信息

拿类的信息：signature.getDeclaringTypeName()

拿方法的信息：signature.getName()

```
package com.itheima.aop;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class ProjectAdvice {
    //匹配业务层的所有方法
    @Pointcut("execution(* com.itheima.service.*Service.*(..))")
    private void servicePt(){}

    //设置环绕通知，在原始操作的运行前后记录执行时间
    @Around("ProjectAdvice.servicePt()")
    public void runSpeed(ProceedingJoinPoint pjp) throws Throwable {
        //获取执行的签名对象
        Signature signature = pjp.getSignature();
        String className = signature.getDeclaringTypeName();
        String methodName = signature.getName();

        long start = System.currentTimeMillis();
    }
}
```

```

        for (int i = 0; i < 10000; i++) {
            pjp.proceed();
        }
        long end = System.currentTimeMillis();
        System.out.println("万次执行: " + className + "." + methodName + "---->" +(end-start) + "ms");
    }

}

```

32、AOP通知获取数据

- 获取切入点方法的参数
 - JoinPoint：适用于前置、后置、返回后、抛出异常后通知
 - ProceedJointPoint：适用于环绕通知
- 获取切入点方法返回值
 - 返回后通知
 - 环绕通知
- 获取切入点方法运行异常信息
 - 抛出异常后通知
 - 环绕通知

```

public class MyAdvice {
    @Pointcut("execution(* com.itheima.dao.BookDao.findName(..))")
    private void pt() {}

    //JoinPoint: 用于描述切入点的对象，必须配置成通知方法中的第一个参数，可用于获取原始方法调用的参数
    //    @Before("pt()")
    public void before(JoinPoint jp) {
        Object[] args = jp.getArgs();
        System.out.println(Arrays.toString(args));
        System.out.println("before advice ...");
    }

    //    @After("pt()")
    public void after(JoinPoint jp) {
        Object[] args = jp.getArgs();
        System.out.println(Arrays.toString(args));
        System.out.println("after advice ...");
    }

    //ProceedingJoinPoint: 专用于环绕通知，是JoinPoint子类，可以实现对原始方法的调用
}

```

```

//    @Around("pt()")
public Object around(ProceedingJoinPoint pjp) {
    Object[] args = pjp.getArgs();
    System.out.println(Arrays.toString(args));
    args[0] = 666;
    Object ret = null;
    try {
        ret = pjp.proceed(args);
    } catch (Throwable t) {
        t.printStackTrace();
    }
    return ret;
}

//设置返回后通知获取原始方法的返回值，要求returning属性值必须与方法形参名相同，JoinPoint
jp必须在第零位
@AfterReturning(value = "pt()",returning = "ret")
public void afterReturning(JoinPoint jp,String ret) {
    System.out.println("afterReturning advice ..."+ret);
}

//设置抛出异常后通知获取原始方法运行时抛出的异常对象，要求throwing属性值必须与方法形参名相
同
@AfterThrowing(value = "pt()",throwing = "t")
public void afterThrowing(Throwable t) {
    System.out.println("afterThrowing advice ..."+t);
}

```

案例 测量业务层接口执行效率

需求：任意业务层接口执行均可显示其执行效率（执行时长）

分析：

①：业务功能：业务层接口执行前后分别记录时间，求差值得到执行效率

②：通知类型选择前后均可以增强的类型——环绕通知

补充说明

当前测试的接口执行效率仅仅是一个理论值，并不是一次完整的执行过程

33、案例-百度网盘密码数据兼容处理



百度网盘分享链接输入密码数据错误兼容性处理

需求：对百度网盘分享链接输入密码时尾部多输入的空格做兼容处理

链接：<https://pan.baidu.com/s/16WHAPgAe8ncZO3hmlS5Q>

提取码：**ezki**

复制这段内容后打开百度网盘手机App，操作更方便哦--来自百度

分析：

①：在业务方法执行之前对所有的输入参数进行格式处理——trim()

②：使用处理后的参数调用原始方法——环绕通知中存在对原始方法的调用

»

```
package com.itheima.aop;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class DataAdvice {
    @Pointcut("execution(boolean com.itheima.service.*Service.*(*, *))")
    private void servicePt() {}

    @Around("DataAdvice.servicePt()")
    public Object trimStr(ProceedingJoinPoint pjp) throws Throwable {
        Object[] args = pjp.getArgs();
        for (int i = 0; i < args.length; i++) {
            //判断参数是不是字符串(如果是就让回去, 不是就null)
            if(args[i].getClass().equals(String.class)){
                args[i] = args[i].toString().trim();
            }
        }
        Object ret = pjp.proceed(args);
        return ret;
    }

}
```

34、AOP总结

- 概念：AOP(Aspect Oriented Programming)面向切面编程，一种编程范式
- 作用：在不惊动原始设计的基础上为方法进行功能**增强**
- 核心概念
 - 代理（Proxy）：SpringAOP的核心本质是采用代理模式实现的
 - 连接点（JoinPoint）：在SpringAOP中，理解为任意方法的执行
 - 切入点（Pointcut）：匹配连接点的式子，也是具有共性功能的方法描述
 - 通知（Advice）：若干个方法的共性功能，在切入点处执行，最终体现为一个方法
 - 切面（Aspect）：描述通知与切入点的对应关系
 - 目标对象（Target）：被代理的原始对象成为目标对象

- 切入点表达式标准格式：动作关键字（访问修饰符 返回值 包名.类/接口名.方法名（参数）异常名）
 - `execution(* com.itheima.service.*Service.*(..))`
- 切入点表达式描述通配符：
 - 作用：用于快速描述，范围描述
 - *：匹配任意符号（常用）
 - ..：匹配多个连续的任意符号（常用）
 - +：匹配子类类型

- 切入点表达式标准格式：动作关键字（访问修饰符 返回值 包名.类/接口名.方法名（参数）异常名）
 - `execution(* com.itheima.service.*Service.*(..))`
- 切入点表达式描述通配符：
 - 作用：用于快速描述，范围描述
 - *：匹配任意符号（常用）
 - ..：匹配多个连续的任意符号（常用）
 - +：匹配子类类型
- 切入点表达式书写技巧
 1. 按**标准规范**开发
 2. 查询操作的返回值建议使用*匹配
 3. 减少使用..的形式描述包
 4. **对接口进行描述**，使用*表示模块名，例如UserService的匹配描述为*Service
 5. 方法名书写保留动词，例如get，使用*表示名词，例如getById匹配描述为getBy*
 6. 参数根据实际情况灵活调整

- 通知类型
 - 前置通知
 - 后置通知
 - 环绕通知（重点）
 - ◆ 环绕通知依赖形参ProceedingJoinPoint才能实现对原始方法的调用
 - ◆ 环绕通知可以隔离原始方法的调用执行
 - ◆ 环绕通知返回值设置为Object类型
 - ◆ 环绕通知中可以对原始方法调用过程中出现的异常进行处理
 - 返回后通知
 - 抛出异常后通知
- 获取切入点方法的参数
 - JoinPoint：适用于前置、后置、返回后、抛出异常后通知，设置为方法的第一个形参
 - ProceedJointPoint：适用于环绕通知
- 获取切入点方法返回值
 - 返回后通知
 - 环绕通知
- 获取切入点方法运行异常信息
 - 抛出异常后通知
 - 环绕通知

35、Spring事务简介

事务就是没报错时，一齐执行，

有报错时，不执行

1、配置事务

```
public interface AccountService {  
    /**  
     * 转账操作  
     * @param out 传出方  
     * @param in 转入方  
     * @param money 金额  
     */  
    //配置当前接口方法具有事务  
    @Transactional  
    public void transfer(String out, String in, Double money);  
}
```

Spring注解事务一般注释到接口中，降低耦合。

可以添加到类中，开启所有事务。

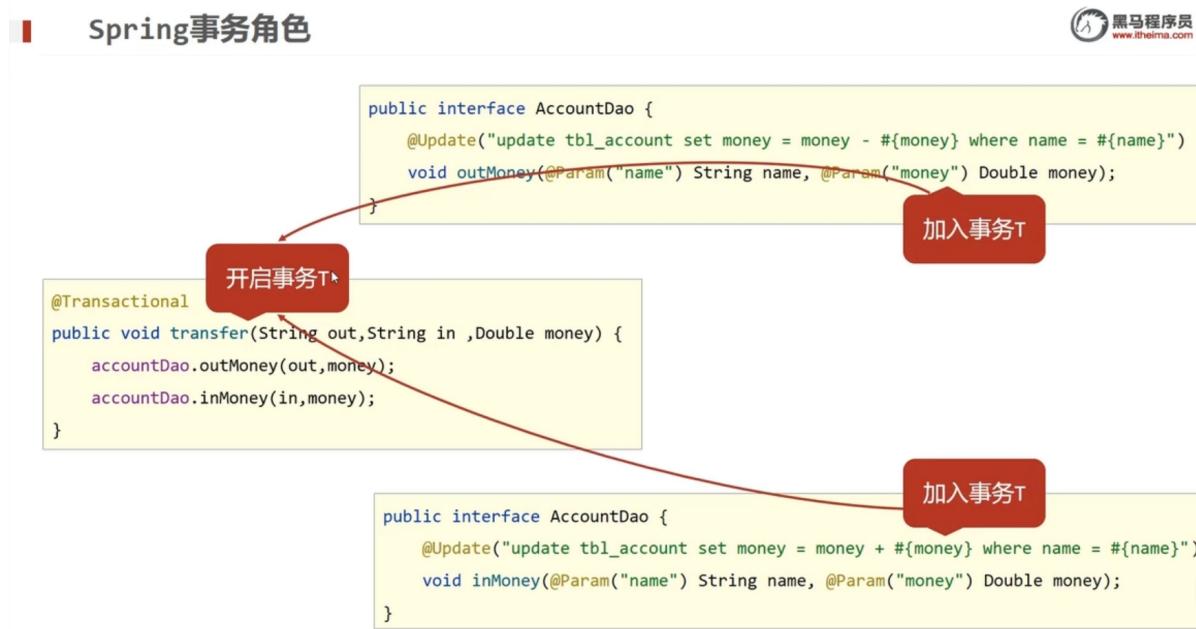
2、设置事务管理器

```
//配置事务管理器，mybatis使用的是jdbc事务  
@Bean  
public PlatformTransactionManager transactionManager(DataSource dataSource){  
    DataSourceTransactionManager transactionManager = new  
    DataSourceTransactionManager();  
    transactionManager.setDataSource(dataSource);  
    return transactionManager;  
}
```

3、开启注解式事务驱动

```
@Configuration  
@ComponentScan("com.itheima")  
@PropertySource("classpath:jdbc.properties")  
@Import({JdbcConfig.class, MybatisConfig.class})  
//开启注解式事务驱动  
@EnableTransactionManagement  
public class SpringConfig {  
}
```

36、Spring事务角色



● 事务角色

- 事务管理员：发起事务方，在Spring中通常指代业务层开启事务的方法
- 事务协调员：加入事务方，在Spring中通常指代数据层方法，也可以是业务层方法

37、spring事务属性

出现事务回滚的异常：1、`RuntimeException`系的。2、运行时异常

在 `@Transactional()` 中添加

属性	作用	示例
readOnly	设置是否为只读事务	readOnly=true 只读事务
timeout	设置事务超时时间	timeout = -1 (永不超时)
rollbackFor	设置事务回滚异常 (class)	rollbackFor = {NullPointerException.class}
rollbackForClassName	设置事务回滚异常 (String)	同上格式为字符串
noRollbackFor	设置事务不回滚异常 (class)	noRollbackFor = {NullPointerException.class}
noRollbackForClassName	设置事务不回滚异常 (String)	同上格式为字符串
propagation	设置事务传播行为

传播属性	事务管理员	事务协调员
REQUIRED (默认)	开启T 无	加入T 新建T2
REQUIRES_NEW	开启T	新建T2
	无	新建T2
SUPPORTS	开启T 无	加入T 无
NOT_SUPPORTED	开启T	无
	无	无
MANDATORY	开启T 无	加入T ERROR
	无	ERROR
NEVER	开启T	ERROR
	无	无
NESTED	设置savePoint,一旦事务回滚, 事务将回滚到savePoint处, 交由客户响应提交/回滚	

二、SpringMVC

1、SpringMVC简介

SpringMVC概述

- SpringMVC技术与Servlet技术功能等同，均属于web层开发技术

- ◆ SpringMVC简介

- ◆ 请求与响应

- ◆ REST风格

- ◆ SSM整合

- ◆ 拦截器

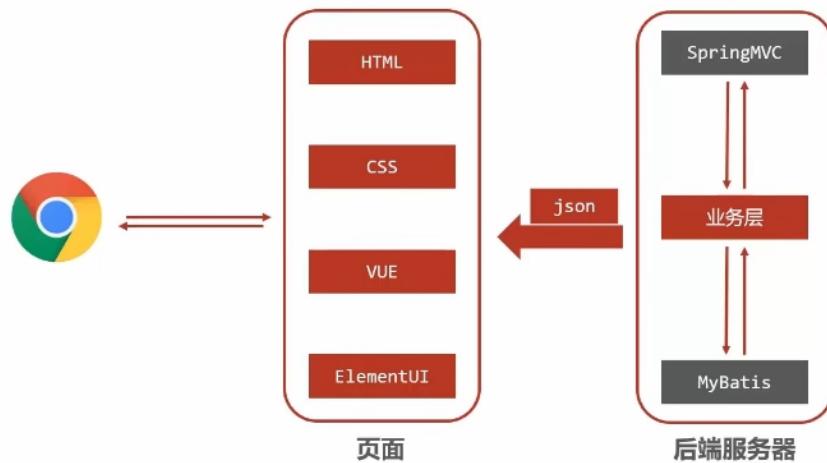
1. 掌握基于SpringMVC获取请求参数与响应json数据操作
2. 熟练应用基于REST风格的请求路径设置与参数传递
3. 能够根据实际业务建立前后端开发通信协议并进行实现
4. 基于SSM整合技术开发任意业务模块功能

01

SpringMVC简介

- SpringMVC概述
- 入门案例 ✓
- 入门案例工作流程分析 ✓
- Controller加载控制
- PostMan

SpringMVC概述



SpringMVC概述

- SpringMVC是一种基于Java实现MVC模型的轻量级Web框架

SpringMVC概述

- SpringMVC是一种基于Java实现MVC模型的轻量级Web框架

- 优点
 - 使用简单，开发便捷（相比于Servlet）

```
@WebServlet("/user/save")
public class UserSaveServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        String name = req.getParameter("name");
        System.out.println("servlet save name ==> " + name);
        resp.setContentType("text/json;charset=utf-8");
        PrintWriter pw = resp.getWriter();
        pw.write("{\"module':'servlet save'}");
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        this.doGet(req, resp);
    }
}
```

```
@Controller
public class UserController{
    @RequestMapping("/save")
    @ResponseBody
    public String save(String name){
        System.out.println("springmvc save name ==> " + name);
        return "{\"module':'springmvc save'}";
    }
}
```

SpringMVC概述

- SpringMVC是一种基于Java实现MVC模型的轻量级Web框架

- 优点
 - 使用简单，开发便捷（相比于Servlet）
 - 灵活性强

1. SpringMVC是一种表现层框架技术

2. SpringMVC用于进行表现层功能开发

2、SpringMVC入门案例

①：使用SpringMVC技术需要先导入SpringMVC坐标与Servlet坐标

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.10.RELEASE</version>
</dependency>
```

②：创建SpringMVC控制器类（等同于Servlet功能）

```
@Controller
public class UserController {
    @RequestMapping("/save")
    @ResponseBody
    public String save(){
        System.out.println("user save ...");
        return "{ 'info' : 'springmvc' }";
    }
}
```

③：初始化SpringMVC环境（同Spring环境），设定SpringMVC加载对应的bean

```
@Configuration
@ComponentScan("com.itheima.controller")
public class SpringMvcConfig {
```

④：初始化Servlet容器，加载SpringMVC环境，并设置SpringMVC技术处理的请求

```
public class ServletContainersInitConfig extends AbstractDispatcherServletInitializer {  
    protected WebApplicationContext createServletApplicationContext() {  
        AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();  
        ctx.register(SpringMvcConfig.class);  
        return ctx;  
    }  
    protected String[] getServletMappings() {  
        return new String[]{"/"};  
    }  
    protected WebApplicationContext createRootApplicationContext() {  
        return null;  
    }  
}
```

- 名称: @Controller
- 类型: **类注解**
- 位置: SpringMVC控制器类定义上方
- 作用: 设定SpringMVC的核心控制器bean
- 范例:

```
@Controller  
public class UserController {  
}
```

- 名称: @RequestMapping
- 类型: **方法注解**
- 位置: SpringMVC控制器方法定义上方
- 作用: 设置当前控制器方法请求访问路径
- 范例:

```
@RequestMapping("/save")  
public void save(){  
    System.out.println("user save ...");  
}
```

- 相关属性
 - value (默认) : 请求访问路径

- SpringMVC入门程序开发总结 (1+N)
 - 一次性工作
 - ◆ 创建工程，设置服务器，加载工程
 - ◆ 导入坐标
 - ◆ 创建web容器启动类，加载SpringMVC配置，并设置SpringMVC请求拦截路径
 - ◆ SpringMVC核心配置类（设置配置类，扫描controller包，加载Controller控制器bean）
 - 多次工作
 - ◆ 定义处理请求的控制器类
 - ◆ 定义处理请求的控制器方法，并配置映射路径（@RequestMapping）与返回json数据（@ResponseBody）

- AbstractDispatcherServletInitializer类是SpringMVC提供的快速初始化Web3.0容器的抽象类
- AbstractDispatcherServletInitializer提供三个接口方法供用户实现
 - createServletApplicationContext()方法，创建Servlet容器时，加载SpringMVC对应的bean并放入WebApplicationContext对象范围中，而WebApplicationContext的作用范围为ServletContext范围，即整个web容器范围

```
protected WebApplicationContext createServletApplicationContext() {
    AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();
    ctx.register(SpringMvcConfig.class);
    return ctx;
}
```

- AbstractDispatcherServletInitializer类是SpringMVC提供的快速初始化Web3.0容器的抽象类
- AbstractDispatcherServletInitializer提供三个接口方法供用户实现
 - createRootApplicationContext()方法，如果创建Servlet容器时需要加载非SpringMVC对应的bean，使用当前方法进行，使用方式同createServletApplicationContext()

```
protected WebApplicationContext createRootApplicationContext() {
    return null;
}
```

- AbstractDispatcherServletInitializer类是SpringMVC提供的快速初始化Web3.0容器的抽象类
- AbstractDispatcherServletInitializer提供三个接口方法供用户实现
 - getServletMappings()方法，设定SpringMVC对应的请求映射路径，设置为/表示拦截所有请求，任意请求都将转入到SpringMVC进行处理

```
protected String[] getServletMappings() {
    return new String[]{"/"};
}
```

3、入门案例工作流程

入门案例工作流程分析

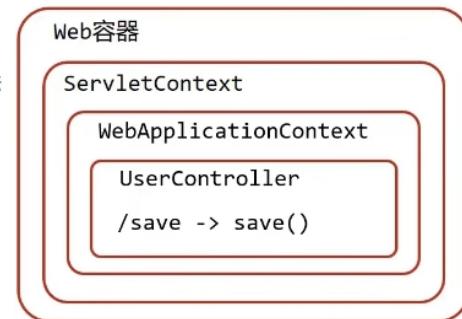
- 启动服务器初始化过程

- 服务器启动，执行ServletContainersInitConfig类，初始化web容器
- 执行createServletApplicationContext方法，创建了WebApplicationContext对象
- 加载SpringMvcConfig
- 执行@ComponentScan加载对应的bean
- 加载UserController，每个@RequestMapping的名称对应一个具体的方法
- 执行getServletMappings方法，定义所有的请求都通过SpringMVC

- 单次请求过程

- 发送请求localhost/save
- web容器发现所有请求都经过SpringMVC，将请求交给SpringMVC处理
- 解析请求路径/save
- 由/save匹配执行对应的方法save()
- 执行save()
- 检测到有@ResponseBody直接将save()方法的返回值作为响应求体返回给请求方

```
localhost/save  
localhost/save  
{'info':'springmvc'}
```



4、bean加载控制

Controller加载控制与业务bean加载控制

- SpringMVC相关bean（表现层bean）

- Spring控制的bean

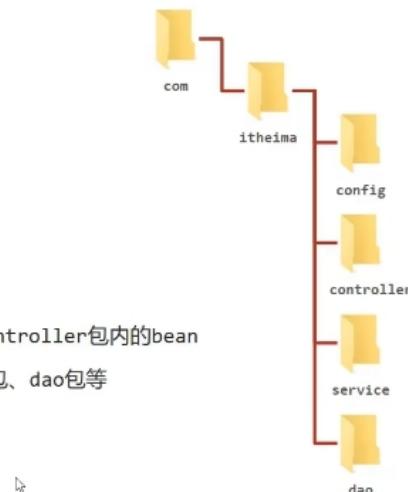
- 业务bean（Service）
- 功能bean（DataSource等）

- SpringMVC相关bean加载控制

- SpringMVC加载的bean对应的包均在com.itheima.controller包内

- Spring相关bean加载控制

- 方式一：Spring加载的bean设定扫描范围为com.itheima，排除掉controller包内的bean
- 方式二：Spring加载的bean设定扫描范围为精准范围，例如service包、dao包等
- 方式三：不区分Spring与SpringMVC的环境，加载到同一个环境中



Controller加载控制与业务bean加载控制

- 名称: @ComponentScan
- 类型: **类注解**
- 范例:

```
@Configuration
@ComponentScan(value = "com.itheima",
    excludeFilters = @ComponentScan.Filter(
        type = FilterType.ANNOTATION,
        classes = Controller.class
    )
)
public class SpringConfig {
```

- 属性

- excludeFilters: 排除扫描路径中加载的bean, 需要指定类别 (type) 与具体项 (classes)
- includeFilters: 加载指定的bean, 需要指定类别 (type) 与具体项 (classes)

Controller加载控制与业务bean加载控制

- bean的加载格式

```
public class ServletContainersInitConfig extends AbstractDispatcherServletInitializer {
    protected WebApplicationContext createServletApplicationContext() {
        AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();
        ctx.register(SpringMvcConfig.class);
        return ctx;
    }
    protected WebApplicationContext createRootApplicationContext() {
        AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();
        ctx.register(SpringConfig.class);
        return ctx;
    }
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }
}
```

Controller加载控制与业务bean加载控制

- 简化开发

```
public class ServletContainersInitConfig extends AbstractAnnotationConfigDispatcherServletInitializer{
    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{SpringMvcConfig.class};
    }
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }
    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{SpringConfig.class};
    }
}
```

5、PostMan工具介绍

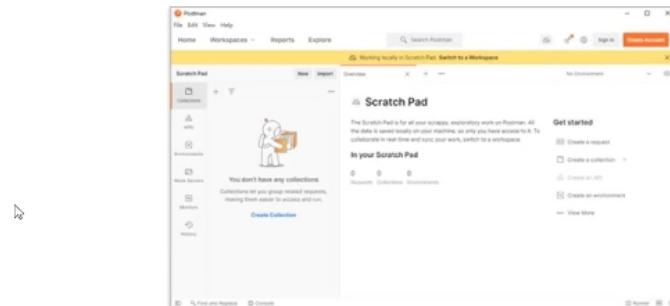
PostMan简介

- Postman是一款功能强大的网页调试与发送网页HTTP请求的Chrome插件
- 作用：常用于进行接口测试
- 特征
 - 简单
 - 实用
 - 美观
 - 大方



PostMan基本使用

- 注册登录
- 创建工作空间/进入工作空间
- 发起请求测试结果



6、设置请求映射路径

02

请求与响应

- 请求映射路径
- 请求参数
- 日期类型参数传递
- 响应json数据

不同的请求路径相同的模块名会报错



1. 团队多人开发，每人设置不同的请求路径，冲突问题如何解决—**设置模块名作为请求路径前缀**

如：

```
@RequestMapping(value="/save")  
public void save()
```

改为：

```
@RequestMapping(value="/user/save")  
public void save()
```

请求映射路径

- 名称: @RequestMapping
- 类型: **方法注解** **类注解**
- 位置: SpringMVC控制器方法定义上方
- 作用: 设置当前控制器方法请求访问路径, 如果设置在类上统一设置当前控制器方法请求访问路径前缀
- 范例:

```
@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping("/save")
    @ResponseBody
    public String save(){
        System.out.println("user save ...");
        return "{\"module':'user save'}";
    }
}
```

- 属性
 - value (默认) : 请求访问路径, 或访问路径前缀

7、get请求与post请求发送普通参数

请求方式

- Get请求
- Post请求

Get请求传参

- 普通参数: url地址传参, 地址参数名与形参变量名相同, 定义形参即可接收参数

The screenshot shows the Postman interface with a GET request to `http://localhost/commonParam?name=itcast&age=15`. The 'Params' tab is selected. In the 'Query Params' section, there are two entries: 'name' with value 'itcast' and 'age' with value '15'. Other tabs include Authorization, Headers, Body, Pre-request Script, Tests, and Settings.

```
@RequestMapping("/commonParam")
@ResponseBody
public String commonParam(String name ,int age){
    System.out.println("普通参数传递 name ==> "+name);
    System.out.println("普通参数传递 age ==> "+age);
    return "{\"module':'common param'}";
}
```

Post请求参数

- 普通参数：form表单post请求传参，表单参数名与形参变量名相同，定义形参即可接收参数

POST http://localhost/commonParam

Params Authorization Headers (9) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL

KEY	VALUE
<input checked="" type="checkbox"/> name	itcast
<input checked="" type="checkbox"/> age	15
Key	Value

```
@RequestMapping("/commonParam")
@ResponseBody
public String commonParam(String name ,int age){
    System.out.println("普通参数传递 name ==> "+name);
    System.out.println("普通参数传递 age ==> "+age);
    return "{\"module':'common param'}";
}
```

Post请求中文乱码处理

- 为web容器添加过滤器并指定字符集，Spring-web包中提供了专用的字符过滤器

```
public class ServletContainersInitConfig extends AbstractAnnotationConfigDispatcherServletInitializer{
    // 配字符编码过滤器
    protected Filter[] getServletFilters() {
        CharacterEncodingFilter filter = new CharacterEncodingFilter();
        filter.setEncoding("utf-8");
        return new Filter[]{filter};
    }
}
```



1. PostMan发送携带参数Get请求
2. PostMan发送携带参数Post请求
3. SpringMVC解决Post请求中文乱码问题

8、5种类型参数传递

请求参数

- 参数种类
 - 普通参数
 - POJO类型参数
 - 嵌套POJO类型参数
 - 数组类型参数
 - 集合类型参数

请求参数

- 普通参数：url地址传参，地址参数名与形参变量名相同，定义形参即可接收参数

GET http://localhost/commonParam?name=itcast&age=15

Params ● Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE
name	itcast
age	15
Key	Value

```
@RequestMapping("/commonParam")
@ResponseBody
public String commonParam(String name ,int age){
    System.out.println("普通参数传递 name ==> "+name);
    System.out.println("普通参数传递 age ==> "+age);
    return "{\"module':'common param'}";
}
```

请求参数

- 普通参数：请求参数名与形参变量名不同，使用@RequestParam绑定参数关系

GET http://localhost/commonParamDifferentName?name=itcast&age=15

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE
<input checked="" type="checkbox"/> name	itcast
<input checked="" type="checkbox"/> age	15
Key	Value

```
@RequestMapping("/commonParamDifferentName")
@ResponseBody
public String commonParamDifferentName(@RequestParam("name")String userName , int age){
    System.out.println("普通参数传递 userName ==> "+userName);
    System.out.println("普通参数传递 age ==> "+age);
    return "{ 'module': 'common param different name'}";
}
```

请求参数

- 普通参数：url地址传参，地址参数名与形参变量名相同，定义形参即可接收参数

GET http://localhost/commonParam?name=itcast&age=15

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE
<input checked="" type="checkbox"/> name	itcast
<input checked="" type="checkbox"/> age	15
Key	Value

```
@RequestMapping("/commonParam")
@ResponseBody
public String commonParam(String name ,int age){
    System.out.println("普通参数传递 name ==> "+name);
    System.out.println("普通参数传递 age ==> "+age);
    return "{ 'module': 'common param'}";
}
```

请求参数

- 名称：@RequestParam
- 类型：形参注解
- 位置：SpringMVC控制器方法形参定义前面
- 作用：绑定请求参数与处理器方法形参间的关系
- 范例：

```
@RequestMapping("/commonParamDifferentName")
@ResponseBody
public String commonParamDifferentName(@RequestParam("name")String userName , int age){
    System.out.println("普通参数传递 userName ==> "+userName);
    System.out.println("普通参数传递 age ==> "+age);
    return "{ 'module': 'common param different name'}";
}
```

- 参数：
 - required：是否为必传参数
 - defaultValue：参数默认值

请求参数

- POJO参数：请求参数名与形参对象属性名相同，定义POJO类型形参即可接收参数

```
@RequestMapping("/pojoParam")
@ResponseBody
public String pojoParam(User user){
    System.out.println("pojo参数传递 user ==> "+user);
    return "{\"module\":\"pojo param\"}";
}
```

- 嵌套POJO参数：POJO对象中包含POJO对象

```
public class User {
    private String name;
    private int age;
    private Address address;
}

public class Address {
    private String province;
    private String city;
}
```

- 嵌套POJO参数：请求参数名与形参对象属性名相同，按照对象层次结构关系即可接收嵌套POJO属性参数

```
@RequestMapping("/pojoContainPojoParam")
@ResponseBody
public String pojoContainPojoParam(User user){
    System.out.println("pojo嵌套pojo参数传递 user ==> "+user);
    return "{\"module\":\"pojo contain pojo param\"}";
}
```

- 集合保存普通参数：请求参数名与形参集合对象名相同且请求参数为多个，`@RequestParam`绑定参数关系

KEY	VALUE
<input checked="" type="checkbox"/> likes	game
<input checked="" type="checkbox"/> likes	music
<input checked="" type="checkbox"/> likes	travel
Key	Value

```

@RequestMapping("/listParam")
@ResponseBody
public String listParam(@RequestParam List<String> likes){
    System.out.println("集合参数传递 likes ==> " + likes);
    return "{\"module\":\"list param\"}";
}

```



1. Url传递参数
2. `@RequestParam`

9、json数据传递参数

请求参数（传递json数据）

- json数组
- json对象（POJO）
- json数组（POJO）

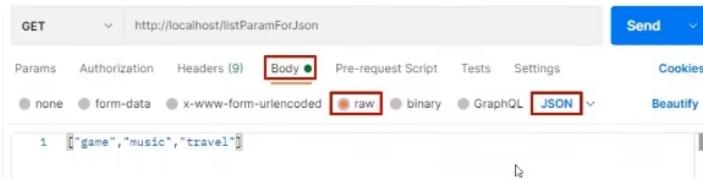
步骤 接收请求中的json数据

①：添加json数据转换相关坐标

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.9.0</version>
</dependency>
```

步骤 接收请求中的json数据

②：设置发送json数据（请求body中添加json数据）



③：开启自动转换json数据的支持

```
@Configuration
@ComponentScan("com.itheima.controller")
@EnableWebMvc
public class SpringMvcConfig { }
```

注意事项

`@EnableWebMvc`注解功能强大，该注解整合了多个功能，此处仅使用其中一部分功能，即json数据进行自动类型转换

④：设置接收json数据

```
@RequestMapping("/listParamForJson")
@ResponseBody
public String listParamForJson(@RequestBody List<String> likes){
    System.out.println("list common(json)参数传递 list ==> "+likes);
    return "{\"module\":\"list common for json param\"}";
}
```

- 名称: @EnableWebMvc
- 类型: 配置类注解
- 位置: SpringMVC配置类定义上方
- 作用: 开启SpringMVC多项辅助功能
- 范例:

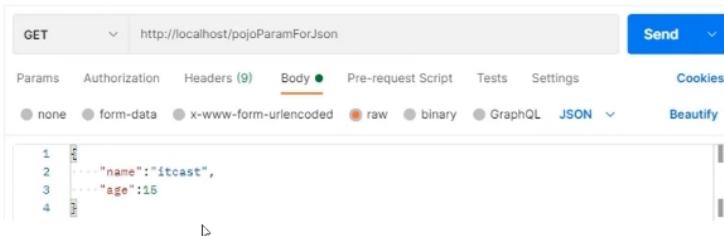
```
@Configuration
@ComponentScan("com.itheima.controller")
@EnableWebMvc
public class SpringMvcConfig {
}
```

- 名称: @RequestBody
- 类型: 形参注解
- 位置: SpringMVC控制器方法形参定义前面
- 作用: 将请求中请求体所包含的数据传递给请求参数, 此注解一个处理器方法只能使用一次
- 范例:

```
@RequestMapping("/listParamForJson")
@ResponseBody
public String listParamForJson(@RequestBody List<String> likes){
    System.out.println("list common(json)参数传递 list ==> "+likes);
    return "{\"module':'list common for json param'}";
}
```

请求参数 (传递json对象)

- POJO参数: json数据与形参对象属性名相同, 定义POJO类型形参即可接收参数



```
@RequestMapping("/pojoParamForJson")
@ResponseBody
public String pojoParamForJson(@RequestBody User user){
    System.out.println("pojo(json)参数传递 user ==> "+user);
    return "{\"module':'pojo for json param'}";
}
```

请求参数（传递json数组）

- POJO集合参数：json数组数据与集合泛型属性名相同，定义List类型形参即可接收参数

```
@RequestMapping("/listPojoParamForJson")
@ResponseBody
public String listPojoParamForJson(@RequestBody List<User> list){
    System.out.println("list pojo(json)参数传递 list ==> " + list);
    return "{ 'module': 'list pojo for json param' }";
}
```

@RequestBody与@RequestParam区别

● 区别

- @RequestParam用于接收url地址传参，表单传参【application/x-www-form-urlencoded】
- @RequestBody用于接收json数据【application/json】

● 应用

- 后期开发中，发送json格式数据为主，@RequestBody应用较广
- 如果发送非json格式数据，选用@RequestParam接收请求参数



1. json数据传递与接收 **(常用)**
2. @EnableWebMvc
3. @RequestBody

10、日期型参数传递

日期类型参数传递

- 日期类型数据基于系统不同格式也不尽相同
 - 2088-08-18
 - 2088/08/18
 - 08/18/2088

- 日期类型数据基于系统不同格式也不尽相同

- 2088-08-18
- 2088/08/18
- 08/18/2088

- 接收形参时，根据不同的日期格式设置不同的接收方式

```
@RequestMapping("/dataParam")
@ResponseBody
public String dataParam(Date date,
    @DateTimeFormat(pattern = "yyyy-MM-dd") Date date1,
    @DateTimeFormat(pattern = "yyyy/MM/dd HH:mm:ss") Date date2){
    System.out.println("参数传递 date ==> "+date);
    System.out.println("参数传递 date(yyyy-MM-dd) ==> "+date1);
    System.out.println("参数传递 date(yyyy/MM/dd HH:mm:ss) ==> "+date2);
    return "{ 'module': 'data param'}";
}
```

```
http://localhost/dataParam?date=2088/08/08&date1=2088-08-18&date2=2088/08/28 8:08:08
```

日期类型参数传递

- 名称: @DateTimeFormat
- 类型: 形参注解
- 位置: SpringMVC控制器方法形参前面
- 作用: 设定日期时间型数据格式
- 范例:

```
@RequestMapping("/dataParam")
@ResponseBody
public String dataParam(Date date){
    System.out.println("参数传递 date ==> "+date);
    return "{ 'module': 'data param'}";
}
```

- 属性: pattern: 日期时间格式字符串

类型转换器

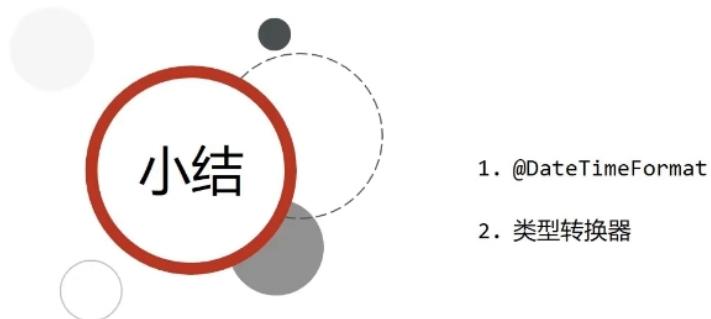
- Converter接口

```
public interface Converter<S, T> {  
    @Nullable  
    T convert(S var1);  
}
```

■ 请求参数年龄数据 (String→Integer)

■ 日期格式转换 (String → Date)

- @EnableWebMvc功能之一：根据类型匹配对应的类型转换器



1. @DateTimeFormat
2. 类型转换器

11、响应

响应

- 响应页面
- 响应数据
 - 文本数据
 - json数据

- 响应页面（了解）

```
@RequestMapping("/toPage")
public String toPage(){
    return "page.jsp";
}
```

- 响应文本数据（了解）

```
@RequestMapping("/toText")
@ResponseBody
public String toText(){
    return "response text";
}
```

- 响应json数据（对象转json）

```
@RequestMapping("/toJsonPOJO")
@ResponseBody
public User toJsonPOJO(){
    User user = new User();
    user.setName("赵云");
    user.setAge(41);
    return user;
}
```

- 响应json数据（对象集合转json数组）

```
@RequestMapping("/toJsonList")
@ResponseBody
public List<User> toJsonList(){
    User user1 = new User();
    user1.setName("赵云");
    user1.setAge(41);
    User user2 = new User();
    user2.setName("master 赵云");
    user2.setAge(40);
    List<User> userList = new ArrayList<User>();
    userList.add(user1);
    userList.add(user2);
    return userList;
}
```

响应

- 名称: @ResponseBody
- 类型: **方法注解**
- 位置: SpringMVC控制器方法定义上方
- 作用: 设置当前控制器方法响应内容为当前返回值, 无需解析
- 范例:

```
@RequestMapping("/save")
@ResponseBody
public String save(){
    System.out.println("save...");
    return "{\"info\":\"springmvc\"}";
}
```

响应

- 名称: @ResponseBody
- 类型: **方法注解**
- 位置: SpringMVC控制器方法定义上方
- 作用: 设置当前控制器返回值作为响应体
- 范例:

```
@RequestMapping("/save")
@ResponseBody
public String save(){
    System.out.println("save...");
    return "{\"info\":\"springmvc\"}";
}
```



1. 响应
2. @ResponseBody
3. 类型转换器(HttpMessageConverter)

12、REST风格简介

REST简介

- **REST** (Representational State Transfer) , 表现形式状态转换
 - 传统风格资源描述形式

```
http://localhost/user/getById?id=1
```

```
http://localhost/user/saveUser
```
 - REST风格描述形式

```
http://localhost/user/1
```

```
http://localhost/user
```
- 优点:
 - 隐藏资源的访问行为, 无法通过地址得知对资源是何种操作
 - 书写简化

REST风格简介

- 按照REST风格访问资源时使用**行为动作**区分对资源进行了何种操作
 - `http://localhost/users` 查询全部用户信息 GET (查询)
 - `http://localhost/users/1` 查询指定用户信息 GET (查询)
 - `http://localhost/users` 添加用户信息 POST (新增/保存)
 - `http://localhost/users` 修改用户信息 PUT (修改/更新)
 - `http://localhost/users/1` 删除用户信息 DELETE (删除)

REST风格简介

- 按照REST风格访问资源时使用**行为动作**区分对资源进行了何种操作
 - `http://localhost/users` 查询全部用户信息 GET (查询)
 - `http://localhost/users/1` 查询指定用户信息 GET (查询)
 - `http://localhost/users` 添加用户信息 POST (新增/保存)
 - `http://localhost/users` 修改用户信息 PUT (修改/更新)
 - `http://localhost/users/1` 删除用户信息 DELETE (删除)
- 根据REST风格对资源进行访问称为**RESTful**



注意事项

上述行为是约定方式，约定不是规范，可以打破，所以称REST风格，而不是REST规范

描述模块的名称通常使用复数，也就是加s的格式描述，表示此类资源，而非单个资源，例如：users、books、accounts.....

小结

1. REST

2. 动作4个

3. RESTful

13、RESTful入门案例

①: 设定http请求动作 (动词)

```
@RequestMapping(value = "/users", method = RequestMethod.POST)
@ResponseBody
public String save(@RequestBody User user){
    System.out.println("user save..." + user);
    return "{\"module':'user save'}";
}

@RequestMapping(value = "/users" ,method = RequestMethod.PUT)
@ResponseBody
public String update(@RequestBody User user){
    System.out.println("user update..."+user);
    return "{\"module':'user update'}";
}
```

②：设定请求参数（路径变量）

```
@RequestMapping(value = "/users/{id}" ,method = RequestMethod.DELETE)
@ResponseBody
public String delete(@PathVariable Integer id){
    System.out.println("user delete..." + id);
    return "{module}:'user delete'";
}
```

入门案例

- 名称: `@RequestMapping`
- 类型: **方法注解**
- 位置: SpringMVC控制器方法定义上方
- 作用: 设置当前控制器方法请求访问路径
- 范例:

```
@RequestMapping(value = "/users", method = RequestMethod.POST)
@ResponseBody
public String save(@RequestBody User user){
    System.out.println("user save..." + user);
    return "{module}:'user save'";
}
```

- 属性
 - `value` (默认) : 请求访问路径
 - `method`: http请求动作, 标准动作 (GET/POST/PUT/DELETE)

`@RequestBody @RequestParam @PathVariable`

- 区别
 - `@RequestParam`用于接收url地址传参或表单传参
 - `@RequestBody`用于接收json数据
 - `@PathVariable`用于接收路径参数, 使用{参数名称}描述路径参数
- 应用
 - 后期开发中, 发送请求参数超过1个时, 以json格式为主, `@RequestBody`应用较广
 - 如果发送非json格式数据, 选用`@RequestParam`接收请求参数
 - 采用RESTful进行开发, 当参数数量较少时, 例如1个, 可以采用`@PathVariable`接收请求路径变量, 通常用于传递id值



1. 入门案例
2. 请求方法设定
3. 请求路径参数



14、RESTful快速开发

RESTful快速开发

```
@RequestMapping(value = "/books", method = RequestMethod.POST)
@ResponseBody
public String save(@RequestBody Book book){
    System.out.println("book save..." + book);
    return "{\"module\":\"book save\"}";
}

@RequestMapping(value = "/books" ,method = RequestMethod.PUT)
@ResponseBody
public String update(@RequestBody Book book){
    System.out.println("book update..." + book);
    return "{\"module\":\"book update\"}";
}
```

RESTful快速开发

- 名称: `@RestController`
- 类型: **类注解**
- 位置: 基于SpringMVC的RESTful开发控制器类定义上方
- 作用: 设置当前控制器类为RESTful风格, 等同于`@Controller`与`@ResponseBody`两个注解组合功能
- 范例:

```
@RestController  
public class BookController {  
}
```

RESTful快速开发

- 名称: `@GetMapping` `@PostMapping` `@PutMapping` `@DeleteMapping`
- 类型: **方法注解**
- 位置: 基于SpringMVC的RESTful开发控制器方法定义上方
- 作用: 设置当前控制器方法请求访问路径与请求动作, 每种对应一个请求动作, 例如`@GetMapping`对应GET请求
- 范例:

```
@GetMapping("/{id}")  
public String getById(@PathVariable Integer id){  
    System.out.println("book getById..."+id);  
    return "{ 'module': 'book getById'}";  
}
```

- 属性

- `value` (默认) : 请求访问路径



1. RESTful快速开发 (标准开发)
2. `@RestController`
3. 标准请求动作映射 (4种)

15、案例：基于RESTful页面数据交互（后台接口开发）

不说了直接上控制类

```
package com.itheima.controller;

import com.itheima.domain.Book;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.List;

@RestController
@RequestMapping("/books")
public class BookController {

    @PostMapping
    public String save(@RequestBody Book book){
        System.out.println("book save ==> "+ book);
        return "{\"module\":\"book save success\"}";
    }

    @GetMapping
    public List<Book> getAll(){
        System.out.println("book getAll is running ...");
        List<Book> bookList = new ArrayList<Book>();

        Book book1 = new Book();
        book1.setType("计算机");
        book1.setName("SpringMVC入门教程");
        book1.setDescription("小试牛刀");
        bookList.add(book1);

        Book book2 = new Book();
        book2.setType("计算机");
        book2.setName("SpringMVC实战教程");
        book2.setDescription("一代宗师");
        bookList.add(book2);

        Book book3 = new Book();
        book3.setType("计算机丛书");
        book3.setName("SpringMVC实战教程进阶");
        book3.setDescription("一代宗师呕心创作");
        bookList.add(book3);

        return bookList;
    }
}
```

16、案例：基于RESTful页面数据交互（页面访问处理）

①：制作SpringMVC控制器，并通过PostMan测试接口功能

```
@RestController
@RequestMapping("/books")
public class BookController {
    @PostMapping
    public String save(@RequestBody Book book){
        System.out.println("book save ==> " + book);
        return "{ 'module': 'book save success' }";
    }
    @GetMapping
    public List<Book> getAll(){
        System.out.println("book getAll is running ...");
        List<Book> bookList = new ArrayList<Book>();
        Book book1 = new Book();
        book1.setType("计算机");
        book1.setName("SpringMVC入门教程");
        book1.setDescription("小试牛刀");
        bookList.add(book1);
        //模拟数据...
        return bookList;
    }
}
```

②：设置对静态资源的访问放行

```
@Configuration
public class SpringMvcSupport extends WebMvcConfigurerSupport {
    @Override
    protected void addResourceHandlers(ResourceHandlerRegistry registry) {
        //当访问/pages/????时候，走/pages目录下的内容
        registry.addResourceHandler("/pages/**").addResourceLocations("/pages/");
        registry.addResourceHandler("/js/**").addResourceLocations("/js/");
        registry.addResourceHandler("/css/**").addResourceLocations("/css/");
        registry.addResourceHandler("/plugins/**").addResourceLocations("/plugins/");
    }
}
```

③：前端页面通过异步提交访问后台控制器

```
//添加
saveBook () {
    axios.post("/books",this.formData).then((res)=>{
        });
},
//主页列表查询
getAll() {
    axios.get("/books").then((res)=>{
        this.dataList = res.data;
    });
},
```

小结

1. 案例：基于RESTful页面数据交互

- 先做后台功能，开发接口并调通接口
- 再做页面异步调用，确认功能可以正常访问
- 最后完成页面数据展示
- 补充：放行静态资源访问

总结

1. REST风格简介

2. RESTful入门案例

3. RESTful快速开发

4. 案例：基于RESTful页面数据交互

17、SSM整合（整合配置）

SSM整合流程

1. 创建工程
2. SSM整合 
- Spring
 - SpringConfig
- MyBatis
 - MybatisConfig
 - JdbcConfig
 - jdbc.properties
- SpringMVC
 - ServletConfig
 - SpringMvcConfig
3. 功能模块
 - 表与实体类
 - dao (接口+自动代理)
 - service (接口+实现类)
 - 业务层接口测试 (整合JUnit)
 - controller
 - 表现层接口测试 (PostMan)

感觉springboot把配置简化好多

18、SSM整合（功能模块开发）

SSM整合流程

1. 创建工程
2. SSM整合 
- Spring
 - SpringConfig
- MyBatis
 - MybatisConfig
 - JdbcConfig
 - jdbc.properties
- SpringMVC
 - ServletConfig
 - SpringMvcConfig
3. 功能模块
 - 表与实体类
 - dao (接口+自动代理)
 - service (接口+实现类)
 - 业务层接口测试 (整合JUnit)
 - controller
 - 表现层接口测试 (PostMan)

参考项目

19、SSM整合（接口测试）

记得接事务

Spring整合MyBatis

- 配置
 - SpringConfig
 - JDBCConfig、jdbc.properties
 - MyBatisConfig
 - 模型
 - Book
 - 数据层标准开发
 - BookDao
 - 业务层标准开发
 - BookService
 - BookServiceImpl
 - 测试接口
 - BookServiceTest
- ```
@Configuration
@ComponentScan("com.itheima")
@PropertySource("classpath:jdbc.properties")
@Import({JdbcConfig.class,MybatisConfig.class})
public class SpringConfig {
}
```

## Spring整合MyBatis

- 配置
  - SpringConfig
  - JDBCConfig、jdbc.properties
  - MyBatisConfig
- 模型
  - Book
- 数据层标准开发
  - BookDao
- 业务层标准开发
  - BookService
  - BookServiceImpl
- 测试接口
  - BookServiceTest

```
public class JdbcConfig {
 @Value("${jdbc.driver}")
 private String driver;
 @Value("${jdbc.url}")
 private String url;
 @Value("${jdbc.username}")
 private String userName;
 @Value("${jdbc.password}")
 private String password;
 @Bean
 public DataSource dataSource(){
 DruidDataSource ds = new DruidDataSource();
 ds.setDriverClassName(driver);
 ds.setUrl(url);
 ds.setUsername(userName);
 ds.setPassword(password);
 return ds;
 }
}
```

## Spring整合MyBatis

- 配置
  - SpringConfig
  - JDBCConfig、jdbc.properties
  - MyBatisConfig
- 模型
  - Book
- 数据层标准开发
  - BookDao
- 业务层标准开发
  - BookService
  - BookServiceImpl
- 测试接口
  - BookServiceTest

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/ssm_db?useSSL=false
jdbc.username=root
jdbc.password=root
```

## Spring整合MyBatis

- 配置
  - SpringConfig
  - JDBCConfig、jdbc.properties
  - MyBatisConfig
- 模型
  - Book
- 数据层标准开发
  - BookDao
- 业务层标准开发
  - BookService
  - BookServiceImpl
- 测试接口
  - BookServiceTest

```
public class MybatisConfig {
 @Bean
 public SqlSessionFactoryBean sqlSessionFactory(DataSource dataSource){
 SqlSessionFactoryBean ssfb = new SqlSessionFactoryBean();
 ssfb.setTypeAliasesPackage("com.itheima.domain");
 ssfb.setDataSource(dataSource);
 return ssfb;
 }
 @Bean
 public MapperScannerConfigurer mapperScannerConfigurer(){
 MapperScannerConfigurer msc = new MapperScannerConfigurer();
 msc.setBasePackage("com.itheima.dao");
 return msc;
 }
}
```

## Spring整合MyBatis

- 配置
  - SpringConfig
  - JDBCConfig、jdbc.properties
  - MyBatisConfig
- 模型
  - Book
- 数据层标准开发
  - BookDao
- 业务层标准开发
  - BookService
  - BookServiceImpl
- 测试接口
  - BookServiceTest

```
public class Book {
 private Integer id;
 private String type;
 private String name;
 private String description;
}
```

## Spring整合MyBatis

- 配置
  - SpringConfig
  - JDBCConfig、jdbc.properties
  - MyBatisConfig
- 模型
  - Book
- 数据层标准开发
  - BookDao
- 业务层标准开发
  - BookService
  - BookServiceImpl
- 测试接口
  - BookServiceTest

```
public interface BookService {
 void save(Book book);
 void delete(Integer id);
 void update(Book book);
 List<Book> getAll();

```

## Spring整合MyBatis

- 配置
  - SpringConfig
  - JDBCConfig、jdbc.properties
  - MyBatisConfig
- 模型
  - Book
- 数据层标准开发
  - BookDao
- 业务层标准开发
  - BookService
  - BookServiceImpl
- 测试接口
  - BookServiceTest

```
@Service
public class BookServiceImpl implements BookService {
 @Autowired
 private BookDao bookDao;
 public void save(Book book) { bookDao.save(book); }

```

## Spring整合MyBatis

- 配置
  - SpringConfig
  - JDBCConfig、jdbc.properties
  - MyBatisConfig
- 模型
  - Book
- 数据层标准开发
  - BookDao
- 业务层标准开发
  - BookService
  - BookServiceImpl
- 测试接口
  - BookServiceTest

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = SpringConfig.class)
public class BookServiceTest {
 @Autowired
 private BookService bookService;
 @Test
 public void testGetdById(){
 System.out.println(bookService.getById(1));
 }
 @Test
 public void testGetAll(){
 System.out.println(bookService.getAll());
 }
}
```

## Spring整合MyBatis

- 事务处理

```
public class JdbcConfig {
 @Bean
 public PlatformTransactionManager transactionManager(DataSource dataSource){
 DataSourceTransactionManager transactionManager = new DataSourceTransactionManager(dataSource);
 return transactionManager;
 }
}

@Configuration
@ComponentScan("com.itheima")
@PropertySource("classpath:jdbc.properties")
@Import({JdbcConfig.class,MybatisConfig.class})
@EnableTransactionManagement
public class SpringConfig {

}

@Transactional
public interface BookService {
```

## Spring整合SpringMVC

- web配置类

```
public class ServletContainersInitConfig extends AbstractAnnotationConfigDispatcherServletInitializer {
 protected Class<?>[] getRootConfigClasses() {
 return new Class[]{SpringConfig.class};
 }
 protected Class<?>[] getServletConfigClasses() {
 return new Class[]{SpringMvcConfig.class};
 }
 protected String[] getServletMappings() {
 return new String[]{"/"};
 }
 //乱码处理
 @Override
 protected Filter[] getServletFilters() {
 CharacterEncodingFilter filter = new CharacterEncodingFilter();
 filter.setEncoding("UTF-8");
 return new Filter[]{filter};
 }
}
```

## Spring整合SpringMVC

- SpringMVC配置类

```
@Configuration
@ComponentScan({"com.itheima.controller"})
@EnableWebMvc
public class SpringMvcConfig {
}
```

## Spring整合SpringMVC

- 基于Restful的Controller开发

```
@RestController
@RequestMapping("/books")
public class BookController {
 @Autowired
 private BookService bookService;
 @PostMapping
 public void save(@RequestBody Book book) {
 bookService.save(book);
 }
}
```

## Spring整合SpringMVC

- 基于Restful的Controller开发

```
@RestController
@RequestMapping("/books")
public class BookController {
 @GetMapping("/{id}")
 public Book getById(@PathVariable Integer id) {
 return bookService.getById(id);
 }
 @GetMapping
 public List<Book> getAll() {
 return bookService.getAll();
 }
}
```



## 1. SSM整合

- Spring配置
- Spring整合MyBatis
- Spring整合SpringMVC
- RESTful标准控制器开发

# 20、SSM整合-表现层与前端数据传输协议定义

## 表现层数据封装

- 前端接收数据格式

- 增删改

true

- 查单条

```
{
 "id": 1,
 "type": "计算机理论",
 "name": "Spring实战 第5版",
 "description": "Spring入门经典教程"
}
```

- 查全部

```
[
 {
 "id": 1,
 "type": "计算机理论",
 "name": "Spring实战 第5版",
 "description": "Spring入门经典教程"
 },
 {
 "id": 2,
 "type": "计算机理论",
 "name": "Spring 5核心原理与30个类手写实战",
 "description": "十年沉淀之作"
 }
]
```

## 表现层数据封装

- 前端接收数据格式

- 增删改

true

格式A

- 查单条

格式B

```
{
 "id": 1,
 "type": "计算机理论",
 "name": "Spring实战 第5版",
 "description": "Spring入门经典教程"
}
```

格式D

格式E

格式G

格式F

- 查全部

格式C

```
[
 {
 "id": 1,
 "type": "计算机理论",
 "name": "Spring实战 第5版",
 "description": "Spring入门经典教程"
 },
 {
 "id": 2,
 "type": "计算机理论",
 "name": "Spring 5核心原理与30个类手写实战",
 "description": "十年沉淀之作"
 }
]
```

## 表现层数据封装

- 前端接收数据格式

- 增删改

```
true
```

- 查单条

```
{
 "id": 1,
 "type": "计算机理论",
 "name": "Spring实战 第5版",
 "description": "Spring入门经典教程"
}
```

- 查全部

```
[
 {
 "id": 1,
 "type": "计算机理论",
 "name": "Spring实战 第5版",
 "description": "Spring入门经典教程"
 },
 {
 "id": 2,
 "type": "计算机理论",
 "name": "Spring 5核心原理与30个类手写实战",
 "description": "十年沉淀之作"
 }
]
```

# 统一格式

## 表现层数据封装

- 前端接收数据格式—创建结果模型类，封装数据到data属性中

- 增删改

```
{
 "data":true
}
```

- 查单条

```
{
 "data":{
 "id": 1,
 "type": "计算机理论",
 "name": "Spring实战 第5版",
 "description": "Spring入门经典教程"
 }
}
```

- 查全部

```
{
 "data": [
 {
 "id": 1,
 "type": "计算机理论",
 "name": "Spring实战 第5版",
 "description": "Spring入门经典教程"
 },
 {
 "id": 2,
 "type": "计算机理论",
 "name": "Spring 5核心原理与30个类手写实战",
 "description": "十年沉淀之作"
 }
]
}
```

## 表现层数据封装

- 前端接收数据格式—创建结果模型类，封装数据到data属性中

- 增删改

```
{
 "data":true
}
```

- 查单条

```
{
 "data":{
 "id": 1,
 "type": "计算机理论",
 "name": "Spring实战 第5版",
 "description": "Spring入门经典教程"
 }
}
```

- 查全部

```
{
 "data": [
 {
 "id": 1,
 "type": "计算机理论",
 "name": "Spring实战 第5版",
 "description": "Spring入门经典教程"
 },
 {
 "id": 2,
 "type": "计算机理论",
 "name": "Spring 5核心原理与30个类手写实战",
 "description": "十年沉淀之作"
 }
]
}
```

新增

修改

删除

## 表现层数据封装

- 前端接收数据格式—封装操作结果到code属性中

- 增删改

```
{
 "code": 20031,
 "data": true
}
```

- 查单条

```
{
 "code": 20041,
 "data": null
}
```

- 查全部

```
{
 "code": 20041,
 "data": [
 {
 "id": 1,
 "type": "计算机理论",
 "name": "Spring实战 第5版",
 "description": "Spring入门经典教程"
 },
 {
 "id": 2,
 "type": "计算机理论",
 "name": "Spring 5核心原理与30个类手写实战",
 "description": "十年沉淀之作"
 }
]
}
```

查询不存在的数据

```
{
 "code": 20041,
 "data": null
}
```

20041表示成功/失败？

## 表现层数据封装

- 前端接收数据格式—封装特殊消息到message(msg)属性中

- 增删改

```
{
 "code": 20031,
 "data": true
}
```

- 查单条

```
{
 "code": 20040,
 "data": null,
 "msg": "数据查询失败，请重试！"
}
```

用户得到什么信息？

- 查全部

```
{
 "code": 20041,
 "data": [
 {
 "id": 1,
 "type": "计算机理论",
 "name": "Spring实战 第5版",
 "description": "Spring入门经典教程"
 },
 {
 "id": 2,
 "type": "计算机理论",
 "name": "Spring 5核心原理与30个类手写实战",
 "description": "十年沉淀之作"
 }
]
}
```

## 表现层数据封装

- 设置统一数据返回结果类

```
public class Result {
 private Object data;
 private Integer code;
 private String msg;
}
```

### 注意事项

Result类中的字段并不是固定的，可以根据需要自行增减  
提供若干个构造方法，方便操作

# 21、SSM整合-表现层与前端数据传输数据协议实现

## 表现层数据封装

- 设置统一数据返回结果编码

```
public class Code {
 public static final Integer SAVE_OK = 20011;
 public static final Integer DELETE_OK = 20021;
 public static final Integer UPDATE_OK = 20031;
 public static final Integer GET_OK = 20041;

 public static final Integer SAVE_ERR = 20010;
 public static final Integer DELETE_ERR = 20020;
 public static final Integer UPDATE_ERR = 20030;
 public static final Integer GET_ERR = 20040;
}
```

### 注意事项

Code类的常量设计也不是固定的，可以根据需要自行增减，例如将查询再进行细分为GET\_OK, GET\_ALL\_OK, GET\_PAGE\_OK

## 表现层数据封装

- 根据情况设定合理的Result

```
@RequestMapping("/books")
public class BookController {
 @Autowired
 private BookService bookService;
 @GetMapping("/{id}")
 public Result getById(@PathVariable Integer id) {
 Book book = bookService.getById(id);
 Integer code = book != null ? Code.GET_OK : Code.GET_ERR;
 String msg = book != null ? "" : "数据查询失败，请重试！";
 return new Result(code, book, msg);
 }
}
```



1. 表现层数据封装

2. Result

3. Code

## 22、SSM整合-异常处理器

### 异常处理器

- 程序开发过程中不可避免的会遇到异常现象

```

GET http://localhost/books/1
{
 "data": [
 {
 "id": 1,
 "type": "计算机理论",
 "name": "Spring实战 第5版",
 "description": "Spring入门经典教程，深入理解Spring框架技术内幕",
 "code": 20041,
 "msg": null
 }
]
}

```

```

HTTP Status 500 - Request processing failed; nested exception is java.lang.ArithmeticException: / by zero
type Exception report
message Request processing failed; nested exception is java.lang.ArithmeticException: / by zero
description The server encountered an internal error that prevented it from fulfilling this request.
exception
java.lang.ArithmeticException: / by zero
 at org.springframework.web.util.NestedServletException.fillInStackTrace(NestedServletException.java:101)
 at org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:1014)
 at org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:890)
 at javax.servlet.http.HttpServlet.service(HttpServlet.java:621)
 at org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:883)
 at javax.servlet.http.HttpServlet.service(HttpServlet.java:729)

```

- 出现异常现象的常见位置与常见诱因如下：

- 框架内部抛出的异常：因使用不合规导致
- 数据层抛出的异常：因外部服务器故障导致（例如：服务器访问超时）
- 业务层抛出的异常：因业务逻辑书写错误导致（例如：遍历业务书写操作，导致索引异常等）
- 表现层抛出的异常：因数据收集、校验等规则导致（例如：不匹配的数据类型间导致异常）
- 工具类抛出的异常：因工具类书写不严谨不够健壮导致（例如：必要释放的连接长期未释放等）



1. 各个层级均出现异常，异常处理代码书写在哪一层  
?



1. 各个层级均出现异常，异常处理代码书写在哪一层  
—所有的异常均抛出到表现层进行处理



1. 表现层处理异常，每个方法中单独书写，代码书写量巨大且意义不强，如何解决？



1. 表现层处理异常，每个方法中单独书写，代码书写量巨大且意义不强，如何解决—**AOP思想**

- 异常处理器

- 集中的、统一的处理项目中出现的异常

```
@RestControllerAdvice
public class ProjectExceptionAdvice {
 @ExceptionHandler(Exception.class)
 public Result doException(Exception ex){
 return new Result(666,null);
 }
}
```

- 名称: `@ExceptionHandler`
- 类型: **方法注解**
- 位置: 专用于异常处理的控制器方法上方
- 作用: 设置指定异常的处理方案, 功能等同于控制器方法, 出现异常后终止原始控制器执行, 并转入当前方法执行
- 范例:

```
@RestControllerAdvice
public class ProjectExceptionAdvice {
 @ExceptionHandler(Exception.class)
 public Result doException(Exception ex){
 return new Result(666,null);
 }
}
```

- 说明
  - 此类方法可以根据处理的异常不同, 制作多个方法分别处理对应的异常

- 名称: `@RestControllerAdvice`
- 类型: **类注解**
- 位置: Rest风格开发的控制器增强类定义上方
- 作用: 为Rest风格开发的控制器类做增强
- 范例:

```
@RestControllerAdvice
public class ProjectExceptionAdvice {
```

- 说明:
  - 此注解自带`@ResponseBody`注解与`@Component`注解, 具备对应的功能

## 异常处理器

- 异常处理器处理效果比对

The screenshot shows two side-by-side Postman requests to the URL `http://localhost/books/1`. Both requests are GET methods. The left request has a complex JSON response body containing multiple fields: `data`, `type`, `name`, `description`, `code`, and `msg`. The right request has a simplified response body with only `code` and `msg` fields.

## 23、SSM整合-项目异常处理

### 项目异常处理方案

- 项目异常分类

The screenshot shows three requests in Postman:

- Request 1: GET `http://localhost/books/1` - Shows a normal JSON response.
- Request 2: GET `http://localhost/books/heihai` - Shows an error response with the message "不规范的用户行为操作产生的异常".
- Request 3: POST `http://localhost/users` - Shows an error response with the message "规范的用户行为产生的异常".

In the third request's body, the `age` field is highlighted in red, indicating it is the source of the validation error.

- 项目异常分类



项目运行过程中可预计且无法避免的异常

```
java.io.FileNotFoundException: log.data1 (系统找不到指定的文件。)
at java.io.FileInputStream.open0(Native Method)
at java.io.FileInputStream.open(FileInputStream.java:195)
at java.io.FileInputStream.<init>(FileInputStream.java:138)
```

编程人员未预期到的异常

- 项目异常分类

- 业务异常 (BusinessException)
  - ◆ 规范的用户行为产生的异常
  - ◆ 不规范的用户行为操作产生的异常
- 系统异常 (SystemException)
  - ◆ 项目运行过程中可预计且无法避免的异常
- 其他异常 (Exception)
  - ◆ 编程人员未预期到的异常

- 项目异常处理方案

- 业务异常 (BusinessException)
  - ◆ 发送对应消息传递给用户，提醒规范操作
- 系统异常 (SystemException)
  - ◆ 发送固定消息传递给用户，安抚用户
  - ◆ 发送特定消息给运维人员，提醒维护
  - ◆ 记录日志
- 其他异常 (Exception)
  - ◆ 发送固定消息传递给用户，安抚用户
  - ◆ 发送特定消息给编程人员，提醒维护（纳入预期范围内）
  - ◆ 记录日志

①：自定义项目系统级异常

```
public class SystemException extends RuntimeException{
 private Integer code;
 public SystemException(Integer code, String message) {
 super(message);
 this.code = code;
 }
 public SystemException(Integer code, String message, Throwable cause) {
 super(message, cause);
 this.code = code;
 }
 public Integer getCode() {
 return code;
 }
 public void setCode(Integer code) {
 this.code = code;
 }
}
```

②：自定义项目业务级异常

```
public class BusinessException extends RuntimeException{
 private Integer code;
 public BusinessException(Integer code, String message) {
 super(message);
 this.code = code;
 }
 public BusinessException(Integer code, String message, Throwable cause) {
 super(message, cause);
 this.code = code;
 }
 public Integer getCode() {
 return code;
 }
 public void setCode(Integer code) {
 this.code = code;
 }
}
```

③：自定义异常编码（持续补充）

```
public class Code {
 public static final Integer SYSTEM_UNKNOW_ERROR = 50001;
 public static final Integer SYSTEM_TIMEOUT_ERROR = 50002;

 public static final Integer PROJECT_VALIDATE_ERROR = 60001;
 public static final Integer PROJECT_BUSINESS_ERROR = 60002;
}
```

#### ④：触发自定义异常

```
@Service
public class BookServiceImpl implements BookService {
 @Autowired
 private BookDao bookDao;
 public Book getById(Integer id) {
 if(id < 0){
 throw new BusinessException(Code.PROJECT_BUSINESS_ERROR,"请勿进行非法操作！");
 }
 return bookDao.getById(id);
 }
}
```

#### ⑤：拦截并处理异常

```
@RestControllerAdvice
public class ProjectExceptionAdvice {
 @ExceptionHandler(BusinessException.class)
 public Result doBusinessException(BusinessException ex){
 return new Result(ex.getCode(),null,ex.getMessage());
 }
 @ExceptionHandler(SystemException.class)
 public Result doSystemException(SystemException ex){
 // 记录日志（错误堆栈）
 // 发送邮件给开发人员
 // 发送短信给运维人员
 return new Result(ex.getCode(),null,ex.getMessage());
 }
 @ExceptionHandler(Exception.class)
 public Result doException(Exception ex){
 // 记录日志（错误堆栈）
 // 发送邮件给开发人员
 // 发送短信给运维人员
 return new Result(Code.SYSTEM_UNKNOW_ERROR,null,"系统繁忙，请联系管理员！");
 }
}
```

在controller包下

#### ⑥：异常处理器效果对比

```
{
 "data": {
 "id": 1,
 "type": "计算机理论",
 "name": "Spring实战 第5版",
 "description": "Spring入门经典教程，深入理解Spring原理技术内幕"
 },
 "code": 20041,
 "msg": null
}
```

```
{
 "data": null,
 "code": 60002,
 "msg": "请勿进行非法操作！"
}
```



## 1. 项目异常处理方案

- 异常处理器
- 自定义异常
- 异常编码
- 自定义消息

↓

## 24、SSM整合-前后台协议联调（列表功能）

```
<!DOCTYPE html>

<html>
 <head>
 <!-- 页面meta -->
 <meta charset="utf-8">
 <meta http-equiv="X-UA-Compatible" content="IE=edge">
 <title>SpringMVC案例</title>
 <meta content="width=device-width,initial-scale=1,maximum-scale=1,user-
scalable=no" name="viewport">
 <!-- 引入样式 -->
 <link rel="stylesheet" href="../plugins/elementui/index.css">
 <link rel="stylesheet" href="../plugins/font-awesome/css/font-
awesome.min.css">
 <link rel="stylesheet" href="../css/style.css">
 </head>
 <body class="hold-transition">
 <div id="app">
 <div class="content-header">
```

```
<h1>图书管理</h1>

</div>

<div class="app-container">

 <div class="box">

 <div class="filter-container">

 <el-input placeholder="图书名称" v-
model="pagination.queryString" style="width: 200px;" class="filter-item"></el-
input>

 <el-button @click="getAll()" class="dalfBut">查询</el-
button>

 <el-button type="primary" class="butT"
@click="handleCreate()">新建</el-button>

 </div>

 <el-table size="small" current-row-key="id" :data="dataList"
stripe highlight-current-row>

 <el-table-column type="index" align="center" label="序
号"></el-table-column>

 <el-table-column prop="type" label="图书类别"
align="center"></el-table-column>

 <el-table-column prop="name" label="图书名称"
align="center"></el-table-column>

 <el-table-column prop="description" label="描述"
align="center"></el-table-column>

 <el-table-column label="操作" align="center">

 <template slot-scope="scope">

 <el-button type="primary" size="mini"
@click="handleUpdate(scope.row)">编辑</el-button>

 <el-button type="danger" size="mini"
@click="handleDelete(scope.row)">删除</el-button>

 </template>

 </el-table-column>

 </el-table>

 <!-- 新增标签弹层 -->
 </div>
```

```
<div class="add-form">

 <el-dialog title="新增图书"
:visible.sync="dialogFormVisible">

 <el-form ref="dataAddForm" :model="formData"
:rules="rules" label-position="right" label-width="100px">

 <el-row>

 <el-col :span="12">
 <el-form-item label="图书类别"
prop="type">
 <el-input v-model="formData.type"/>
 </el-form-item>
 </el-col>

 <el-col :span="12">
 <el-form-item label="图书名称"
prop="name">
 <el-input v-model="formData.name"/>
 </el-form-item>
 </el-col>

 </el-row>

 <el-row>

 <el-col :span="24">
 <el-form-item label="描述">
 <el-input v-
model="formData.description" type="textarea"/>
 </el-form-item>
 </el-col>

 </el-row>

 </el-form>

 <div slot="footer" class="dialog-footer">
```

```
 <el-button @click="dialogFormVisible = false">取
消</el-button>

 <el-button type="primary" @click="handleAdd()">确
定</el-button>

 </div>

 </el-dialog>

</div>

<!-- 编辑标签弹层 -->

<div class="add-form">

 <el-dialog title="编辑检查项"
:visible.sync="dialogFormVisible4Edit">

 <el-form ref="dataEditForm" :model="formData"
:rules="rules" label-position="right" label-width="100px">

 <el-row>

 <el-col :span="12">

 <el-form-item label="图书类别"
prop="type">

 <el-input v-model="formData.type"/>

 </el-form-item>

 </el-col>

 <el-col :span="12">

 <el-form-item label="图书名称"
prop="name">

 <el-input v-model="formData.name"/>

 </el-form-item>

 </el-col>

 </el-row>

 <el-row>

 <el-col :span="24">

 <el-form-item label="描述">

 </el-col>

 </el-row>

 </el-form>
 </el-dialog>
</div>
```

```
 <el-input v-
model="formData.description" type="textarea"></el-input>

 </el-form-item>

 </el-col>

 </el-row>

 </el-form>

 <div slot="footer" class="dialog-footer">

 <el-button @click="dialogFormVisible4Edit =
false">取消</el-button>

 <el-button type="primary" @click="handleEdit()">
确定</el-button>

 </div>

 </el-dialog>

</div>

</div>

</div>

</body>

<!-- 引入组件库 -->

<script src="../js/vue.js"></script>

<script src="../plugins/elementui/index.js"></script>

<script type="text/javascript" src="../js/jquery.min.js"></script>

<script src="../js/axios-0.18.0.js"></script>

<script>
 var vue = new Vue({
 el: '#app',
 data: {
 pagination: {},
 dataList: [],//当前页要展示的列表数据
 formData: {},//表单数据
 dialogFormVisible: false,//控制表单是否可见
 dialogFormVisible4Edit:false,//编辑表单是否可见
 rules: {}//校验规则
 }
 })
</script>
```

```

 type: [{ required: true, message: '图书类别为必填项', trigger:
'blur' }],
 name: [{ required: true, message: '图书名称为必填项', trigger:
'blur' }]
 },
},
//钩子函数，VUE对象初始化完成后自动执行
created() {
 this.getAll();
},
methods: {
 //列表
 getAll() {
 //发送ajax请求
 axios.get("/books").then((res)=>{
 this.dataList = res.data.data;
 });
 },
}

```

前后端分离后后端就没这个了

## 25、SSM整合前后台协议联调（添加功能）

上html

```

//弹出添加窗口
handleCreate() {
 this.dialogFormVisible = true;
 this.resetForm();
},
//添加
handleAdd () {
 //发送ajax请求
 axios.post("/books",this.formData).then((res)=>{
 this.dialogFormVisible = false;
 this.getAll();
 });
}

```

## 26、SSM整合-前后台协议联调（添加功能状态处理）

上html

```
//列表
getAll() {
 //发送ajax请求
 axios.get("/books").then((res)=>{
 this.dataList = res.data.data;
 });
}

//弹出添加窗口
handleCreate() {
 this.dialogFormVisible = true;
 this.resetForm();
}

//重置表单
resetForm() {
 this.formData = {};
}

//添加
handleAdd () {
 //发送ajax请求
 axios.post("/books",this.formData).then((res)=>{
 console.log(res.data);
 //如果操作成功，关闭弹层，显示数据
 if(res.data.code == 20011){
 this.dialogFormVisible = false;
 this.$message.success("添加成功");
 }else if(res.data.code == 20010){
 this.$message.error("添加失败");
 }else{
 this.$message.error(res.data.msg);
 }
 }).finally(()=>{
 this.getAll();
 });
}
```

改一下接收数据的类

```
package com.itheima.dao;

import com.itheima.domain.Book;
import org.apache.ibatis.annotations.Delete;
```

```

import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.annotations.Update;

import java.util.List;

public interface BookDao {

 // @Insert("insert into tbl_book values(null,#{type},#{name},#
 {description})")
 @Insert("insert into tbl_book (type,name,description) values(#{type},#
 {name},#{description})")
 public int save(Book book);

 @Update("update tbl_book set type = #{type}, name = #{name}, description = #
 {description} where id = #{id}")
 public int update(Book book);

 @Delete("delete from tbl_book where id = #{id}")
 public int delete(Integer id);

 @Select("select * from tbl_book where id = #{id}")
 public Book getById(Integer id);

 @Select("select * from tbl_book")
 public List<Book> getAll();
}

```

## 26、SSM整合-前后台协议联调（修改功能）

```

//弹出编辑窗口
handleUpdate(row) {
 // console.log(row); //row.id 查询条件
 //查询数据，根据id查询
 axios.get("/books/" + row.id).then((res) => {
 // console.log(res.data.data);
 if (res.data.code == 20041) {
 //展示弹层，加载数据
 this.formData = res.data.data;
 this.dialogFormVisible4Edit = true;
 } else {
 this.$message.error(res.data.msg);
 }
 });
},
};

//编辑
handleEdit() {
 //发送ajax请求
 axios.put("/books", this.formData).then((res) => {
 //如果操作成功，关闭弹层，显示数据
 });
}

```

```
 if(res.data.code == 20031){
 this.dialogFormVisible4Edit = false;
 this.$message.success("修改成功");
 }else if(res.data.code == 20030){
 this.$message.error("修改失败");
 }else{
 this.$message.error(res.data.msg);
 }
 }).finally(()=>{
 this.getAll();
});
},
},
```

## 28、SSM整合-前后台协议联调（删除功能）

```
// 删除
handleDelete(row) {
 //1.弹出提示框
 this.$confirm("此操作永久删除当前数据，是否继续？","提示",{
 type:'info'
 }).then(()=>{
 //2.做删除业务
 axios.delete("/books/"+row.id).then((res)=>{
 if(res.data.code == 20021){
 this.$message.success("删除成功");
 }else{
 this.$message.error("删除失败");
 }
 }).finally(()=>{
 this.getAll();
 });
 }).catch(()=>{
 //3.取消删除
 this.$message.info("取消删除操作");
 });
}
```



## 案例：SSM整合标准开发

①：自定义项目系统级异常

```
axios.get("/books").then((res)=>{});
axios.post("/books",this.formData).then((res)=>{});
axios.delete("/books/"+row.id).then((res)=>{});
axios.put("/books",this.formData).then((res)=>{});
axios.get("/books/"+row.id).then((res)=>{});
```

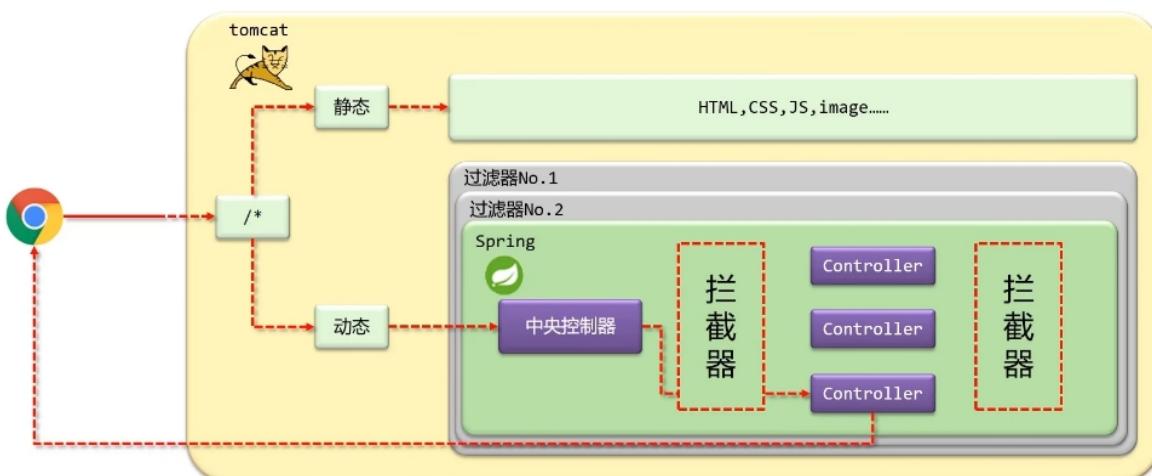


### 1. 案例：SSM整合标准开发

- 列表
- 新增
- 跳转到修改
- 修改
- 删除

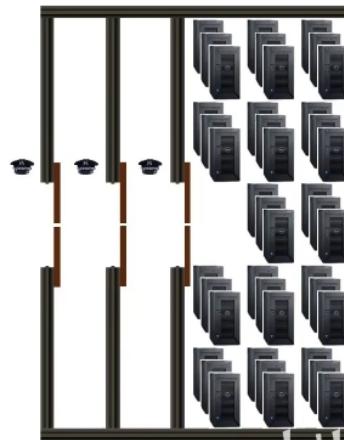
## 29、拦截器简介

拦截器概念



## 拦截器概念

- 拦截器 ( Interceptor ) 是一种动态拦截方法调用的机制
- 作用：
  - 在指定的方法调用前后执行预先设定后的代码
  - 阻止原始方法的执行



## 拦截器与过滤器区别

- 归属不同：Filter属于Servlet技术，Interceptor属于SpringMVC技术
- 拦截内容不同：Filter对所有访问进行增强，Interceptor仅针对SpringMVC的访问进行增强



### 1. 拦截器

- 概念
- 作用
- 执行时机

## 30、拦截器入门案例

①：声明拦截器的bean，并实现HandlerInterceptor接口（注意：扫描加载bean）

```
@Component
public class ProjectInterceptor implements HandlerInterceptor {
 public boolean preHandle(..) throws Exception {
 System.out.println("preHandle...");
 return true;
 }
 public void postHandle(..) throws Exception {
 System.out.println("postHandle...");
 }
 public void afterCompletion(..) throws Exception {
 System.out.println("afterCompletion...");
 }
}
```

在controller包下

②：定义配置类，继承WebMvcConfigurationSupport，实现addInterceptor方法（注意：扫描加载配置）

```
@Configuration
public class SpringMvcSupport extends WebMvcConfigurationSupport {
 @Override
 public void addInterceptors(InterceptorRegistry registry) {
 ...
 }
}
```

```
package com.itheima.config;

import com.itheima.controller.interceptor.ProjectInterceptor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import
org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport;

@Configuration
public class SpringMvcSupport extends WebMvcConfigurationSupport {
 @Autowired
 private ProjectInterceptor projectInterceptor;

 @Override
 protected void addResourceHandlers(ResourceHandlerRegistry registry) {
```

```

 registry.addResourceHandler("/pages/**").addResourceLocations("/pages/");
 }

 @Override
 protected void addInterceptors(InterceptorRegistry registry) {
 //配置拦截器

 registry.addInterceptor(projectInterceptor).addPathPatterns("/books", "/books/*");
 }
}

```

④：使用标准接口WebMvcConfigurer简化开发（注意：侵入式较强）

```

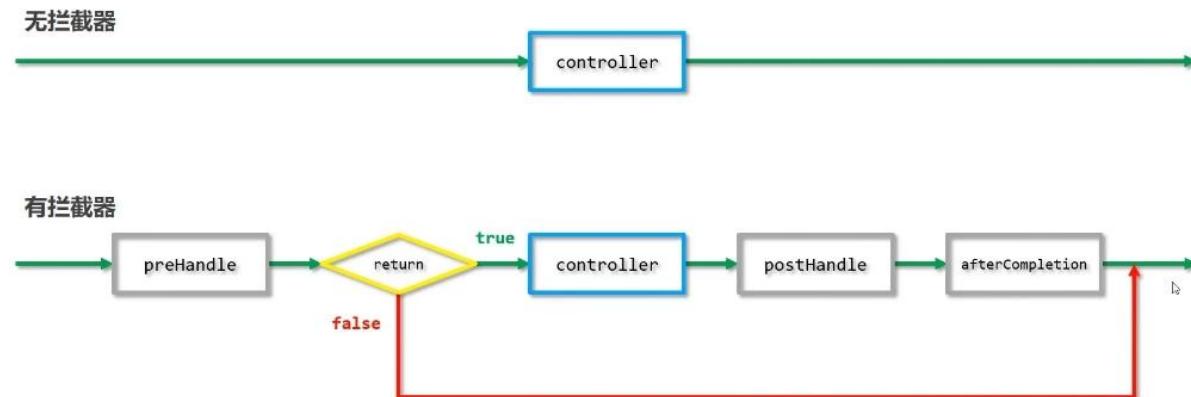
@Configuration
@ComponentScan("com.itheima.controller")
@EnableWebMvc
public class SpringMvcConfig implements WebMvcConfigurer{
 @Autowired
 private ProjectInterceptor projectInterceptor;

 public void addInterceptors(InterceptorRegistry registry) {
 registry.addInterceptor(projectInterceptor).addPathPatterns("/books", "/books/*");
 }
}

```

用这个就不用配SpringMvcSupport了

### 执行流程



## 1. 拦截器

- 定义
- 配置

## 2. 拦截器执行顺序

- preHandle
  - return true
    - ◆ controller
    - ◆ postHandle
    - ◆ afterCompletion
  - return false
    - ◆ 结束



小结

## 31、拦截器参数

### 拦截器参数

- 前置处理

```
public boolean preHandle(HttpServletRequest request,
 HttpServletResponse response,
 Object handler) throws Exception {
 System.out.println("preHandle...");
 return true;
}
```

- 参数

- request:请求对象
- response:响应对象
- handler:被调用的处理器对象，本质上是一个方法对象，对反射技术中的Method对象进行了再包装

- 返回值

- 返回值为false，被拦截的处理器将不执行

### 拦截器参数

- 后置处理

```
public void postHandle(HttpServletRequest request,
 HttpServletResponse response,
 Object handler,
 ModelAndView modelAndView) throws Exception {
 System.out.println("postHandle...");
}
```

- 参数

- modelAndView:如果处理器执行完成具有返回结果，可以读取到对应数据与页面信息，并进行调整

## 拦截器参数

- 完成后处理

```
public void afterCompletion(HttpServletRequest request,
 HttpServletResponse response,
 Object handler,
 Exception ex) throws Exception {
 System.out.println("afterCompletion...");
}
```

- 参数

- ex:如果处理器执行过程中出现异常对象，可以针对异常情况进行单独处理



### 1. 拦截器参数

- request
- response
- handle
- modelAndView
- ex

## 32、拦截器链配置

实现

```
package com.itheima.config;

import com.itheima.controller.interceptor.ProjectInterceptor;
import com.itheima.controller.interceptor.ProjectInterceptor2;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
```

```

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
@ComponentScan({"com.itheima.controller"})
@EnableWebMvc
//实现WebMvcConfigurer接口可以简化开发，但具有一定的侵入性
public class SpringMvcConfig implements WebMvcConfigurer {
 @Autowired
 private ProjectInterceptor projectInterceptor;
 @Autowired
 private ProjectInterceptor2 projectInterceptor2;

 @Override
 public void addInterceptors(InterceptorRegistry registry) {
 //配置多拦截器

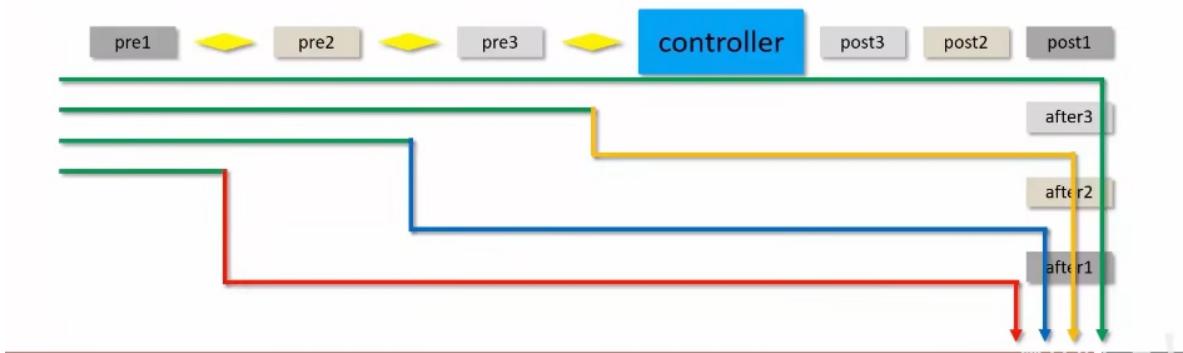
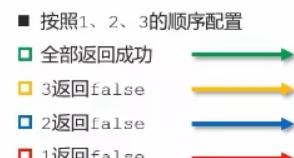
 registry.addInterceptor(projectInterceptor).addPathPatterns("/books","/books/*");
 }

 registry.addInterceptor(projectInterceptor2).addPathPatterns("/books","/books/*");
}
}

```

### 多拦截器执行顺序

- 当配置多个拦截器时，形成拦截器链
- 拦截器链的运行顺序参照拦截器添加顺序为准
- 当拦截器中出现对原始处理器的拦截，后面的拦截器均终止运行
- 当拦截器运行中断，仅运行配置在前面的拦截器的afterCompletion操作





1. 拦截器链配置方式
2. 拦截器链的运行顺序
  - preHandle: 与配置顺序相同, 必定运行
  - postHandle: 与配置顺序相反, 可能不运行
  - afterCompletion: 与配置顺序相反, 可能不运行



1. 拦截器概念
2. 入门案例
3. 拦截器参数
4. 多拦截器执行顺序

### 三、MyBatis

---

#### 1、MyBatis简介

# MyBatis

## 什么是MyBatis?

- MyBatis 是一款优秀的持久层框架，用于简化 JDBC 开发
- MyBatis 本是 Apache 的一个开源项目 iBatis, 2010 年这个项目由 apache software foundation 迁移到了 google code，并且改名为 MyBatis。2013 年 11 月迁移到 Github
- 官网：<https://mybatis.org/mybatis-3/zh/index.html>

## 持久层

- 负责将数据到保存到数据库的那一层代码
- JavaEE 三层架构：表现层、业务层、持久层

## 框架

- 框架就是一个半成品软件，是一套可重用的、通用的、软件基础代码模型
- 在框架的基础之上构建软件编写更加高效、规范、通用、可扩展

## MyBatis

来自 [黑马程序员](http://www.itheima.com)

### MyBatis 简化

#### 1. 硬编码 ➔ 配置文件

- 注册驱动，获取连接
- SQL 语句

```
// 需要驱动
Class.forName("com.mysql.jdbc.Driver");
// 获取connection连接
String url = "jdbc:mysql://db1?useSSL=false";
String user = "root";
String pwd = "1234";
Connection conn = DriverManager.getConnection(url, user, pwd);

String gender = "男";
// 定义sql
String sql = "select * from tb_user where gender = ?";

PreparedStatement pstmt = conn.prepareStatement(sql);
// 设置参数
pstmt.setString(1, gender);
// 执行sql
ResultSet rs = pstmt.executeQuery();
// 循环结果集
User user = null;
ArrayList<User> users = new ArrayList<>();
while (rs.next()) {
 // 取数据
 int id = rs.getInt("id");
 String username = rs.getString("username");
 String password = rs.getString("password");

 // 创建对象，设置属性值
 user = new User();
 user.setId(id);
 user.setUsername(username);
 user.setPassword(password);
 user.setGender(gender);
 users.add(user);
}
```

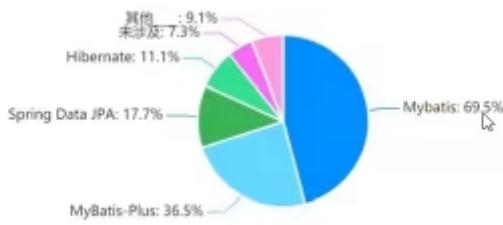
```
<!-- 连接池信息-->
<dataSource type="POOLED">
 <property name="driver" value="com.mysql.jdbc.Driver"/>
 <property name="url" value="jdbc:mysql://db1?useSSL=false"/>
 <property name="username" value="root"/>
 <property name="password" value="1234"/>
</dataSource>

<select id="selectByGender" parameterType="string" resultType="com.itheima.pojo.User">
 select * from tb_user where gender = #{gender};
</select>
```

#### 2. 操作繁琐 ➔ 自动完成

- 手动设置参数
- 手动封装结果集

MyBatis 免除了几乎所有的 JDBC 代码  
以及设置参数和获取结果集的工作



● Mybatis    ● MyBatis-Plus    ● Spring Data JPA    ● Hibernate    ● 未涉及    ● 其他\_\_



- ◆ MyBatis 快速入门
- ◆ Mapper 代理开发
- ◆ MyBatis 核心配置文件
- ◆ 配置文件完成增删改查
- ◆ 注解完成增删改查
- ◆ 动态 SQL

## 2、MyBatis快速入门

### MyBatis 快速入门



#### 步骤 查询user表中所有数据

1. 创建user表，添加数据
2. 创建模块，导入坐标
3. 编写 MyBatis 核心配置文件 --> 替换连接信息 解决硬编码问题
4. 编写 SQL 映射文件 --> 统一管理sql语句，解决硬编码问题
5. 编码
  1. 定义POJO类
  2. 加载核心配置文件，获取 SqlSessionFactory 对象
  3. 获取 SqlSession 对象，执行 SQL 语句
  4. 释放资源

```
<!--连接配置-->
<dataSource type="POOLED">
 <property name="driver" value="com.mysql.jdbc.Driver"/>
 <property name="url" value="jdbc:mysql://127.0.0.1:3306/testDB?useSSL=false"/>
 <property name="username" value="root"/>
 <property name="password" value="1234"/>
</dataSource>
```

```
<select id="selectAll" resultType="com.itheima.pojo.User">
 select * from tb_user;
</select>
```

#### 1、先增加MyBatis的依赖

```
<!--mybatis 依赖-->
<dependency>
 <groupId>org.mybatis</groupId>
 <artifactId>mybatis</artifactId>
 <version>3.5.5</version>
</dependency>
```

#### 2、增加Mysql的驱动

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

### 3、增加junit单元测试

```
<!--junit 单元测试-->
<dependency>
 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <version>4.13</version>
 <scope>test</scope>
</dependency>
```

### 4、logback坐标信息

```
<!-- 添加slf4j日志api -->
<dependency>
 <groupId>org.slf4j</groupId>
 <artifactId>slf4j-api</artifactId>
 <version>1.7.20</version>
</dependency>
<!-- 添加logback-classic依赖 -->
<dependency>
 <groupId>ch.qos.logback</groupId>
 <artifactId>logback-classic</artifactId>
 <version>1.2.3</version>
</dependency>
<!-- 添加logback-core依赖 -->
<dependency>
 <groupId>ch.qos.logback</groupId>
 <artifactId>logback-core</artifactId>
 <version>1.2.3</version>
</dependency>
```

### 5、从XML中配置SqlSessionFactory

```

<?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>

 <environments default="development">
 <environment id="development">
 <transactionManager type="JDBC"/>
 <dataSource type="POOLED">
 <!--数据库连接信息-->
 <property name="driver" value="com.mysql.jdbc.Driver"/>
 <property name="url" value="jdbc:mysql:///mybatis?
useSSL=false"/>
 <property name="username" value="root"/>
 <property name="password" value="1234"/>
 </dataSource>
 </environment>
 </environments>
 <mappers>
 <!--加载sql映射文件-->
 <mapper resource="com/itheima/mapper/UserMapper.xml"/>

 </mappers>
</configuration>

```

## 6、配置SQL映射文件

文件名一般为 `xxxMapper.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: 命名空间
 -->
 <mapper namespace="test">
 <select id="selectAll" resultType="com.itheima.pojo.User">
 select * from tb_user;
 </select>
 </mapper>

```

`resultType`后面是要写对应类的地址

## 7、构造User类

```
package com.itheima.pojo;

// alt + 鼠标左键 整列编辑
public class User {

 private Integer id;
 private String username;
 private String password;
 private String gender;
 private String addr;

 //Alt+insert
 public Integer getId() {
 return id;
 }

 public void setId(Integer id) {
 this.id = id;
 }

 public String getUsername() {
 return username;
 }

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

 public String getPassword() {
 return password;
 }

 public void setPassword(String password) {
 this.password = password;
 }

 public String getGender() {
 return gender;
 }

 public void setGender(String gender) {
 this.gender = gender;
 }

 public String getAddr() {
 return addr;
 }

 public void setAddr(String addr) {
```

```

 this.addr = addr;
}

@Override
public String toString() {
 return "User{" +
 "id=" + id +
 ", username='" + username + '\'' +
 ", password='" + password + '\'' +
 ", gender='" + gender + '\'' +
 ", addr='" + addr + '\'' +
 '}';
}
}

```

## 8、加载核心配置文件

```


/**
 * Mybatis 快速入门代码
 */
public class MyBatisDemo {

 public static void main(String[] args) throws IOException {

 //1. 加载mybatis的核心配置文件，获取 SqlSessionFactory
 String resource = "mybatis-config.xml";
 InputStream inputStream = Resources.getResourceAsStream(resource);
 SqlSessionFactory sqlSessionFactory = new
 SqlSessionFactoryBuilder().build(inputStream);

 //2. 获取SqlSession对象，用它来执行sql
 SqlSession sqlSession = sqlSessionFactory.openSession();
 //3. 执行sql
 List<User> users = sqlSession.selectList("test.selectAll");
 System.out.println(users);
 //4. 释放资源
 sqlSession.close();

 }
}


```

## 3、解决SQL语句警告问题

往ide添加数据库即可

## 4、Mapper代理开发

### Mapper 代理开发

黑马程序员 www.itheima.com

- 目的

- 解决原生方式中的硬编码
- 简化后期执行SQL

```
//3. 执行sql
List<User> users = sqlSession.selectList(statement: "test.selectAll");
System.out.println(users);
```

```
//3. 获取接口代理对象
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
//4. 执行方法，其实就是执行sql语句
List<User> users = userMapper.selectAll();
```

### Mapper 代理开发



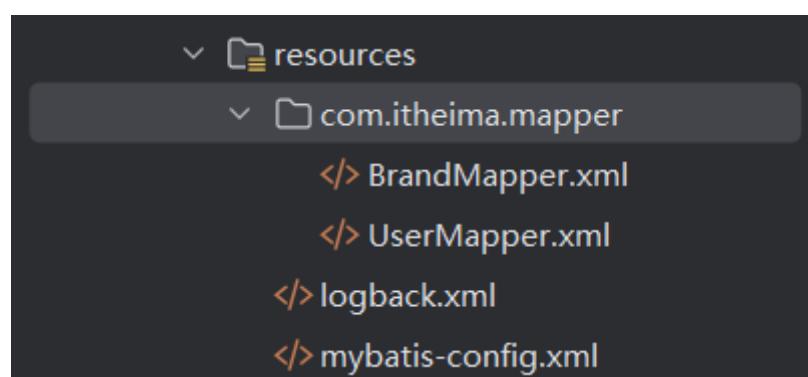
#### 步骤 使用 Mapper 代理方式完成入门案例

1. 定义与SQL映射文件同名的Mapper接口，并且将Mapper接口和SQL映射文件放置在同一目录下
2. 设置SQL映射文件的namespace属性为Mapper接口全限定名
3. 在 Mapper 接口中定义方法，方法名就是SQL映射文件中sql语句的id，并保持参数类型和返回值类型一致
4. 编码
  1. 通过 SqlSession 的 getMapper方法获取 Mapper接口的代理对象
  2. 调用对应方法完成sql的执行

细节：如果Mapper接口名称和SQL映射文件名称相同，并在同一目录下，则可以使用包扫描的方式简化SQL映射文件的加载

```
<mappers>
 <!-- 加载sql的映射文件-->
 <!--<mapper resource="com/itheima/mapper2/UserMapper.xml"/>-->
 <package name="com.itheima.mapper"/>
</mappers>
```

Mapper文件放入文件夹



编译文件时要com/itheima/mapper

UserMapper.xml

```
<mapper namespace="test">

 <!--statement-->
 <select id="selectAll" resultType="com.itheima.pojo.User">
 select *
 from tb_user;
 </select>

 <select id="select" resultType="com.itheima.pojo.User">
 select *
 from tb_user
 where
 username = #{arg0}
 and password = #{param2}
 </select>

</mapper>
```

mabatis-config也要改一下

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

</mappers>
```

写一下启动类

```
public class MyBatisDemo2 {

 public static void main(String[] args) throws IOException {
 //1. 加载mybatis的核心配置文件，获取 SqlSessionFactory
 String resource = "mybatis-config.xml";
 InputStream inputStream = Resources.getResourceAsStream(resource);
 SqlSessionFactory sqlSessionFactory = new
 SqlSessionFactoryBuilder().build(inputStream);
```

```
//2. 获取sqlSession对象, 用它来执行sql
SqlSession sqlSession = sqlSessionFactory.openSession();
//3. 执行sql
//List<User> users = sqlSession.selectList("test.selectAll");
//3.1 获取UserMapper接口的代理对象
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
List<User> users = userMapper.selectAll();

System.out.println(users);
//4. 释放资源
sqlSession.close();

}
}
```

Usermapper

```
public interface UserMapper {
 List<User> selectAll();
}
```

## 5、Mybatis核心配置文件

### Mapper 代理开发



- 目的

- 解决原生方式中的硬编码
- 简化后期执行SQL

```
//3. 执行sql
List<User> users = sqlSession.selectList(statement: "test.selectAll");
System.out.println(users);
```

```
//3. 获取接口代理对象
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
//4. 执行方法, 其实就是执行sql语句
List<User> users = userMapper.selectAll();
```

```

<?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>
 <typeAliases>
 <package name="com.itheima.pojo"/>
 </typeAliases>
 <!--
 environments: 配置数据连接环境信息，可以配置多个environment，通过default属性切换不同的environment
 -->
 <environments default="development">
 <environment id="development">
 <transactionManager type="JDBC"/>
 </environment>
 </environments>
</configuration>

```

Run: MyBatisDemo2

```

[DEBUG] 16:45:32.386 [main] c.i.m.U.selectAll - <== Total: 4
[User{id=1, username='zhangsan', password='123', gender='男', addr='北京'}, User{id=2, username='李四', password='234', gender='女', addr='天津'}]
[DEBUG] 16:45:32.387 [main] o.a.i.t.j.JdbcTransaction - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@5d47c63f]
[DEBUG] 16:45:32.387 [main] o.a.i.t.j.JdbcTransaction - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@5d47c63f]
[DEBUG] 16:45:32.388 [main] o.a.i.d.p.PooledDataSource - Returned connection 1564984895 to pool.

```

Process finished with exit code 0

这个意思就是把pugu这个包里的所有类起了一个别名user(不区分大小写)

## 6、MyBatis案例-环境准备



## 7、查询-查询所有&结果映射

## 配置文件完成增删改查

### 练习 查询-查询所有数据

序号	品牌LOGO	品牌名称	企业名称	排序	启用状态
010		三只松鼠	近距离企业名称	15	<input checked="" type="checkbox"/>
009		优衣库	近距离企业名称	16	<input checked="" type="checkbox"/>
008		小米	近距离企业名称	1	<input checked="" type="checkbox"/>
007		阿里巴巴	近距离企业名称	14	<input checked="" type="checkbox"/>
006		屈臣氏	近距离企业名称	8	<input type="checkbox"/>

1. 编写接口方法: Mapper接口

➤ 参数: 无

➤ 结果: List<Brand>

2. 编写 SQL语句: SQL映射文件:

3. 执行方法, 测试

List<Brand> selectAll();

```
<select id="selectAll" resultType="brand">
 select * from tb_brand;
</select>
```

1、创建BrandMapper方法

```
public interface BrandMapper {
 /**
 * 查询所有
 */
 List<Brand> selectAll();
}
```

2、创建BrandMapper.xml文件

```
<?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">

<!--
数据库表的字段名称 和 实体类的属性名称 不一样，则不能自动封装数据
* 起别名：对不一样的列名起别名，让别名和实体类的属性名一样
* 缺点：每次查询都要定义一次别名
* sql片段
* 缺点：不灵活
* resultMap:
1. 定义<resultMap>标签
-->
```

## 2. 在<select>标签中，使用resultMap属性替换 resultType属性

```
-->
<!--
 数据库表的字段名称 和 实体类的属性名称 不一样，则不能自动封装数据
-->
<mapper namespace="com.itheima.mapper.BrandMapper">
 <select id="selectAll" resultMap="brandResultMap">
 select *
 from tb_brand;
 </select>

 <!--
 sql片段
 -->
 <sql id="brand_column">
 id, brand_name as brandName, company_name as companyName, ordered,
description, status
 </sql>

 <select id="selectAll" resultType="brand">
 select
 <include refid="brand_column" />
 from tb_brand;
 </select>

 <!--
 id: 唯一标识
 type: 映射的类型，支持别名
 -->
 <resultMap id="brandResultMap" type="brand">
 <!--
 id: 完成主键字段的映射
 column: 表的列名
 property: 实体类的属性名
 result: 完成一般字段的映射
 column: 表的列名
 property: 实体类的属性名
 -->
 <result column="brand_name" property="brandName"/>
 <result column="company_name" property="companyName"/>
 </resultMap>
```

### 起别名

```
<select id="selectAll" resultType="brand">
 select id, brand_name as brandName, company_name as companyName, ordered, description, status
 from tb_brand;
</select>
```

### 3、写测试用例

```

public class MyBatisTest {

 @Test
 public void testSelectAll() throws IOException {
 //1. 获取SqlSessionFactory
 String resource = "mybatis-config.xml";
 InputStream inputStream = Resources.getResourceAsStream(resource);
 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

 //2. 获取SqlSession对象
 SqlSession sqlSession = sqlSessionFactory.openSession();

 //3. 获取Mapper接口的代理对象
 BrandMapper brandMapper = sqlSession.getMapper(BrandMapper.class);

 //4. 执行方法
 List<Brand> brands = brandMapper.selectAll();
 System.out.println(brands);

 //5. 释放资源
 sqlSession.close();
 }
}

```

## 8、查询-查看详情

配置文件完成增删改查



### 练习 查询-查看详情

品牌名称	企业名称	排序	当前状态	操作
三只松鼠	这里是企业名称	15	<input checked="" type="radio"/>	删除 编辑 查看详情
优衣库	这里是企业名称	16	<input checked="" type="radio"/>	删除 编辑 查看详情
小米	这里是企业名称	1	<input checked="" type="radio"/>	删除 编辑 查看详情

1. 编写接口方法：Mapper接口
- 参数: id
- 结果: Brand
2. 编写 SQL语句：SQL映射文件
3. 执行方法，测试

Brand selectById(int id);

```

<select id="selectById" parameterType="int" resultType="brand">
 select * from tb_brand where id = #{id};
</select>

```

1、配置BrandaMapper接口

```
public interface BrandMapper {

 /**
 * 查看详情：根据Id查询
 */
 Brand selectById(int id);
}
```

## 2、配置BrandMapper.xml文件

```
<!--
 * 参数占位符:
 1. #{:}：会将其替换为 ?，为了防止SQL注入
 2. ${:}：拼sql。会存在SQL注入问题
 3. 使用时机:
 * 参数传递的时候: #{:}
 * 表名或者列名不固定的情况下: ${:} 会存在SQL注入问题

 * 参数类型: parameterType: 可以省略
 * 特殊字符处理:
 1. 转义字符:
 2. CDATA区:
-->
<!-- <select id="selectById" resultMap="brandResultMap">
 select *
 from tb_brand where id = #{id};

 </select>
-->
<select id="selectById" resultMap="brandResultMap">
 select *
 from tb_brand
 where id
 <! [CDATA[
 <
]]>
 #{id};

 </select>
```

## 3、写测试用例

```
@Test
public void testSelectById() throws IOException {
 //接收参数
 int id = 1;
```

```
//1. 获取SqlSessionFactory
String resource = "mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

//2. 获取SqlSession对象
SqlSession sqlSession = sqlSessionFactory.openSession();

//3. 获取Mapper接口的代理对象
BrandMapper brandMapper = sqlSession.getMapper(BrandMapper.class);

//4. 执行方法
Brand brand = brandMapper.selectById(id);
System.out.println(brand);

//5. 释放资源
sqlSession.close();

}
```

#### 1. 参数占位符:

- 1) #{}: 执行SQL时, 会将#{}占位符替换为?, 将来自动设置参数值
- 2) \${}: 拼SQL。会存在SQL注入问题
3. 使用时机:
  - \* 参数传递, 都使用#{}  - \* 如果要对表名、列名进行动态设置, 只能使用\${}进行sql拼接。

#### 2. parameterType:

- \* 用于设置参数类型, 该参数可以省略

#### 3. SQL语句中特殊字符处理:

- \* 转义字符
- \* <![CDATA[ 内容 ]]>: CD提示

## 9、查询-条件查询

练习 查询-多条件查询


1. 编写接口方法: Mapper接口
- 参数: 所有查询条件
- 结果: List<Brand>
2. 编写 SQL语句: SQL映射文件
3. 执行方法, 测试

```
List<Brand> selectByCondition(@Param("status")int status, @Param("companyName") String companyName, @Param("brandName") String brandName);

List<Brand> selectByCondition(Brand brand);

List<Brand> selectByCondition(Map map);

<select id="selectByCondition" resultMap="brandResultMap">
 select *
 from tb_brand
 where
 status = #{status}
 and company_name like #{companyName}
 and brand_name like #{brandName}
</select>
```

## 1、分析sql语句

**1. 条件表达式**  
**2. 如何连接**  
 当前状态

## 2、配置BrandMapper.xml文件

```
<!--
 条件查询
-->
<select id="selectByCondition" resultMap="brandResultMap">
 select *
 from tb_brand
 where status = #{status}
 and company_name like #{companyName}
 and brand_name like #{brandName}
</select>
```

## 3、配置BrandMapper接口

```
/**
```

```

* 条件查询
* * 参数接收
* 1. 散装参数：如果方法中有多个参数，需要使用@Param("SQL参数占位符名称")
* 2. 对象参数：对象的属性名称要和参数占位符名称一致
* 3. map集合参数
*
* @param status
* @param companyName
* @param brandName
* @return
*
*/

```

```

List<Brand> selectByCondition(@Param("status") int status,
@Param("companyName") String companyName, @Param("brandName") String brandName);

```

```

List<Brand> selectByCondition(Brand brand);

List<Brand> selectByCondition(Map map);

/**
* 单条件动态查询
* @param brand
* @return
*/
List<Brand> selectByConditionSingle(Brand brand);

```

#### 4、配置测试类

```

@Test
public void testSelectByCondition() throws IOException {
 //接收参数
 int status = 1;
 String companyName = "华为";
 String brandName = "华为";

 // 处理参数
 companyName = "%" + companyName + "%";
 brandName = "%" + brandName + "%";

 //封装对象
 /* Brand brand = new Brand();
 brand.setStatus(status);
 brand.setCompanyName(companyName);
 brand.setBrandName(brandName); */

 Map map = new HashMap();
 // map.put("status" , status);
 map.put("companyName", companyName);
 // map.put("brandName" , brandName);
}

```

```

//1. 获取SqlSessionFactory
String resource = "mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

//2. 获取SqlSession对象
SqlSession sqlSession = sqlSessionFactory.openSession();

//3. 获取Mapper接口的代理对象
BrandMapper brandMapper = sqlSession.getMapper(BrandMapper.class);

//4. 执行方法

//List<Brand> brands = brandMapper.selectByCondition(status,
companyName, brandName);
// List<Brand> brands = brandMapper.selectByCondition(brand);
List<Brand> brands = brandMapper.selectByCondition(map);
System.out.println(brands);

//5. 释放资源
sqlSession.close();
}

```

## 10、查询-动态条件查询

### 练习 查询-多条件-动态条件查询

- SQL语句会随着用户的输入或外部条件的变化而变化，我们称为 **动态SQL**

```

<select id="selectByCondition" resultMap="brandResultMap">
 select *
 from tb_brand
 where
 if(status != null)
 status = #{status}
 and company_name like #{companyName}
 and brand_name like #{brandName}
</select>

```

- MyBatis 对动态SQL有很强大的支撑：
  - if
  - choose (when, otherwise)
  - trim (where, set)
  - foreach

在BrandMapper.xml中，改写为：

```

<!--
 动态条件查询
 * if: 条件判断
 * test: 逻辑表达式
 * 问题:
 * 恒等式 (where 1=1)
 * <where> 替换 where 关键字
-->
<select id="selectByCondition" resultMap="brandResultMap">
 select *
 from tb_brand
 /* where 1 = 1*/
 <where>

 <if test="status != null">
 and status = #{status}
 </if>
 <if test="companyName != null and companyName != '' ">
 and company_name like #{companyName}
 </if>
 <if test="brandName != null and brandName != '' ">
 and brand_name like #{brandName}
 </if>
 </where>

</select>

```

## 配置文件完成增删改查



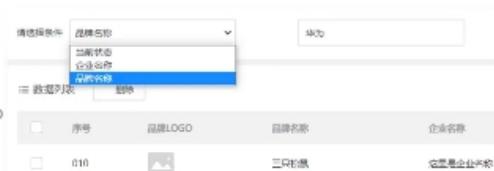
### 动态SQL

- if:** 用于判断参数是否有值，使用test属性进行条件判断
- \* 存在的问题：第一个条件不需要逻辑运算符
  - \* 解决方案：
    - 1) 使用恒等式让所有条件格式都一样
    - 2) <where> 标签替换 where 关键字

练习 查询-单条件-动态条件查询

- 从多个条件中选择一个
- choose (when, otherwise): 选择, 类似于Java 中的 switch 语句

```
<select id="selectByConditionSingle" resultMap="brandResultMap">
 select *
 from tb_brand
 where
 <choose> <!--类似于switch-->
 <when test="status != null"> <!--类似于case-->
 status = #{status}
 </when>
 <when test="companyName != null and companyName != ''">
 company_name like #{companyName}
 </when>
 <when test="brandName != null and brandName != ''">
 brand_name like #{brandName}
 </when>
 <otherwise> <!--类似于default-->
 1 = 1
 </otherwise>
 </choose>
</select>
```



配置BrandMapper.xml文件

```
<select id="selectByConditionSingle" resultMap="brandResultMap">
 select *
 from tb_brand
 <where>
 <choose><!--相当于switch-->
 <when test="status != null"><!--相当于case-->
 status = #{status}
 </when>
 <when test="companyName != null and companyName != ''"><!--相当于
case-->
 company_name like #{companyName}
 </when>
 <when test="brandName != null and brandName != ''"><!--相当于case-
-->
 brand_name like #{brandName}
 </when>
 </choose>
 </where>
</select>
```

配置测试文件

```
@Test
```

```
public void testSelectByConditionSingle() throws IOException {
 //接收参数
 int status = 1;
 String companyName = "华为";
 String brandName = "华为";

 // 处理参数
 companyName = "%" + companyName + "%";
 brandName = "%" + brandName + "%";

 //封装对象
 Brand brand = new Brand();
 //brand.setStatus(status);
 brand.setCompanyName(companyName);
 //brand.setBrandName(brandName);

 //1. 获取SqlSessionFactory
 String resource = "mybatis-config.xml";
 InputStream inputStream = Resources.getResourceAsStream(resource);
 SqlSessionFactory sqlSessionFactory = new
 SqlSessionFactoryBuilder().build(inputStream);

 //2. 获取SqlSession对象
 SqlSession sqlSession = sqlSessionFactory.openSession();

 //3. 获取Mapper接口的代理对象
 BrandMapper brandMapper = sqlSession.getMapper(BrandMapper.class);

 //4. 执行方法

 //List<Brand> brands = brandMapper.selectByCondition(status,
 companyName, brandName);
 // List<Brand> brands = brandMapper.selectByCondition(brand);

 List<Brand> brands = brandMapper.selectByConditionSingle(brand);
 System.out.println(brands);

 //5. 释放资源
 sqlSession.close();
}

}
```

## 11、添加&修改功能

## 配置文件完成增删改查

练习 添加

1. 编写接口方法：Mapper接口

```
void add(Brand brand);
```

➢ 参数：除了id之外的所有数据

➢ 结果：void

2. 编写SQL语句：SQL映射文件

```
<insert id="add">
 insert into tb_brand (brand_name, company_name, ordered, description, status)
 values (#{brandName}, #{companyName}, #{ordered}, #{description}, #{status});
</insert>
```

3. 执行方法，测试

• MyBatis事务：

➢ openSession()：默认开启事务，进行增删改操作后需要使用 sqlSession.commit()；手动提交事务

➢ openSession(true)：可以设置为自动提交事务（关闭事务）



1、配置BrandMapper.xml文件

```
<insert id="add" useGeneratedKeys="true" keyProperty="id">
 insert into tb_brand (brand_name, company_name, ordered, description,
 status)
 values (#{brandName}, #{companyName}, #{ordered}, #{description}, #
 {status});

</insert>
```

2、要提交事务

```
@Test
public void testUpdate() throws IOException {
 //接收参数
 int status = 0;
 String companyName = "波导手机";
 String brandName = "波导";
 String description = "波导手机,手机中的战斗机";
 int ordered = 200;
 int id = 6;

 //封装对象
 Brand brand = new Brand();
 brand.setStatus(status);
 // brand.setCompanyName(companyName);
 // brand.setBrandName(brandName);
 // brand.setDescription(description);
```

```

// brand.setOrdered(ordered);
brand.setId(id);

//1. 获取sqlSessionFactory
String resource = "mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

//2. 获取sqlSession对象
SqlSession sqlSession = sqlSessionFactory.openSession();
//SqlSession sqlSession = sqlSessionFactory.openSession(true);

//3. 获取Mapper接口的代理对象
BrandMapper brandMapper = sqlSession.getMapper(BrandMapper.class);

//4. 执行方法

int count = brandMapper.update(brand);
System.out.println(count);
//提交事务
sqlSession.commit();

//5. 释放资源
sqlSession.close();

}

```

**练习 修改 – 修改全部字段**

编辑商品品牌

品牌LOGO

\* 品牌名称:

\* 企业名称:

排序: 20

备注信息: 在逻辑清空的基础上，用相对简洁的语言进行描述，每句尽量不要超过15个字，如果一句描述超过15个字，尽量从中间带引号的地方，在逻辑清空的基础上，用相对简洁的语言进行描述。

当前状态:  启用  禁用

提交 取消

- 编写接口方法: Mapper接口
- 参数: 所有数据
- 结果: void

2. 编写 SQL语句: SQL映射文件

3. 执行方法, 测试

```

<update id="update">
 update tb_brand
 set brand_name = #{brandName},
 company_name = #{companyName},
 ordered = #{ordered},
 description = #{description},
 status = #{status}
 where id = #{id};
</update>

```

## 练习 修改 – 修改动态字段

1. 编写接口方法: Mapper接口

➢ 参数: 部分数据, 封装到对象中

➢ 结果: void

2. 编写 SQL语句: SQL映射文件

3. 执行方法, 测试

```
<update id="update">
 update tb_user
 set
 username = #{username},
 password = #{password},
 gender = #{gender},
 addr = #{addr}
 where
 id = #{id}
</update>
```



```
<update id="update">
 update tb_brand
 <set>
 <if test="brandName != null and brandName != ''">
 brand_name = #{brandName},
 </if>
 <if test="companyName != null and companyName != ''">
 company_name = #{companyName},
 </if>
 <if test="ordered != null">
 ordered = #{ordered},
 </if>
 <if test="description != null and description != ''">
 description = #{description},
 </if>
 <if test="status != null">
 status = #{status},
 </if>
 </set>
 where id = #{id};
</update>
```

xml文件

```
<!--
 <set>跟动态sql有关
-->
<update id="update">
 update tb_brand
 <set>
 <if test="brandName != null and brandName != ''">
 brand_name = #{brandName},
 </if>
 <if test="companyName != null and companyName != ''">
 company_name = #{companyName},
 </if>
 <if test="ordered != null">
 ordered = #{ordered},
 </if>
 <if test="description != null and description != ''">
 description = #{description},
 </if>
 <if test="status != null">
 status = #{status}
 </if>
 </set>
 where id = #{id};
</update>
```

set为自动分隔

## 12、删除功能

练习：删除一个

企业名称	排序	当前状态	操作
国美企业名称	6	正常	<span style="border: 2px solid red; padding: 2px;">删除</span>   编辑   查看详情
国美企业名称	15	正常	删除   编辑   查看详情

1. 编写接口方法：Mapper接口  
➤ 参数：id  
➤ 结果：void

2. 编写 SQL语句：SQL映射文件

3. 执行方法，测试

```
<delete id="deleteById">
 delete from tb_brand where id = #{id};
</delete>
```

xml

```
<delete id="deleteById">
 delete from tb_brand where id = #{id};
</delete>
```

方法

```
@Test
public void testDeleteById() throws IOException {
 //接收参数

 int id = 6;

 //1. 获取SqlSessionFactory
 String resource = "mybatis-config.xml";
 InputStream inputStream = Resources.getResourceAsStream(resource);
 SqlSessionFactory sqlSessionFactory = new
 SqlSessionFactoryBuilder().build(inputStream);

 //2. 获取SqlSession对象
 SqlSession sqlSession = sqlSessionFactory.openSession();
 //SqlSession sqlSession = sqlSessionFactory.openSession(true);

 //3. 获取Mapper接口的代理对象
 BrandMapper brandMapper = sqlSession.getMapper(BrandMapper.class);
```

```

//4. 执行方法

brandMapper.deleteById(id);

//提交事务
sqlSession.commit();

//5. 释放资源
sqlSession.close();

}

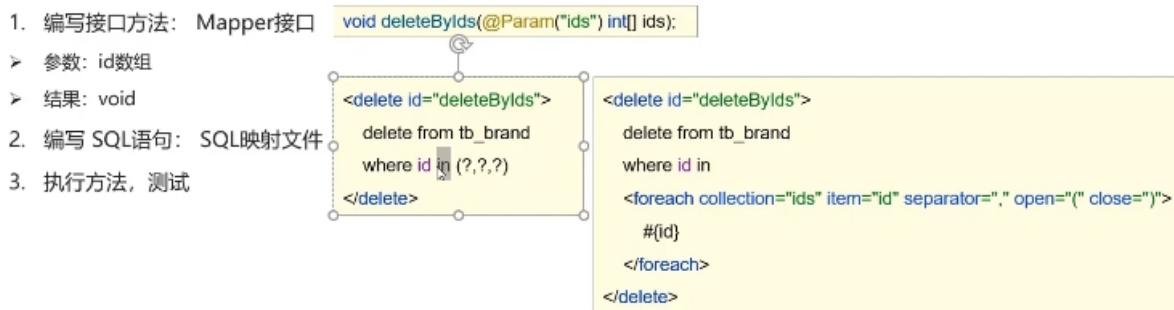
```

## 配置文件完成增删改查



### 练习 批量删除

<input type="checkbox"/>	序号	品牌LOGO	品牌名称	企业名称
<input checked="" type="checkbox"/>	010		三只松鼠	坚果零食企业名称
<input checked="" type="checkbox"/>	009		优衣库	快时尚企业名称
<input type="checkbox"/>	003		小米	智能家居企业名称



xml

```

<!--
mybatis会将数组参数，封装为一个Map集合。
* 默认： array = 数组
* 使用@Param注解改变map集合的默认key的名称
* separator:用逗号分开
-->

<delete id="deleteByIds">
 delete from tb_brand where id
 in
 <foreach collection="array" item="id" separator="," open="("
close="")">
 #{id}
 </foreach>
 ;
</delete>

```

## 方法

```
@Test
public void testDeleteByIds() throws IOException {
 //接收参数

 int[] ids = {5,7,8};

 //1. 获取sqlSessionFactory
 String resource = "mybatis-config.xml";
 InputStream inputStream = Resources.getResourceAsStream(resource);
 SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

 //2. 获取sqlSession对象
 SqlSession sqlSession = sqlSessionFactory.openSession();
 //SqlSession sqlSession = sqlSessionFactory.openSession(true);

 //3. 获取Mapper接口的代理对象
 BrandMapper brandMapper = sqlSession.getMapper(BrandMapper.class);

 //4. 执行方法

 brandMapper.deleteByIds(ids);

 //提交事务
 sqlSession.commit();

 //5. 释放资源
 sqlSession.close();

}
```

注意：要么在接口写@param，要么在xml中写array

```
/**
 * 批量删除
 */
void deleteByIds(@Param("ids") int[] ids);
```

## 13、参数传递

MyBatis 接口方法中可以接收各种各样的参数，MyBatis底层对于这些参数进行不同的封装处理方式

- 单个参数：
  - 1. POJO类型:
  - 2. Map集合:
  - 3. Collection:
  - 4. List:
  - 5. Array:
  - 6. 其他类型:
- 多个参数:

```
User select(@Param("username") String username,@Param("password")String password);

<select id="select" resultType="user">
 select *
 from tb_user
 where
 username = #{username}
 and password = #{password};
</select>
```

MyBatis提供了 ParamNameResolver 类来进行参数封装

```
public interface UserMapper {

 List<User> selectAll();

 @Select("select * from tb_user where id = #{id}")
 User selectById(int id);

 /*

 MyBatis 参数封装:
 * 单个参数:
 1. POJO类型: 直接使用, 属性名 和 参数占位符名称 一致
 2. Map集合: 直接使用, 键名 和 参数占位符名称 一致
 3. Collection: 封装为Map集合, 可以使用@Param注解, 替换Map集合中默认的arg键名
 map.put("arg0", collection集合);
 map.put("collection", collection集合);
 4. List: 封装为Map集合, 可以使用@Param注解, 替换Map集合中默认的arg键名
 map.put("arg0", list集合);
 map.put("collection", list集合);
 map.put("list", list集合);
 5. Array: 封装为Map集合, 可以使用@Param注解, 替换Map集合中默认的arg键名
 map.put("arg0", 数组);
 map.put("array", 数组);
 6. 其他类型: 直接使用
 * 多个参数: 封装为Map集合, 可以使用@Param注解, 替换Map集合中默认的arg键名
 map.put("arg0", 参数值1)
 map.put("param1", 参数值1)
 map.put("param2", 参数值2)
 map.put("arg1", 参数值2)
 -----@Param("username")
 map.put("username", 参数值1)
```

```

 map.put("param1",参数值1)
 map.put("param2",参数值2)
 map.put("agr1",参数值2)

 */
User select(@Param("username") String username,String password);
User select(Collection collection);
}

```

## 14、注解开发

使用注解开发会比配置文件开发更加方便

```

@Select("select * from tb_user where id = #{id}")
public User selectById(int id);

```

- 查询: @Select 提示:
- 添加: @Insert ➤ 注解完成简单功能
- 修改: @Update ➤ 配置文件完成复杂功能
- 删除: @Delete

使用注解来映射简单语句会使得代码量更加简洁，但对于稍微复杂一点的语句，Java 注解不仅力不从心，还会让你本就复杂的 SQL 语句更加混乱不堪。因此，如果你需要做一些很复杂的操作，最好用 XML 来映射语句。

选择哪种方式来配置映射，以及认为是否应该要统一映射语句定义的形式，完全取决于你和你的团队。换句话说，永远不要拘泥于一种方式。你可以很轻松的在基于注解和 XML 的语句映射方式间自由移植和切换。

```

public interface UserMapper {
 List<User> selectAll();
 @Select("select * from tb_user where id = #{id}")
 User selectById(int id);
}

```