

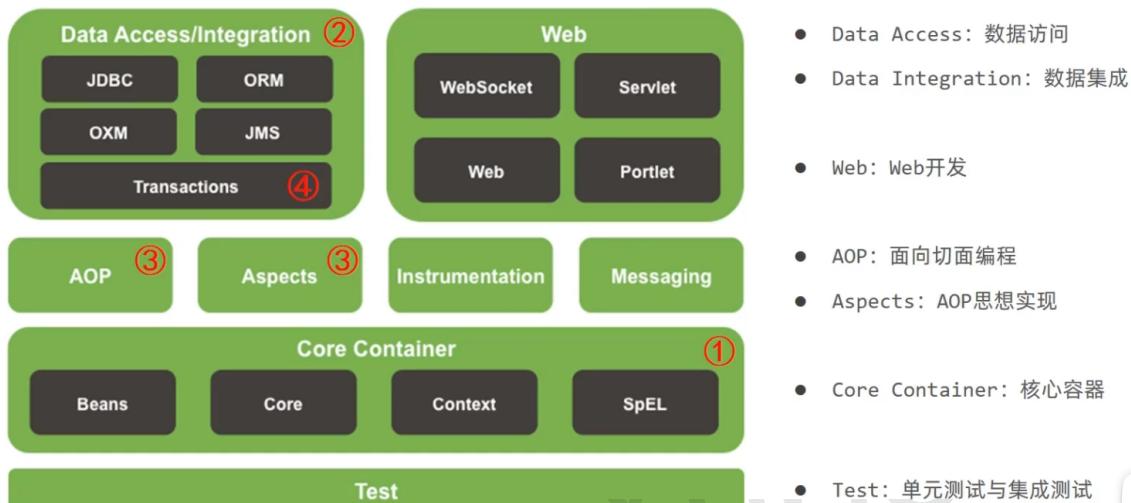
SSM

[SSM框架-导学内容](#)哔哩哔哩bilibili

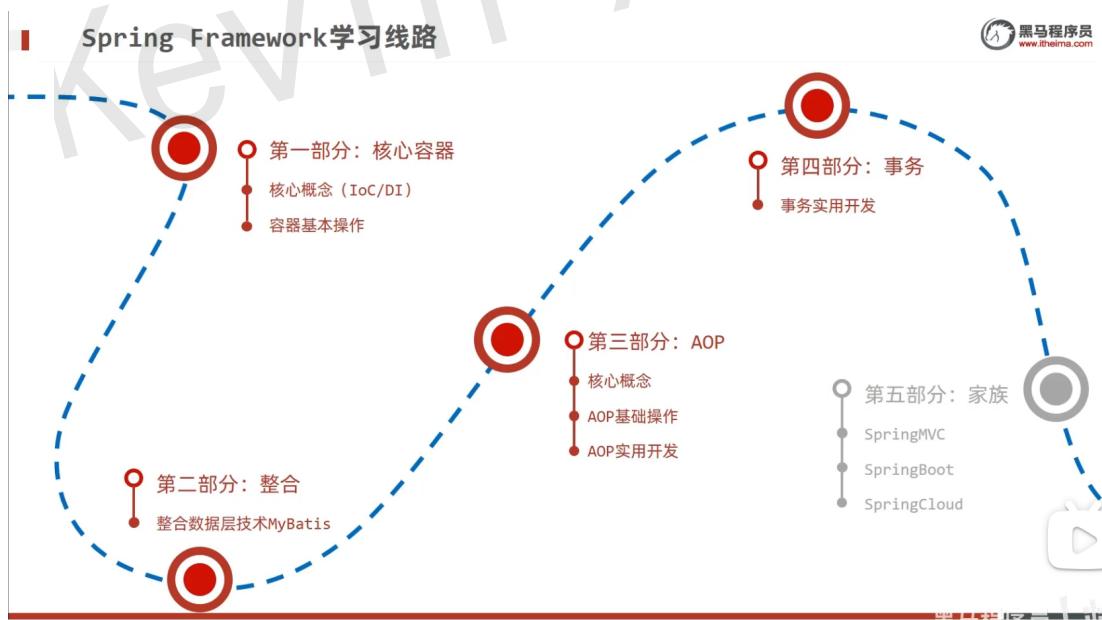
Spring

Spring相关概念

1. Spring Framework系统架构



2. 学习路线



3. IOC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。

举例：

业务层要用数据层的类对象，以前是自己new的，现在自己不new了，交给“别人”（即外部）来创建对象，这样外部，就反转控制了数据层对象的创建权，这种思想就是控制反转。

Spring和IOC的关系：

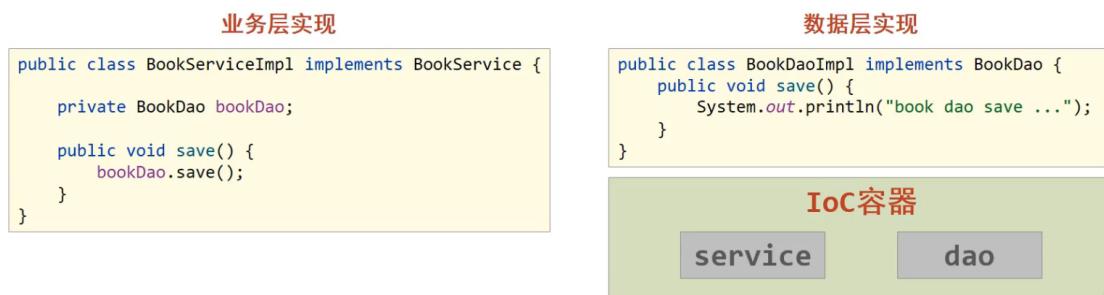
Spring技术对IOC思想进行了实现，提供了一个容器，称为“IOC容器”，用来充当IOC思想中的“外部”。

4. IOC容器

IOC容器负责对象的创建、初始化等一系列工作，其中包含了数据层和业务层的类对象。

Bean对象：

被创建或被管理的对象在IOC容器中统称为“Bean”，即IOC容器中放的就是一个个的Bean对象。



5. DI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。

举例：

业务层要用数据层的类对象，以前是自己new的，现在自己不new了，靠IOC容器注入进来即可使用，这种思想就是依赖注入。

哪些bean之间要建立依赖关系：

根据业务需求提前建立好关系，如业务层需要依赖数据层，service就要和dao建立依赖关系。

6. 概念小结

IOC和DI的设计目的就是充分解耦，具体来说即

- 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)
- 最终结果为：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。

入门案例(使用步骤)

IOC入门案例

1. 思路分析

(1)Spring是使用容器来管理bean对象的，那么管什么？

- 主要管理项目中所使用到的类对象，比如(Service和Dao)

(2)如何将被管理的对象告知IOC容器？

- 使用配置文件

(3)被管理的对象交给IOC容器，要想从容器中获取对象，就先得思考如何获取到IOC容器？

- Spring框架提供相应的接口

(4)IOC容器得到后，如何从容器中获取bean？

- 调用Spring框架提供对应接口中的方法

(5)使用Spring导入哪些坐标？

- 用别人的东西，就需要在pom.xml添加对应的依赖

2. 具体步骤

先准备要管理的类（接口）

创建BookService, BookServiceImpl, BookDao和BookDaoImpl四个类

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

IOC步骤：

(1) 在pom.xml导入Spring的坐标Spring-context

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.10.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

(2) 在resources下创建applicationContext.xml，配置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
                           http://www.springframework.org/schema/beans/spring-
                           beans.xsd">

    <!--bean标签表示配置bean
        id属性标示给bean起名字，id在同一个配置文件中不能重复
        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"/>

</beans>
```

(3) 获取IOC容器

使用Spring提供的接口完成IOC容器的创建，创建App类，编写main方法

```
public class App {  
    public static void main(String[] args) {  
        //获取IOC容器，ApplicationContext是接口，new的是实现类，实现  
        //类构造参数是配置bean的文件  
        ApplicationContext ctx = new  
        ClassPathXmlApplicationContext("applicationContext.xml");  
    }  
}
```

(4) 获取bean

```
public class App {  
    public static void main(String[] args) {  
        //获取IOC容器  
        ApplicationContext ctx = new  
        ClassPathXmlApplicationContext("applicationContext.xml");  
        //用getBean方法获取bean，入参是配置文件写的bean id  
        BookDao bookDao = (BookDao) ctx.getBean("bookDao");  
        //使用bean  
        bookDao.save();  
  
        BookService bookService = (BookService)  
        ctx.getBean("bookService");  
        bookService.save();  
    }  
}
```

DI入门案例

1. 思路分析

(1)要想实现依赖注入，必须要基于IOC管理Bean

- DI的入门案例要依赖于前面IOC的入门案例

(2)Service中使用new形式创建的Dao对象是否保留？

- 需要删除掉，最终要使用IOC容器中的bean对象

(3)Service中需要的Dao对象如何进入到Service中？

- 在Service中提供方法，让Spring的IOC容器可以通过该方法传入bean对象

(4)Service与Dao间的关系如何描述？

- 使用配置文件

2. 具体步骤

(1) 去除代码中new对象的行为，彻底解耦

```
public class BookServiceImpl implements BookService {  
    //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();  
    }  
}
```

(2) 删除new后，为该对象设置set方法

```
public class BookServiceImpl implements BookService {  
    //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();  
    }  
    //提供对应的set方法  
    public void setBookDao(BookDao bookDao) {  
        this.bookDao = bookDao;  
    }  
}
```

(3) 在applicationContext.xml修改配置完成注入

```
<?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标签标示配置bean  
        id属性标示给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">
    <!--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>

</beans>

```

说明

1. 由于在BookServiceImpl中注入bookDao对象，因此在bookService的bean中进行配置。
- 2.name属性指的是在实现类中，要使用setter注入的对象名。
- 3.ref属性表示用哪个bean去进行注入，即使用的是配置文件中bean的id。

IOC相关内容

bean基础配置

1. 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	<beans> <bean/> <bean></bean> </beans>
属性列表	id : 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 class : 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>

2. 别名配置

用name属性可以为bean配置别名

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>

3. 作用范围配置

用scope属性可以为bean配置作用范围。bean默认是单例的（不同对象指向的是同一个地址）。

singleton表示单例，prototype表示非单例。

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

bean设置为单例避免了对象的频繁创建与销毁，达到了bean对象的复用，性能高。

适合交给容器进行管理的对象有：表现层对象、业务层对象、数据层对象、工具对象。

```
<?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">

    <!--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 dao"
          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>
```

bean实例化(创建)

bean实例化就是bean对象的创建。bean的实例化有三种方法，使用不同的实例化方法配置的写法也会有所不同。默认的Spring框架中使用的是第一种构造方法实例化。

1. 构造方法实例化 (默认)

bean本质上就是对象，对象在new的时候会使用构造方法完成，那创建bean也是使用构造方法完成的。

准备一个BookDao和BookDaoImpl类

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

在创建bean对象时，实际上创建的是BookDaoImpl对象，因此Spring中bean的构造方法实例化等同于：

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

在上述的测试程序中，将构造函数改为private仍可正确执行，设置为有参不可执行，说明：Spring底层使用的是无参构造方法，且用的是反射原理（能获取到私有构造器）。

2. 静态工厂实例化

bean的实例化也可以用静态工厂实现。

首先研究工厂方式创建bean的方法：

(1) 准备一个OrderDao和OrderDaoImpl类

```
public interface OrderDao {  
    public void save();  
}  
  
public class OrderDaoImpl implements OrderDao {  
    public void save() {  
        System.out.println("order dao save ...");  
    }  
}
```

(2) 创建一个工厂类OrderDaoFactory并提供一个静态方法

```
//静态工厂创建对象  
public class OrderDaoFactory {  
    public static OrderDao getOrderDao(){  
        return new OrderDaoImpl();  
    }  
}
```

(3) 编写AppForInstanceOrder运行类，在类中通过工厂获取对象

```
public class AppForInstanceOrder {  
    public static void main(String[] args) {  
        //通过静态工厂创建对象  
        OrderDao orderDao = OrderDaoFactory.getOrderDao();  
        orderDao.save();  
    }  
}
```

使用静态工厂实例化依据的就是上述原理，若通过静态工厂方式实现bean的实例化，修改配置即可。

(1) class写的是工厂的全类名，factory-method写的是工厂中实际构造bean对象的方法。

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

(2) 使用从IOC容器中获取bean的方法

```
public class AppForInstanceOrder {
    public static void main(String[] args) {
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");

        OrderDao orderDao = (OrderDao)
ctx.getBean("orderDao");

        orderDao.save();

    }
}
```

3. 实例工厂与FactoryBean

- (1) 创建一个UserDaoFactoryBean的类，实现FactoryBean接口，重写接口的方法

```
public class UserDaoFactoryBean implements
FactoryBean<UserDao> {
    //代替原始实例工厂中创建对象的方法
    public UserDao getObject() throws Exception {
        return new UserDaoImpl();
    }
    //返回所创建类的Class对象
    public Class<?> getObjectType() {
        return UserDao.class;
    }
}
```

- (2) 在Spring的配置文件中进行配置

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

- (3) 从IOC容器中获取bean的方法不需要改变

bean生命周期

1. 概念

bean生命周期：bean对象从创建到销毁的整体过程。

bean生命周期控制是什么：在bean创建后到销毁前做一些事情。

2. 生命周期设置

具体的控制有两个阶段：

- bean创建之后，想要添加内容，比如用来初始化需要用到的资源。
- bean销毁之前，想要添加内容，比如用来释放用到的资源。

步骤1：添加初始化和销毁方法

针对这两个阶段，我们在BookDaoImpl类中分别添加两个方法，方法名任意

```
public class BookDaoImpl implements BookDao {  
    public void save() {  
        System.out.println("book dao save ...");  
    }  
    //表示bean初始化对应的操作  
    public void init(){  
        System.out.println("init...");  
    }  
    //表示bean销毁前对应的操作  
    public void destory(){  
        System.out.println("destory...");  
    }  
}
```

步骤2：配置生命周期

在配置文件添加配置，如下：

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

运行后发现init方法会执行，但destory方法不会执行，原因：Spring的IOC容器是运行在JVM中，main方法执行完后，JVM退出，这个时候IOC容器中的bean还没有来得及销毁就已经结束了。

3. 关闭容器方法

要让bean销毁前的方法得以执行，需要关闭IOC容器，IOC关闭有两种方法：

(1) close关闭容器

之前获取IOC容器时，用ApplicationContext来接收，该接口无close方法，但其实现类有close方法。因此用其实现类接收即可

```

public class AppForLifeCycle {
    public static void main( String[] args ) {
        ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
        BookDao bookDao = (BookDao) ctx.getBean("bookDao");
        bookDao.save();

        ctx.close();
    }
}

```

(2) 注册钩子关闭容器，在虚拟机退出前关闭容器

调用ctx的registerShutdownHook()方法

```

public class AppForLifeCycle {
    public static void main( String[] args ) {
        ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
        ctx.registerShutdownHook();

        BookDao bookDao = (BookDao) ctx.getBean("bookDao");
        bookDao.save();
    }
}

```

close()是在调用的时候关闭，registerShutdownHook()是在JVM退出前调用关闭，因此钩子写在哪里都行，而close必须写在bean使用后。

4. 用接口进行生命周期设置

Spring提供了两个接口来完成生命周期的控制，好处是可以不用再进行配置init-method和destroy-method。

```

public class BookServiceImpl implements BookService,
InitializingBean, DisposableBean {
    private BookDao bookDao;
    public void setBookDao(BookDao bookDao) { //优先于初始化操作执行
        this.bookDao = bookDao;
    }
    public void save() {
        System.out.println("book service save ...");
    }
}

```

```

        bookDao.save();
    }
    public void destroy() throws Exception {
        System.out.println("service destroy");
    }
    public void afterPropertiesSet() throws Exception { //初始化操作
        System.out.println("service init");
    }
}

```

5. 生命周期各阶段小结

(1) 初始化容器

- 1. 创建对象(内存分配)
- 2. 执行构造方法
- 3. 执行属性注入(set操作)
- 4. 执行bean初始化方法

(2) 使用bean

- 1. 执行业务操作

(3) 关闭/销毁容器

- 1. 执行bean销毁方法

DI相关内容

setter注入

1. 注入引用数据类型

需求：在bookServiceImpl对象中注入userDao

(1) 步骤1：声明属性并提供setter方法

在BookServiceImpl中声明userDao属性，并提供setter方法

```

public class BookServiceImpl implements BookService{
    private BookDao bookDao;
    private UserDao userDao;

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

```

```

    }

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

```

(2) 步骤2：配置文件中进行注入配置

在applicationContext.xml配置文件中使用property标签注入

```

<?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="userDao"
          class="com.itheima.dao.impl.UserDaoImpl"/>
    <bean id="bookService"
          class="com.itheima.service.impl.BookServiceImpl">
        <property name="bookDao" ref="bookDao"/>
        <property name="userDao" ref="userDao"/>
    </bean>
</beans>

```

2. 注入简单数据类型+String

需求：给BookDaoImpl注入一些简单数据类型的数据

(1) 步骤1：声明属性并提供setter方法

在BookDaoImpl类中声明对应的简单数据类型的属性，并提供对应的setter方法

```

public class BookDaoImpl implements BookDao {

    private String databaseName;
    private int connectionNum;

    public void setConnectionNum(int connectionNum) {
        this.connectionNum = connectionNum;
    }
}

```

```

public void setDatabaseName(String databaseName) {
    this.databaseName = databaseName;
}

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

```

(2) 步骤2：配置文件中进行注入配置

在applicationContext.xml配置文件中使用property标签注入

```

<?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">
        <property name="databaseName" value="mysql"/>
        <property name="connectionNum" value="10"/>
    </bean>
    <bean id="userDao"
          class="com.itheima.dao.impl.UserDaoImpl"/>
    <bean id="bookService"
          class="com.itheima.service.impl.BookServiceImpl">
        <property name="bookDao" ref="bookDao"/>
        <property name="userDao" ref="userDao"/>
    </bean>
</beans>

```

构造器注入

1. 注入引用数据类型

需求：将BookServiceImpl类中的bookDao修改成使用构造器的方式注入。

(1) 步骤1：删除setter方法并提供构造方法

在BookServiceImpl类中将bookDao的setter方法删除掉，并添加带有bookDao参数的构造方法

```

public class BookServiceImpl implements BookService{
    private BookDao bookDao;

    public BookServiceImpl(BookDao bookDao) {
        this.bookDao = bookDao;
    }

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

```

(2) 步骤2：配置文件中进行配置构造方式注入

在applicationContext.xml中配置

```

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

```

说明：

标签<constructor-arg>中

name属性对应的值为构造函数中方法形参的参数名，必须要保持一致。

ref属性指向的是spring的IOC容器中其他bean对象。

2. 注入多个引用数据类型

需求：在BookServiceImpl使用构造函数注入多个引用数据类型，比如userDao。

(1) 步骤1：提供多个属性的构造函数

在BookServiceImpl声明userDao并提供多个参数的构造函数

```

public class BookServiceImpl implements BookService{
    private BookDao bookDao;
    private UserDao userDao;

    public BookServiceImpl(BookDao bookDao,UserDao userDao) {
        this.bookDao = bookDao;
        this.userDao = userDao;
    }

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

```

(2) 步骤2：配置文件中配置多参数注入

在applicationContext.xml中配置注入

```

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

```

3. 注入多个简单数据类型

需求：在BookDaoImpl中，使用构造函数注入databaseName和connectionNum两个参数。

(1) 步骤1：添加多个简单属性并提供构造方法

修改BookDaoImpl类，添加构造方法

```

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);
    }
}

```

(2) 步骤2：配置完成多个属性构造器注入

在applicationContext.xml中进行注入配置

```

<?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">
        <constructor-arg name="databaseName" value="mysql"/>
        <constructor-arg name="connectionNum" value="666"/>
    </bean>
    <bean id="userDao"
          class="com.itheima.dao.impl.UserDaoImpl"/>
    <bean id="bookService"
          class="com.itheima.service.impl.BookServiceImpl">
        <constructor-arg name="bookDao" ref="bookDao"/>
        <constructor-arg name="userDao" ref="userDao"/>
    </bean>
</beans>

```

自动装配

1. 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。
2. 按类型自动装配

实现按照类型注入的配置

```
<?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"/>

    <!--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.BookServiceImpl"
          autowire="byType"/>

```

注意：

- (1) 需要注入属性的类中对应属性的setter方法不能省略。
- (2) 被注入的对象必须要被Spring的IOC容器管理。
- (3) 按照类型在Spring的IOC容器中如果找到多个对象，如上图有两个bean对象，会报错，此时需要用按名称装配。

3. 按名称自动注入

```

<?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="bookDao"
          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.BookServiceImpl"
          autowire="byName"/>

</beans>

```

这里的按名称注入指：根据类中要注入的属性，找到与其名字相同的bean对象进行装配注入。

集合注入

1. 环境准备

- (1) 项目中添加添加BookDao、BookDaoImpl类

```

public interface BookDao {
    public void save();
}

public class BookDaoImpl implements BookDao {

    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 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);
}

//setter....方法省略，自己使用工具生成
}

```

(2) resources下提供spring的配置文件， applicationContext.xml

```

<?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"/>

```

(3) 编写AppForDICollection运行类，加载Spring的IOC容器，并从中获取对应的bean对象

```

public class AppForDICollection {
    public static void main( String[] args ) {
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
        BookDao bookDao = (BookDao) ctx.getBean("bookDao");
        bookDao.save();
    }
}

```

2. 注入

在BookDaoImpl中进行注入，在bookDao的bean标签中使用进行注入。

(1) 注入数组类型数据

```
<property name="array">
    <array>
        <value>100</value>
        <value>200</value>
        <value>300</value>
    </array>
</property>
```

(2) 注入List类型数据

```
<property name="list">
    <list>
        <value>itcast</value>
        <value>itheima</value>
        <value>boxuegu</value>
        <value>chuanzhihui</value>
    </list>
</property>
```

(3) 注入Set类型数据

```
<property name="set">
    <set>
        <value>itcast</value>
        <value>itheima</value>
        <value>boxuegu</value>
        <value>boxuegu</value>
    </set>
</property>
```

(4) 注入Map类型数据

```
<property name="map">
    <map>
        <entry key="country" value="china"/>
        <entry key="province" value="henan"/>
        <entry key="city" value="kaifeng"/>
    </map>
</property>
```

(5) 注入Properties类型数据

```
<property name="properties">
    <props>
        <prop key="country">china</prop>
        <prop key="province">henan</prop>
        <prop key="city">kaifeng</prop>
    </props>
</property>
```

集合中要添加引用类型，只需要把`<value>`标签改成`<ref>`标签，这种方式用的比较少。

配置管理第三方bean

数据源对象管理

1. 环境准备

此前的bean都是类中的属性，假设要注入的bean来自于第三方，需要有不同的配置。

这里以管理数据库连接池druid为例：

pom.xml添加依赖

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

resources下添加spring的配置文件applicationContext.xml

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

</beans>
```

编写一个运行类App

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

2. 实现Druid管理

(1) 步骤1：导入druid的依赖

pom.xml中添加依赖

```
<dependency>  
    <groupId>com.alibaba</groupId>  
    <artifactId>druid</artifactId>  
    <version>1.1.16</version>  
</dependency>
```

(2) 步骤2：配置第三方bean

在applicationContext.xml配置文件中添加DruidDataSource 的配置

```
<?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">  
    <!--管理DruidDataSource对象-->  
    <bean id="dataSource"  
          class="com.alibaba.druid.pool.DruidDataSource">  
        <property name="driverClassName"  
                 value="com.mysql.jdbc.Driver"/>  
        <property name="url"  
                 value="jdbc:mysql://localhost:3306/spring_db"/>  
        <property name="username" value="root"/>  
        <property name="password" value="root"/>  
    </bean>  
</beans>
```

(3) 步骤3：从IOC容器中获取对应的bean对象

```

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);
    }
}

```

加载properties文件

进行第三方管理时，可以配置可以先写在properties文件中，再将其属性写到aoolicationContext.xml文件中。

- 开启context命名空间

```

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

```

- 使用context命名空间，加载指定properties文件

```
<context:property-placeholder location="jdbc.properties"/>
```

- 使用\${}读取加载的属性值

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

核心容器与总结

1. 创建容器

- 方式一：类路径加载配置文件

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
```

- 方式二：文件路径加载配置文件

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("D:\\\\applicationContext.xml");
```

- 加载多个配置文件

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("bean1.xml", "bean2.xml");
```

2. 获取bean

- 方式一：使用bean名称获取

```
BookDao bookDao = (BookDao) ctx.getBean("bookDao");
```

- 方式二：使用bean名称获取并指定类型

```
BookDao bookDao = ctx.getBean("bookDao", BookDao.class);
```

- 方式三：使用bean类型获取

```
BookDao bookDao = ctx.getBean(BookDao.class);
```

若使用方式三，要求容器中该类型的bean仅有一个。

3. 总结

(1) 容器

- 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。

(2) bean

<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
/ >	

(3) 依赖注入相关

<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"/>	构造器注入简单类型
<constructor-arg type="java.lang.String" index="3" value="WARN"/>	类型匹配与索引匹配
<property name="bookDao" ref="bookDao"/>	setter注入引用类型
<property name="userDao" ref="userDao"/>	
<property name="msg" value="WARN"/>	setter注入简单类型
<property name="names">	setter注入集合类型
<list>	list集合
<value>itcast</value>	集合注入简单类型
<ref bean="dataSource"/>	集合注入引用类型
</list>	
</property>	
</bean>	

IOC/DI注解开发

注解开发定义bean

1. 流程

(1) 步骤1：删除原XML配置

将配置文件中的`<bean>`标签删除掉

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

(2) 步骤2：使用@Component定义bean

在实现类上添加@Component注解，该注解不能添加在接口上，因为接口无法创建对象。

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

@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();
    }
}
```

(3) 步骤3：配置Spring的注解包扫描

为了让Spring框架能够扫描到写在类上的注解，需要在配置文件上进行包扫描。

```

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

```

base-package指定Spring框架扫描的包路径，它会扫描指定包及其子包中的所有类上的注解。

2. @Component注解

Spring提供@Component注解的三个衍生注解

- @Controller：用于表现层bean定义
- @Service：用于业务层bean定义
- @Repository：用于数据层bean定义

名称	@Component/@Controller/@Service/@Repository
类型	类注解
位置	类定义上方
作用	设置该类为spring管理的bean
属性	value（默认）：定义bean的id

纯注解开发模式

1. 流程

实现思路：将配置文件applicationContext.xml删除掉，使用类来替换。

(1) 步骤1：创建配置类

创建一个配置类SpringConfig。

```

public class SpringConfig {
}

```

(2) 步骤2：标识该类为配置类

在配置类上添加@Configuration注解，将其标识为一个配置类，替换applicationContext.xml。

```
@Configuration  
public class SpringConfig {  
}
```

(3) 步骤3：用注解替换包扫描配置

在配置类上添加包扫描注解@ComponentScan替换<context:component-scan base-package="" />。

```
@Configuration  
@ComponentScan("com.itheima")  
public class SpringConfig {  
}
```

(4) 创建运行类并执行

创建一个新的运行类AppForAnnotation，读取Spring核心配置文件初始化容器对象切换为读取Java配置类初始化容器对象。

```
public class AppForAnnotation {  
  
    public static void main(String[] args) {  
        ApplicationContext ctx = new  
        AnnotationConfigApplicationContext(SpringConfig.class);  
        BookDao bookDao = (BookDao) ctx.getBean("bookDao");  
        System.out.println(bookDao);  
        BookService bookService =  
        ctx.getBean(BookService.class);  
        System.out.println(bookService);  
    }  
}
```

2. @Configuration注解

名称	@Configuration
类型	类注解
位置	类定义上方
作用	设置该类为spring配置类
属性	value (默认) : 定义bean的id

3. @ComponentScan注解

名称	@ComponentScan
类型	类注解
位置	类定义上方
作用	设置spring配置类扫描路径，用于加载使用注解格式定义的bean
属性	value（默认）：扫描路径，此路径可以逐层向下扫描

bean作用范围与生命周期管理

1. 作用范围

单例还是非单例用注解@Scope设置

```

@Repository
//@Scope设置bean的作用范围
@Scope("prototype")
public class BookDaoImpl implements BookDao {

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

```

2. 生命周期管理

只需要在对应的方法上添加@PostConstruct 和 @PreDestroy 注解即可。

```

@Repository
public class BookDaoImpl implements BookDao {
    public void save() {
        System.out.println("book dao save ...");
    }
    @PostConstruct //在构造方法之后执行，替换 init-method
    public void init() {
        System.out.println("init ...");
    }
    @PreDestroy //在销毁方法之前执行，替换 destroy-method
    public void destroy() {
        System.out.println("destroy ...");
    }
}

```

3. 相关注解说明

@Scope

名称	@Scope
类型	类注解
位置	类定义上方
作用	设置该类创建对象的作用范围 可用于设置创建出的bean是否为单例对象
属性	value (默认) : 定义bean作用范围, ==默认值singleton (单例) , 可选值prototype (非单例) ==

@PostConstruct

名称	@PostConstruct
类型	方法注解
位置	方法上
作用	设置该方法为初始化方法
属性	无

@PreDestroy

名称	@PreDestroy
类型	方法注解
位置	方法上
作用	设置该方法为销毁方法
属性	无

注解开发依赖注入

1. 环境准备

pom.xml添加Spring的依赖

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

添加一个配置类 SpringConfig

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

添加 BookDao、BookDaoImpl、BookService、BookServiceImpl 类

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

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

public interface BookService {
    public void save();
}

@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();
    }
}
```

创建运行类 App

```
public class App {  
    public static void main(String[] args) {  
        AnnotationConfigApplicationContext ctx = new  
        AnnotationConfigApplicationContext(SpringConfig.class);  
        BookService bookService =  
        ctx.getBean(BookService.class);  
        bookService.save();  
    }  
}
```

2. 按照类型注入

在实现类的属性上添加@Autowired注解。自动装配基于暴力反射，因此不需要提供set方法。

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

如果有多个实现类，就需要按名称注入

先给两个Dao类分别起个名称，此时会根据名称匹配去找到名为"bookDao"的实现类进行注入。

```
@Repository("bookDao")  
public class BookDaoImpl implements BookDao {  
    public void save() {  
        System.out.println("book dao save ...");  
    }  
}  
@Repository("bookDao2")  
public class BookDaoImpl2 implements BookDao {  
    public void save() {  
        System.out.println("book dao save ...2");  
    }  
}
```

3. 按照名称注入

使用@Qualifier来指定注入哪个名称的bean对象。

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    @Qualifier("bookDao1") // @Qualifier必须配合@Autowired进行使用
    private BookDao bookDao;

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

4. 简单类型注入

使用@Value注解，将值写入注解的参数中就行了

```
@Repository("bookDao")
public class BookDaoImpl implements BookDao {
    @Value("itheima")
    private String name;

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

5. 注解读取properties文件

(1) 步骤1：resource下准备properties文件

jdbc.properties

```
name=itheima888
```

(2) 步骤2：使用注解加载properties配置文件

在配置类上添加@PropertySource注解

```

@Configuration
@ComponentScan("com.itheima")
@PropertySource("jdbc.properties")
public class SpringConfig {
}

```

(3) 步骤3：使用@Value读取配置文件中的内容

```

@Repository("bookDao")
public class BookDaoImpl implements BookDao {
    @Value("${name}")
    private String name;
    public void save() {
        System.out.println("book dao save ..." + name);
    }
}

```

//说明：
//如果读取的properties配置文件有多个，可以使用@PropertySource的属性来指定多个
@PropertySource({"jdbc.properties", "xxx.properties"})
//注解属性中不支持使用通配符*，运行会报错
//注解属性中可以把`classpath:`加上，代表从当前项目的根路径找文件
@PropertySource({"classpath:jdbc.properties"})

6. 依赖注入相关注解

@Autowired

名称	@Autowired
类型	属性注解 或 方法注解（了解） 或 方法形参注解（了解）
位置	属性定义上方 或 标准set方法上方 或 类set方法上方 或 方法形参前面
作用	为引用类型属性设置值

名称	@Autowired
属性	required: true/false, 定义该属性是否允许为null

@Qualifier

名称	@Qualifier
类型	属性注解 或 方法注解 (了解)
位置	属性定义上方 或 标准set方法上方 或 类set方法上方
作用	为引用类型属性指定注入的beanId
属性	value (默认) : 设置注入的beanId

@Value

名称	@Value
类型	属性注解 或 方法注解 (了解)
位置	属性定义上方 或 标准set方法上方 或 类set方法上方
作用	为 基本数据类型 或 字符串类型 属性设置值
属性	value (默认) : 要注入的属性值

@PropertySource

名称	@PropertySource
类型	类注解
位置	类定义上方
作用	加载properties文件中的属性值
属性	value (默认) : 设置加载的properties文件对应的文件名或文件名组成的数组

注解开发管理第三方bean

管理第三方bean

1. 流程

以管理数据库连接池druid为例：

(1) 导入对应的jar包

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.16</version>
</dependency>
```

(2) 在配置类中添加一个方法

注意该方法的返回值就是要创建的Bean对象类型

```
@Configuration
public class SpringConfig {
    public DataSource dataSource(){
        DruidDataSource ds = new DruidDataSource();
        ds.setDriverClassName("com.mysql.jdbc.Driver");
        ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
        ds.setUsername("root");
        ds.setPassword("root");
        return ds;
    }
}
```

(3) 在方法上添加@Bean注解

@Bean注解的作用是将方法的返回值制作作为Spring管理的一个bean对象

```

@Configuration
public class SpringConfig {
    @Bean
    public DataSource dataSource(){
        DruidDataSource ds = new DruidDataSource();
        ds.setDriverClassName("com.mysql.jdbc.Driver");
        ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
        ds.setUsername("root");
        ds.setPassword("root");
        return ds;
    }
}

```

在第(2)步里，把配置都写到SpringConfig里不利于阅读和代码管理，常见做法是写入新的配置类里：

```

public class JdbcConfig {
    @Bean
    public DataSource dataSource(){
        DruidDataSource ds = new DruidDataSource();
        ds.setDriverClassName("com.mysql.jdbc.Driver");
        ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
        ds.setUsername("root");
        ds.setPassword("root");
        return ds;
    }
}

```

然后在Spring的配置类中引入

```

@Configuration
//@ComponentScan("com.itheima.config")
@Import({JdbcConfig.class}) //有多个就需要用{}括起来
public class SpringConfig {
}

```

2. @Bean和@Import

@Bean

名称	@Bean
类型	方法注解

名称	@Bean
位置	方法定义上方
作用	设置该方法的返回值作为spring管理的bean
属性	value (默认) : 定义bean的id

@Import

名称	@Import
类型	类注解
位置	类定义上方
作用	导入配置类
属性	value (默认) : 定义导入的配置类类名, 当配置类有多个时使用数组格式一次性导入多个配置类

为第三方bean注入资源

当第三方bean创建过程中需要用到其他资源，此时就需要注入资源。

1. 简单数据类型注入

(1) 步骤1：类中提供属性

```
public class JdbcConfig {
    private String driver;
    private String url;
    private String userName;
    private String password;

    @Bean
    public DataSource dataSource(){
        DruidDataSource ds = new DruidDataSource();
        ds.setDriverClassName("com.mysql.jdbc.Driver");
        ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
        ds.setUsername("root");
        ds.setPassword("root");
        return ds;
    }
}
```

(2) 步骤2：使用@Value注解引入值

```
public class JdbcConfig {  
    @Value("com.mysql.jdbc.Driver")  
    private String driver;  
    @Value("jdbc:mysql://localhost:3306/spring_db")  
    private String url;  
    @Value("root")  
    private String userName;  
    @Value("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;  
    }  
}
```

这种方法可以通过注入properties文件里的属性解耦。

2. 引用数据类型注入

假设构建DataSource对象时需要用到BookDao对象。

(1) 步骤1：在SpringConfig中扫描BookDao

扫描的目的是让Spring能管理到BookDao,也就是说要让IOC容器中有一个bookDao对象

```
@Configuration  
@ComponentScan("com.itheima.dao")  
@Import({JdbcConfig.class})  
public class SpringConfig {  
}
```

(2) 步骤2：在JdbcConfig类的方法上添加形参即可。

```

@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;
}

```

配置开发注解开发对比

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

Spring整合

要用了再回来学

AOP

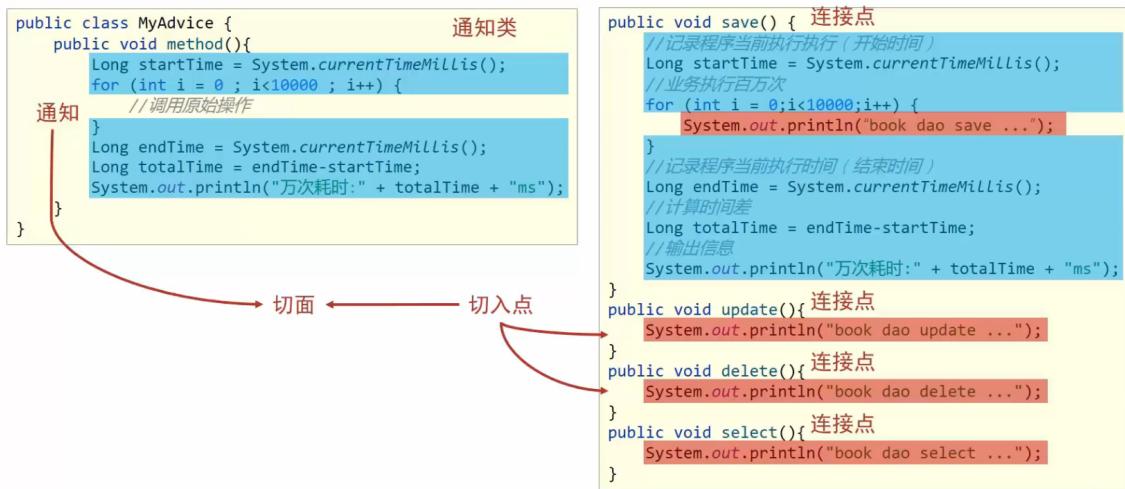
AOP简介&入门案例

1. 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。

AOP作用是在不惊动原始设计的基础上为其进行功能增强，即不改动原有代码基础上实现功能增强，符合Spring倡导的“无侵入式”理念。

2. AOP概念



以上图为例，蓝色部分表示用来进行功能增强的代码。

连接点：save()、update()、delete()、select()，即原始方法。

切入点：要追加功能的方法，图中update()、delete()中要追加蓝色部分代码功能，称这两个方法为切入点。

通知：抽取出的共有方法，也就是要追加的功能。

切面：绑定通知与切入点，描述通知与切入点的关系，即哪个切入点上执行哪些通知。

通知类：定义通知的类。

由于AOP的概念不止存在于Spring中，因此更一般的：

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，一个切入点可以描述一个具体方法，也可匹配多个方法。

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。

3. AOP入门案例

需求：在接口执行前输出当前系统时间。

思路分析：

- 1.导入坐标(pom.xml)
- 2.制作连接点(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)

环境准备

pom.xml添加Spring依赖

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

添加BookDao和BookDaoImpl类

```
public interface BookDao {
    public void save();
    public void update();
}

@Repository
public class BookDaoImpl implements BookDao {

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

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

创建Spring的配置类

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

编写App运行类

```
public class App {  
    public static void main(String[] args) {  
        ApplicationContext ctx = new  
AnnotationConfigApplicationContext(SpringConfig.class);  
        BookDao bookDao = ctx.getBean(BookDao.class);  
        bookDao.save();  
    }  
}
```

需求要求我们在不改变update方法的前提下让其具有打印系统时间的功能。

实现流程

(1) 步骤1：添加aspect依赖

在pom.xml中添加依赖

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

(2) 步骤2：定义接口和实现类

如环境准备的那样，不需要改变。

(3) 步骤3：定义通知类和通知

通知就是将共性功能抽取出来后形成的方法，在这个案例中共性功能指的就是当前系统时间的打印。

```
public class MyAdvice {  
    public void method(){  
        System.out.println(System.currentTimeMillis());  
    }  
}
```

(4) 步骤4：定义切入点

BookDaolmpl中有两个方法，分别是save和update，我们要增强的是update方法，则update方法需要设置为切入点。

切入点定义：写一个私有方法，无形参、无方法体、无返回值，在上面加上@Pointcut注解，里面写入"execution(void com.itheima.dao.BookDao.update())"。这里void表示要设置为切入点的update方法返回值为void，后面写update方法所在的全类名。

```
public class MyAdvice {  
    @Pointcut("execution(void  
com.itheima.dao.BookDao.update())")  
    private void pt(){  
  
        public void method(){  
            System.out.println(System.currentTimeMillis());  
        }  
    }  
}
```

(5) 步骤5：制作切面

用通知类型来绑定共性功能和切入点之间的关系。看method方法需要在切入点的什么位置执行，若在切入点之前执行即为@Before。

```
public class MyAdvice {  
    @Pointcut("execution(void  
com.itheima.dao.BookDao.update())")  
    private void pt(){  
  
        @Before("pt()")  
        public void method(){  
            System.out.println(System.currentTimeMillis());  
        }  
    }  
}
```

(6) 步骤6：将通知类配给容器并标识其为切面类

在通知类上添加注解@Aspect，让Spring扫描到通知类后将其当做AOP来处理。

```
@Component  
@Aspect  
public class MyAdvice {  
    @Pointcut("execution(void  
com.itheima.dao.BookDao.update())")  
    private void pt(){  
  
        @Before("pt()")  
        public void method(){  
            System.out.println(System.currentTimeMillis());  
        }  
    }  
}
```

(7) 步骤7：开启注解格式AOP功能，让Spring对AOP注解驱动支持

在配置类上添加注解，告诉Spring程序中有用注解开发的AOP。

```

@Configuration
@ComponentScan("com.itheima")
@EnableAspectJAutoProxy
public class SpringConfig {
}

```

(8) 步骤8：运行程序进行验证

```

public class App {
    public static void main(String[] args) {
        ApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);
        BookDao bookDao = ctx.getBean(BookDao.class);
        bookDao.update(); //此时执行update方法也会打印系统时间
    }
}

```

4. 相关注解

@EnableAspectJAutoProxy

名称	@EnableAspectJAutoProxy
类型	配置类注解
位置	配置类定义上方
作用	开启注解格式AOP功能

@Aspect

名称	@Aspect
类型	类注解
位置	切面类定义上方
作用	设置当前类为AOP切面类

@Pointcut

名称	@Pointcut
类型	方法注解
位置	切入点方法定义上方

名称	@Pointcut
作用	设置切入点方法
属性	value (默认) : 切入点表达式

@Before

名称	@Before
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前运行

AOP工作流程

1. 工作流程

- (1) Spring容器启动：加载需要被增强的类、通知类
- (2) 读取所有切面位置中的切入点：指定义且被使用了的切入点，如下图中pt()被使用，而ptx()并未被使用。因此只会读取pt()。

```

@Component
@Aspect
public class MyAdvice {
    @Pointcut("execution(void com.itheima.dao.BookDao.save())")
    private void ptx(){}

    @Pointcut("execution(void com.itheima.dao.BookDao.update())")
    private void pt(){}           ↴

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

```

- (3) 初始化bean，判断bean对应的类中的方法是否匹配到任意切入点
 - 匹配失败，创建原始对象
 - 匹配成功，创建原始对象的代理对象
- (4) 获取bean执行方法

- 获取的bean是原始对象时，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

2. 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。

解释：

目标对象就是要增强的类[如:BookServiceImpl类]对应的对象，也叫原始对象，不能说它不能运行，只能说它在运行的过程中对于要增强的内容是缺失的。

SpringAOP是在不改变原有设计(代码)的前提下对其进行增强的。SpringAOP的底层采用的是代理模式实现的，所以要对原始对象进行增强，就需要对原始对象创建代理对象，在代理对象中的方法把通知[如:MyAdvice中的method方法]内容加进去，就实现了增强，这就是我们所说的代理(Proxy)。

AOP切入点表达式

1. 切入点表达式语法格式

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，控制给哪个方法加AOP。

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)

举例：

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

- execution：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
- public：访问修饰符，还可以是public, private等，可以省略
- User：返回值，写返回值类型
- com.itheima.service：包名，多级包使用点连接，可以用接口的包也可以用实现类的包
- UserService：类/接口名称
- findById：方法名

- int: 参数，直接写参数的类型，多个类型用逗号隔开

- 异常名：方法定义中抛出指定异常，可以省略

2. 通配符

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（出现*则必须有，如在形参列表中则必须有形参）

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

表示匹配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(..))
```

+: 专用于匹配子类类型

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

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

```
execution(void com.itheima.dao.BookDao.update())
```

匹配接口，能匹配到

```
execution(void com.itheima.dao.impl.BookDaoImpl.update())
```

匹配实现类，能匹配到

```
execution(* com.itheima.dao.impl.BookDaoImpl.update())
```

返回值任意，能匹配到

```
execution(* com.itheima.dao.impl.BookDaoImpl.update(*))
```

返回值任意，但是update方法必须要有一个参数，无法匹配，要想匹配需要在update接口和实现类添加参数

```
execution(void com.*.*.*.*.update())
```

返回值为void，com包下的任意包三层包下的任意类的update方法，匹配到的是实现类，能匹配

```
execution(void com.*.*.*.update())
```

返回值为void，com包下的任意两层包下的任意类的update方法，匹配到的是接口，能匹配

```
execution(void *..update())
```

返回值为void，方法名是update的任意包下的任意类，能匹配

```

execution(* *..*(..))
匹配项目中任意类的任意方法，能匹配，但是不建议使用这种方式，影响范围广

execution(* *..u*(..))
匹配项目中任意包任意类下只要以u开头的方法，update方法能满足，能匹配

execution(* *..*e(..))
匹配项目中任意包任意类下只要以e结尾的方法，update和save方法能满足，能匹配

execution(void com..*(..))
返回值为void, com包下的任意包任意类任意方法，能匹配，*代表的是方法

execution(* com.itheima.*.*Service.find*(..))
将项目中所有业务层方法的以find开头的方法匹配

execution(* com.itheima.*.*Service.save*(..))
将项目中所有业务层方法的以save开头的方法匹配

```

3. 书写技巧

- 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**

AOP通知类型

1. 类型介绍

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通知类型分为五种：

- (1) 前置通知：追加功能到方法执行前。
- (2) 后置通知：追加功能到方法执行后，不管方法执行的过程中有没有抛出异常都会执行。
- (3) 返回后通知（了解）：追加功能到方法执行后，只有方法正常执行结束后才进行，如果方法执行抛出异常，返回后通知将不会被添加。
- (4) 抛出异常后通知（了解）：追加功能到方法抛出异常后，只有方法执行出异常才进行。
- (5) 环绕通知（重点）：环绕通知功能比较强大，它可以追加功能到方法执行的前后，这也是比较常用的方式，它可以实现其他四种通知类型的功能。

2. 前置通知

```

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

    @Before("pt()")
    //此处也可以写成 @Before("MyAdvice.pt()"),不建议
    public void before() {
        System.out.println("before advice ...");
    }
}

```

3. 后置通知

```

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

    @Before("pt()")
    public void before() {
        System.out.println("before advice ...");
    }
    @After("pt()")
    public void after() {
        System.out.println("after advice ...");
    }
}

```

4. 返回后通知

名称	@AfterReturning
类型	方法注解
位置	通知方法定义上方

名称	@AfterReturning
作用	设置当前通知方法与切入点之间绑定关系，当前通知方法在原始切入点方法正常执行完毕后执行

5. 抛出异常后通知

名称	@AfterThrowing
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

6. 环绕通知

```

@Component
@Aspect
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() {}

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

```

说明

1. 方法抛出异常原因：不确定调用时原始操作是否会产生异常
2. `ProceedingJoinPoint`参数能对原始方法进行调用。
3. `pjp.proceed()`进行调用并且能获得原始操作的返回值。
4. 返回的是`Object`, 因为`Object`类型更通用。
5. 如果没有`pjp.proceed()`, 将跳过对原始操作的调用。
6. 如果原始操作无返回值, 可以不接收。

业务层接口执行效率

1. 案例需求：测量业务层执行效率

```
@Component
@Aspect
public class ProjectAdvice {
    //配置业务层的所有方法
    @Pointcut("execution(* com.itheima.service.*Service.*(..))")
    private void servicePt(){}
    // @Around("ProjectAdvice.servicePt()") 可以简写为下面的方式
    @Around("servicePt()")
    public void runSpeed(ProceedingJoinPoint pjp){
        //获取执行签名信息，签名信息可以理解为封装了该次执行过程
        Signature signature = pjp.getSignature();
        //通过签名获取执行操作名称(接口名)
        String className = signature.getDeclaringTypeName();
        //通过签名获取执行操作名称(方法名)
        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");
    }
}
```

AOP通知获取切入点方法数据

1. 概述

目前的AOP仅仅是在原始方法前后追加一些操作。可以从原始操作中获取数据，并在通知中使用这些操作。

通知能从原始方法中可以获得的数据分为三类：原始方法的形参、原始方法的返回值、原始方法的异常。

- 获取切入点方法的参数，所有的通知类型都可以获取参数
 - JoinPoint：适用于前置、后置、返回后、抛出异常后通知
 - ProceedingJoinPoint：适用于环绕通知
- 获取切入点方法的返回值，前置和抛出异常后通知是没有返回值，后置通知可有可无，所以不做研究
 - 返回后通知
 - 环绕通知
- 获取切入点方法运行异常信息，前置和返回后通知是不会有，后置通知可有可无，所以不做研究
 - 抛出异常后通知
 - 环绕通知

2. 获取切入点形参

(1) 非环绕通知获取形参

使用JoinPoint的方式获取，这种获取方式适用于前置、后置、返回后、抛出异常后通知。

```
@Component  
@Aspect  
public class MyAdvice {  
    @Pointcut("execution(*  
com.itheima.dao.BookDao.findName(..))")  
    private void pt(){  
  
        @Before("pt()")  
        public void before(JoinPoint jp)  
            Object[] args = jp.getArgs(); //接收所有的切入点形参  
            System.out.println(Arrays.toString(args));  
            System.out.println("before advice ...");  
    }  
    //...其他的略  
}
```

(2) 环绕通知获取形参

ProceedingJoinPoint是JoinPoint类的子类，所以对于ProceedingJoinPoint类中也会有对应的getArgs()方法。

```
@Component
```

```

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

    @Around("pt()")
    public Object around(ProceedingJoinPoint pjp) throws
Throwable {
        Object[] args = pjp.getArgs();
        System.out.println(Arrays.toString(args));
        Object ret = pjp.proceed();
        return ret;
    }
    //其他的略
}

```

pjp.proceed()方法里面也可以传入一个Object类型的数组，可以用来在调用原始方法之前修改原始方法的参数。

```

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

    @Around("pt()")
    public Object around(ProceedingJoinPoint pjp) throws
Throwable{
        Object[] args = pjp.getArgs();
        System.out.println(Arrays.toString(args));
        args[0] = 666;
        Object ret = pjp.proceed(args); //调用时切入点方法的形参已
发生改变
        return ret;
    }
    //其他的略
}

```

有了这个特性后，我们就可以在环绕通知中对原始方法的参数进行拦截过滤，避免由于参数的问题导致程序无法正确运行，保证代码的健壮性。

3. 获取切入点返回值

(1) 环绕通知获取返回值

ret就是返回值，可以直接获取，也可以在获取后修改。

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

    @Around("pt()")
    public Object around(ProceedingJoinPoint pjp) throws
Throwable{
        Object[] args = pjp.getArgs();
        System.out.println(Arrays.toString(args));
        args[0] = 666;
        Object ret = pjp.proceed(args);
        return ret;
    }
    //其他的略
}
```

(2) 返回后通知获取返回值

需要在返回后通知传入形参表示接收返回值。

同时需要修改@AfterReturning，注解的value属性设置切入点表达式，returning属性写上通知中传入的形参名。表示如果原始方法有返回值，把这个返回值装入通知的形参中。

注意：若返回后通知同时需要传入JoinPoint，需要把JoinPoint放在形参首位。

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

    @AfterReturning(value = "pt()", returning = "ret")
    public void afterReturning(Object ret) {
        System.out.println("afterReturning advice ..."+ret);
    }
    //其他的略
}
```

4. 获取切入点异常

(1) 环绕通知获取异常

之前是直接抛出异常，现在只需要将异常捕获，就可以获取到原始方法的异常信息了。

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

    @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 throwable){
            throwable.printStackTrace();
        }
        return ret;
    }
    //其他的略
}
```

在catch方法中就可以获取到异常，至于获取到异常以后该如何处理，这个就和你的业务需求有关了。

(2) 抛出异常后通知获取异常

类似 返回后通知获取返回值的格式。

```

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

    @AfterThrowing(value = "pt()", throwing = "t")
    public void afterThrowing(Throwable t) {
        System.out.println("afterThrowing advice ..."+t);
    }
    //其他的略
}

```

AOP总结

1. 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

2. 切入点表达式: 表达式、通配符。

3. 通知类型: 前置、后置、环绕、返回后、抛出异常后。

4. 通知中获取原始方法参数: 获取切入点的入参、返回值、异常。

Spring事务

事务简介&事务管理案例

1. Spring事务

作用: 在数据层或业务层保障一系列的数据库操作同成功同失败。

Spring为事务管理提供了一个平台事务管理器PlatformTransactionManager。

```
public interface PlatformTransactionManager{  
    void commit(TransactionStatus status) throws TransactionException;  
    void rollback(TransactionStatus status) throws TransactionException;  
}
```

```
public class DataSourceTransactionManager {  
    ....  
}
```

commit用来提交事务， rollback用来回滚事务。

PlatformTransactionManager是一个接口， DataSourceTransactionManager是一个实现类， 只需要给它一个DataSource对象， 它就可以帮你去在业务层管理事务。

2. 事务管理案例

需求：实现任意两个账户间转账操作

需求微缩：A账户减钱，B账户加钱

分析：

- ①：数据层提供基础操作，指定账户减钱（outMoney），指定账户加钱（inMoney）
- ②：业务层提供转账操作（transfer），调用减钱与加钱的操作
- ③：提供2个账号和操作金额执行转账操作
- ④：基于Spring整合MyBatis环境搭建上述操作

环境准备（具体实现，配置类需要另外配置）

根据表创建模型类

```
public class Account implements Serializable {  
  
    private Integer id;  
    private String name;  
    private Double money;  
    //setter...getter...toString...方法略  
}
```

创建Dao接口

```
public interface AccountDao {  
  
    @Update("update tbl_account set money = money + #{money}  
    where name = #{name}")  
    void inMoney(@Param("name") String name, @Param("money")  
    Double money);  
  
    @Update("update tbl_account set money = money - #{money}  
    where name = #{name}")  
    void outMoney(@Param("name") String name, @Param("money")  
    Double money);  
}
```

创建Service接口和实现类

```
public interface AccountService {  
    /**  
     * 转账操作  
     * @param out 传出方  
     * @param in 转入方  
     * @param money 金额  
     */  
    public void transfer(String out, String in, Double money);  
}  
  
@Service  
public class AccountServiceImpl implements AccountService {  
  
    @Autowired  
    private AccountDao accountDao;  
  
    public void transfer(String out, String in, Double money) {  
        accountDao.outMoney(out, money);  
        accountDao.inMoney(in, money);  
    }  
}
```

假设现在实现类出现异常

```
@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;

    public void transfer(String out, String in, Double money) {
        accountDao.outMoney(out, money);
        int i = 1/0;
        accountDao.inMoney(in, money);
    }

}
```

因此需要为转账和收款设置事务，保证两个操作的一致性。

以下是事务管理的具体流程

(1) 步骤1：在业务层接口上或实现类上添加Spring事务管理

```
public interface AccountService {
    @Transactional
    public void transfer(String out, String in, Double money);
}
```

说明

@Transactional可以写在接口类上、接口方法上、实现类上和实现类方法上

1. 写在接口类上，该接口的所有实现类的所有方法都会有事务
2. 写在接口方法上，该接口的所有实现类的该方法都会有事务
3. 写在实现类上，该类中的所有方法都会有事务
4. 写在实现类方法上，该方法上有事务
5. ==建议写在实现类或实现类的方法上==（有待商榷）

(2) 步骤2：在JdbcConfig类中配置事务管理器

```
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;
    }

    //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) 步骤3：开启注解式事务驱动

在SpringConfig的配置类中开启

```

@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 {
}

```

3. 事务管理注解

@EnableTransactionManagement

名称	@EnableTransactionManagement
类型	配置类注解
位置	配置类定义上方
作用	设置当前Spring环境中开启注解式事务支持

@Transactional

名称	@Transactional
类型	接口注解 类注解 方法注解
位置	业务层接口上方 业务层实现类上方 业务方法上方
作用	为当前业务层方法添加事务 (如果设置在类或接口上方则类或接口中所有方法均添加事务)

事务角色

1. 目的：事务角色可以说明Spring的事务是如何实现的。

2. 事务角色

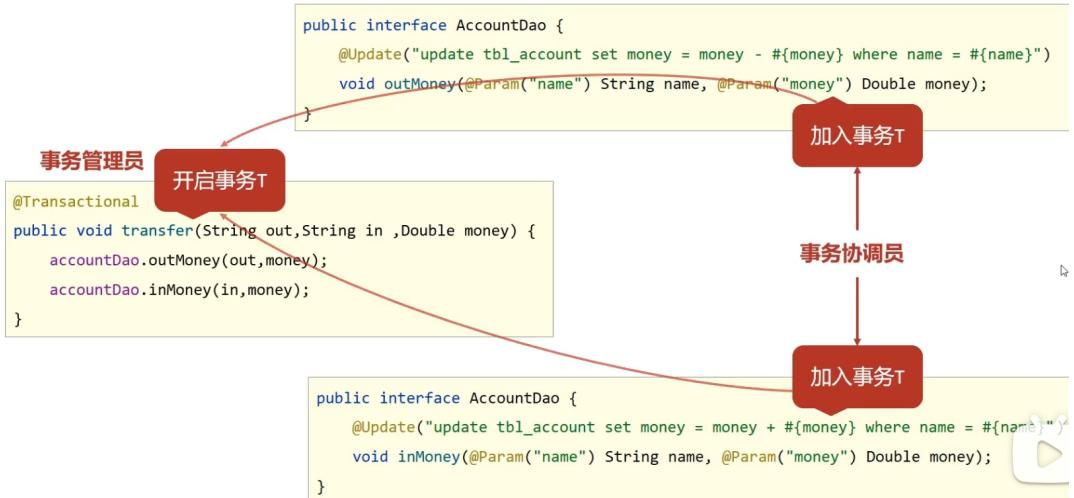
事务管理员：发起事务方，在Spring中通常指代业务层开启事务的方法。

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。

在案例中，初始有三个事务：

```
public interface AccountDao {  
    @Update("update tbl_account set money = money - #{money} where name = #{name}")  
    void outMoney(@Param("name") String name, @Param("money") Double money);  
}  
  
@Transactional  
public void transfer(String out, String in, Double money) {  
    accountDao.outMoney(out, money);  
    accountDao.inMoney(in, money);  
}  
  
public interface AccountDao {  
    @Update("update tbl_account set money = money + #{money} where name = #{name}")  
    void inMoney(@Param("name") String name, @Param("money") Double money);  
}
```

开启事务后，三个事务变成了一个，会同时回滚：



事务属性

1. 事务配置

配置项都能写在@Transactional注解中

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

rollbackFor: 指定遇到某一类异常回滚事务。不设置的情况下，程序只有在遇到 RuntimeException异常和Error异常时才会回滚，否则不会回滚。配置这个属性能选择要进行回滚的异常类型。

rollbackForClassName: 效果同上，只是设置属性时传入的是异常名而不是字节码。

如下，使用rollbackFor属性来设置出现IOException异常不回滚

```

@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;
    @Transactional(rollbackFor = {IOException.class})
    public void transfer(String out, String in, Double money)
    throws IOException{
        accountDao.outMoney(out, money);
        //int i = 1/0; //这个异常事务会回滚
    }
}

```

```
        if(true){  
            throw new IOException(); //这个异常事务就不会回滚  
        }  
        accountDao.inMoney(in,money);  
    }  
  
}
```

2. 事务传播行为

案例：基于转账操作案例添加日志模块，实现数据库中记录日志，要求无论转账操作是否成功，均进行转账操作的日志留痕。

环境准备

添加LogDao接口

```
public interface LogDao {  
    @Insert("insert into tbl_log (info,createDate) values(#  
{info},now())")  
    void log(String info);  
}
```

添加LogService接口与实现类

```
public interface LogService {  
    void log(String out, String in, Double money);  
}  
@Service  
public class LogServiceImpl implements LogService {  
  
    @Autowired  
    private LogDao logDao;  
    @Transactional  
    public void log(String out, String in, Double money) {  
        logDao.log("转账操作由"+out+"到"+in+", 金额: "+money);  
    }  
}
```

在转账的业务中添加记录日志

```
public interface AccountService {  
    /**  
     * 转账操作  
     * @param out 传出方  
     * @param in 转入方
```

```

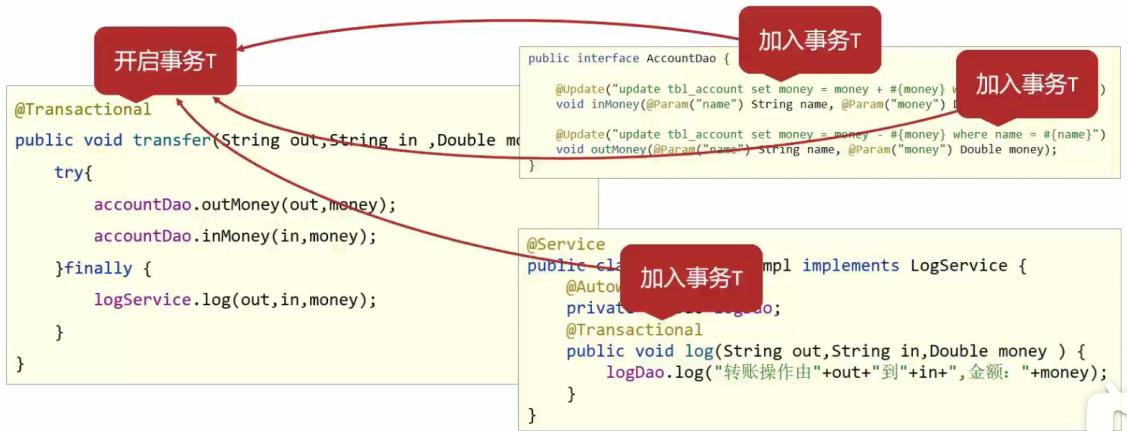
    * @param money 金额
    */
    //配置当前接口方法具有事务
    public void transfer(String out, String in, Double
    money) throws IOException ;
}

@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;
    @Autowired
    private LogService logService;
    @Transactional
    public void transfer(String out, String in, Double money) {
        try{
            accountDao.outMoney(out,money);
            accountDao.inMoney(in,money);
        }finally {
            logService.log(out,in,money);
        }
    }
}

```

按上述代码并不能满足需求，因为日志事务和转账收款事务都被加入到同一个事务中，失败时同时回滚。



为了让日志不回滚，需要让日志实现类单独成为一个事务，而不会被加入到transfer方法的事务中。

事务传播行为：事务协调员对事务管理员所携带事务的处理态度。propagation属性可以设置事务传播行为。

```

@Service
public class LogServiceImpl implements LogService {

    @Autowired
    private LogDao logDao;
    //propagation设置事务属性：传播行为设置为当前操作需要新事务
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void log(String out, String in, Double money) {
        logDao.log("转账操作由" + out + "到" + in + "，金额：" + money);
    }
}

```

propagation属性是一个枚举类，其枚举值如下：

传播属性	事务管理员	事务协调员
REQUIRED (默认)	开启T	加入T
	无	新建T2
REQUIRES_NEW	开启T	新建T2
	无	新建T2
SUPPORTS	开启T	加入T
	无	无
NOT_SUPPORTED	开启T	无
	无	无
MANDATORY	开启T	加入T
	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	

SpringMVC

概述与入门案例

1. SpringMVC概述

SpringMVC隶属于Spring框架的一部分，是一种表现层框架技术，用于进行表现层功能开发。

SpringMVC对Servlet进行了封装，可以认为是Servlet的代替。

案例制作与使用流程

1. 步骤1：导入jar包

添加SpringMVC和Servlet的依赖，加入Tomcat相关配置信息。

```
<?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>springmvc_01_quickstart</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>

  <dependencies>
    <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>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.1</version>
        <configuration>
          <port>80</port>
          <path>/</path>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>

```

provided代表的是该包只在编译和测试的时候用，运行的时候无效直接使用tomcat中的，就避免冲突。

2. 步骤2：创建Controller类

具体分为

- (1) 使用@Controller定义bean。
- (2) @RequestMapping设置当前操作的访问路径。
- (3) @ResponseBody，意思是把方法里返回的东西整体作为响应给前端的内容，即设置当前操作的返回值类型。

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

3. 步骤3：SpringMVC配置

创建SpringMvcConfig，作用是加载bean。

```
@Configuration  
@ComponentScan("com.itheima.controller") //扫描的是Controller类所在的包  
public class SpringMvcConfig {  
}
```

4. 步骤4：定义一个Servlet容器启动的配置类（SpringMVC专用）

作用是启动服务器时，能加载SpringMVC的配置，替换了原先的web.xml文件。

```
public class ServletContainersInitConfig extends  
AbstractDispatcherServletInitializer {  
    //加载springmvc配置类  
    protected webApplicationContext  
createServletApplicationContext() {  
        //初始化WebApplicationContext对象  
        AnnotationConfigWebApplicationContext ctx = new  
AnnotationConfigWebApplicationContext();  
        //加载指定配置类  
        ctx.register(SpringMvcConfig.class);  
        return ctx;  
    }  
  
    //设置由springmvc控制器处理的请求映射路径  
    protected String[] getServletMappings() {
```

```

        return new String[]{"//"};
    }

    //加载spring容器配置类
    protected WebApplicationContext
createRootApplicationContext() {
    return null;
}
}

```

5. 相关注解

@Controller

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

@RequestMapping

名称	@RequestMapping
类型	类注解或方法注解
位置	SpringMVC控制器类或方法定义上方
作用	设置当前控制器方法请求访问路径
相关属性	value(默认), 请求访问路径

@ResponseBody

名称	@ResponseBody
类型	类注解或方法注解
位置	SpringMVC控制器类或方法定义上方
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

工作流程解析

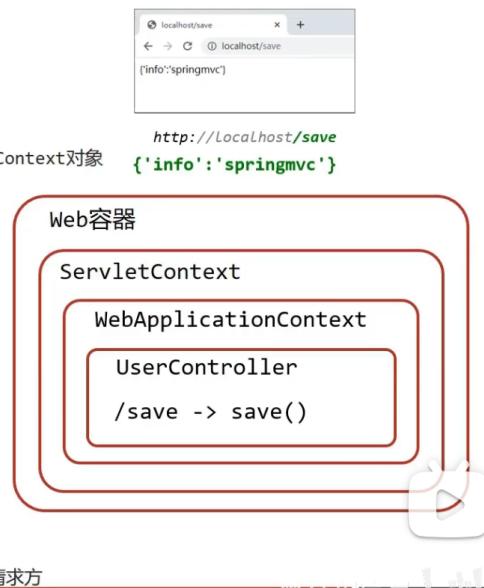
入门案例工作流程分析

启动服务器初始化过程

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

单次请求过程

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



bean加载控制

1. 问题分析

在一组文件中，SpringMVC相关的bean只有表现层的bean，其他如service的bean、dao层的bean等都是Spring控制的。目标是控制Spring不去加载SpringMVC的bean。

2. 解决方案：加载Spring控制的bean的时候排除掉SpringMVC控制的bean。

具体来说：

SpringMVC相关bean加载控制与之前相同。

Spring相关bean加载控制

(1) 方式1：排除掉Controller包的的bean。该方法较为复杂，在Springboot源码中会用到。

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

excludeFilters属性：设置扫描加载bean时，排除的过滤规则。

type属性：设置排除规则，当前使用按照bean定义时的注解类型进行排除。

- ANNOTATION: 按照注解排除
- ASSIGNABLE_TYPE:按照指定的类型过滤
- ASPECTJ:按照Aspectj表达式排除，基本上不会用
- REGEX:按照正则表达式排除
- CUSTOM:按照自定义规则排除

classes属性：设置排除的具体注解类，当前设置排除@Controller定义的bean。

(2) 方式2：Spring加载的bean设定扫描范围为精确范围，即具体到哪个包。实际应用都用这种方式。

```
@Configuration
@ComponentScan({"com.itheima.service", "comitheima.dao"})
public class SpringConfig { }
```

3. Tomcat加载配置类简化

在tomcat服务器启动加载，需要修改ServletContainersInitConfig

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

对于上述的配置方式，Spring还提供了一种更简单的配置方式，可以不用再去创建AnnotationConfigWebApplicationContext对象，不用手动register对应的配置类。

```
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[]{"/"};  
    }  
}
```

4. @ComponentScan

名称	@ComponentScan
类型	类注解
位置	类定义上方
作用	设置spring配置类扫描路径，用于加载使用注解格式定义的bean
相关属性	excludeFilters:排除扫描路径中加载的bean,需要指定类别(type)和具体项(classes) includeFilters:加载指定的bean，需要指定类别(type)和具体项(classes)

请求与响应

请求路径

```
@Controller  
@RequestMapping("/user")  
public class UserController {  
  
    @RequestMapping("/save") //访问路径由拼接所得，/user/save  
    @ResponseBody
```

```
public String save(){
    System.out.println("user save ...");
    return "{\"module\":\"user save\"}";
}

@RequestMapping("/delete")
@ResponseBody
public String save(){
    System.out.println("user delete ...");
    return "{\"module\":\"user delete\"}";
}

}

@Controller
@RequestMapping("/book")
public class BookController {

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

参数传递

指定url里传递的参数，后端如何在Controller层接收。

1. 普通参数

(1) 当接收的形参变量名和url地址带的参数一致，直接用形参接收即可

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

后端接收参数：

```
@Controller  
public class UserController {  
  
    @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': 'commonParam' }";  
    }  
}
```

(2) 当接收的形参变量名和url地址带的参数不一致，使用@RequestParam注解

```
http://localhost/commonParamDifferentName?name=张三&age=18
```

后端接收参数：

```
@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' }";  
}
```

2. 实体类数据

如果url里的参数比较多，可以定义一个实体类来接收这些参数，要求实体类里的属性名和url地址带的参数名都能对应上。

此时需要使用前面准备好的POJO类，先来看下User

```
public class User {  
    private String name; //与url里的传参名相同  
    private int age;  
    //setter...getter...略  
}
```

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

后端接收参数：

```
//POJO参数：请求参数与形参对象中的属性对应即可完成参数传递
@RequestMapping("/pojoParam")
@ResponseBody
public String pojoParam(User user){
    System.out.println("pojo参数传递 user ==> "+user);
    return "{\"module\":\"pojo param\"}";
}
```

3. 嵌套实体类数据

如果POJO对象中嵌套了其他的POJO类，如

```
public class Address {
    private String province;
    private String city;
    //setter...getter...略
}

public class User {
    private String name;
    private int age;
    private Address address;
    //setter...getter...略
}
```

依然要求实体类里的属性名和url地址带的参数名都能对应上

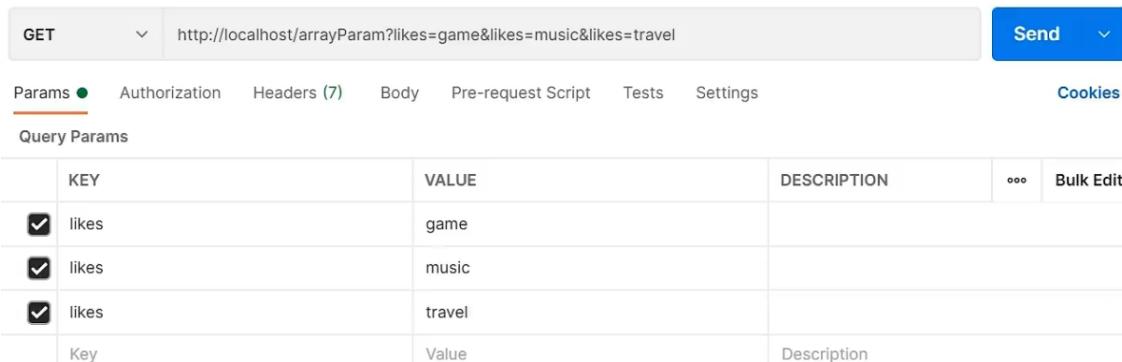
KEY	VALUE	DESCRIPTION	Bulk Edit
name	itcast		x
age	15		
address.city	beijing		
address.province	beijing		
Key	Value	Description	

后端接收参数：

```
//POJO参数：请求参数与形参对象中的属性对应即可完成参数传递
@RequestMapping("/pojoParam")
@ResponseBody
public String pojoParam(User user){
    System.out.println("pojo参数传递 user ==> "+user);
    return "{\"module\":\"pojo param\"}";
}
```

4. 数组参数

请求参数名与形参对象属性名相同且请求参数为多个，定义数组类型即可接收参数。



The screenshot shows the Postman interface with a GET request to `http://localhost/arrayParam?likes=game&likes=music&likes=travel`. The 'Params' tab is selected, displaying the following table:

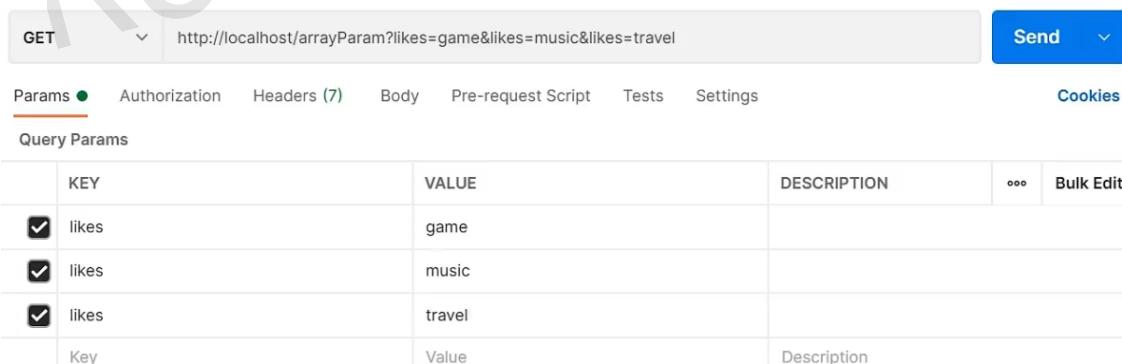
KEY	VALUE	DESCRIPTION	...	Bulk Edit
likes	game			
likes	music			
likes	travel			

后端接收参数：

```
//数组参数：同名请求参数可以直接映射到对应名称的形参数组对象中
@RequestMapping("/arrayParam")
@ResponseBody
public String arrayParam(String[] likes){
    System.out.println("数组参数传递 likes ==> "+
Arrays.toString(likes));
    return "{ 'module':'array param'}";
}
```

5. 集合类型参数

与数组类似，在url传参相同，但后台接收需要使用`@RequestParam`注解



The screenshot shows the Postman interface with a GET request to `http://localhost/arrayParam?likes=game&likes=music&likes=travel`. The 'Params' tab is selected, displaying the following table:

KEY	VALUE	DESCRIPTION	...	Bulk Edit
likes	game			
likes	music			
likes	travel			

后端接收参数：

```
//集合参数：同名请求参数可以使用@RequestParam注解映射到对应名称的集合对象
中作为数据
@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'}";
}
```

6. @RequestParam

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

JSON类型参数传递

1. 前置配置

SpringMVC默认使用的是jackson来处理json的转换，所以需要在pom.xml添加jackson依赖

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

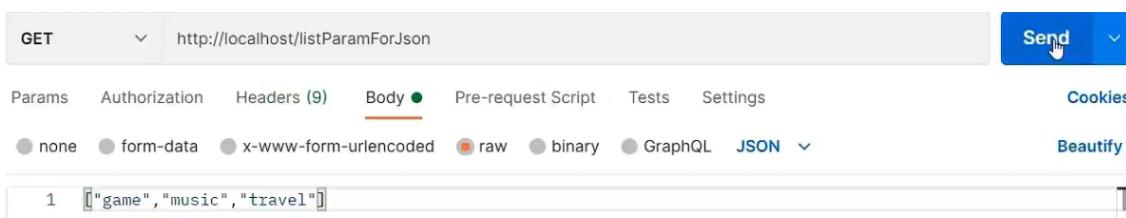
在SpringMVC的配置类中开启SpringMVC的注解支持，这里面就包含了将JSON转换成对象的功能。

```
@Configuration
@ComponentScan("com.itheima.controller")
//开启json数据类型自动转换
@EnableWebMvc
public class SpringMvcConfig {
```

2. JSON普通数组

形式：(["value1","value2","value3",...])

由于JSON数据写在请求体中，接收时要加上@RequestBody注解。



后端接收参数：

```
//使用@RequestBody注解将外部传递的json数组数据映射到形参的集合对象中作为  
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'}";  
}
```

3. JSON对象数据

形式： ({key1:value1,key2:value2,...})

The screenshot shows the Postman interface with a GET request to `http://localhost/pojoParamForJson`. The 'Body' tab is selected, showing a JSON object:

```
1 {  
2     "name": "itcast",  
3     "age": 15,  
4     "address": "",  
5     "province": "beijing",  
6     "city": "beijing"  
7 }  
8 }
```

后端接收数据：

```
@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'}";  
}
```

4. JSON对象数组

形式： ([{key1:value1,...},{key2:value2,...}])

The screenshot shows the Postman interface with a GET request to `http://localhost/listPojoParamForJson`. The 'Body' tab is selected, showing a JSON array of objects:

```
1 [  
2     {"name": "itcast", "age": 15},  
3     {"name": "itheima", "age": 12}  
4 ]  
5 ]
```

后端接收数据：

```

@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\"}";
}

```

5. 相关注解

@EnableWebMvc

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

@RequestBody

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

6. @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】

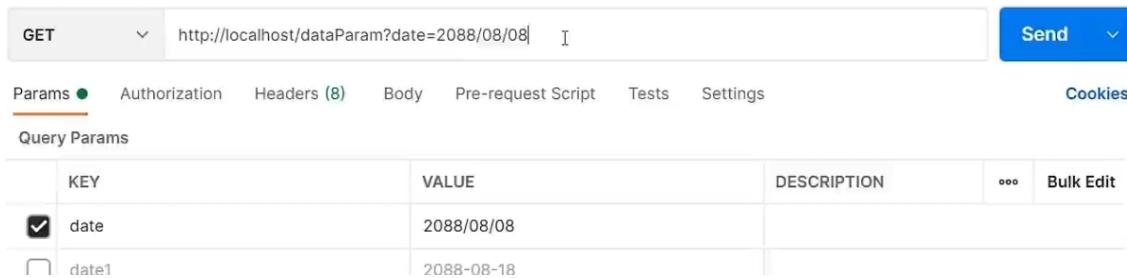
应用

- 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。

日期类型参数传递

1. yyyy/MM/dd格式

该格式默认可以传递，直接用日期类型的对象接收即可。



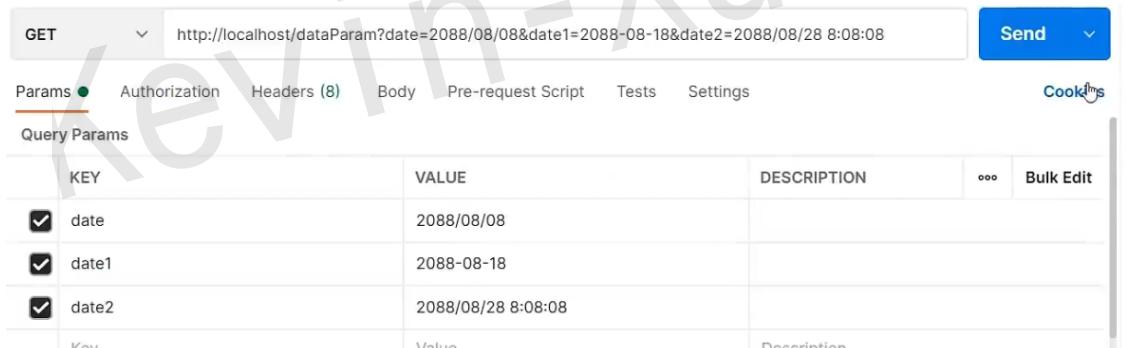
Postman screenshot showing a GET request to `http://localhost/dataParam?date=2088/08/08`. The 'Query Params' table contains two entries: 'date' with value '2088/08/08' and 'date1' with value '2088-08-18'. The 'Headers' tab shows 8 items.

后端接收数据：

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

2. 其他格式

需要用`@DateTimeFormat`注解说明从前端url传来的日期类型。



Postman screenshot showing a GET request to `http://localhost/dataParam?date=2088/08/08&date1=2088-08-18&date2=2088/08/28 8:08:08`. The 'Query Params' table shows three entries: 'date' (value '2088/08/08'), 'date1' (value '2088-08-18'), and 'date2' (value '2088/08/28 8:08:08'). The 'Headers' tab shows 8 items.

后端接收数据：

```

@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 date1(yyyy-MM-dd) ==> "+date1);
    System.out.println("参数传递 date2(yyyy/MM/dd HH:mm:ss) ==>
"+date2);
    return "{\"module\":\"data param\"}";
}

```

3. @DateTimeFormat

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
相关属性	pattern: 指定日期时间格式字符串

4. 内部实现原理

Converter接口

```

/**
 *   S: the source type
 *   T: the target type
 */
public interface Converter<S, T> {
    @Nullable
    //该方法就是将从页面上接收的数据(S)转换成我们想要的数据类型(T)返回
    T convert(S source);
}

```

有很多该接口的实现类负责各种类型转换。

响应

1. 响应页面 (了解)

```
@Controller
public class UserController {

    @RequestMapping("/toJumpPage")
    //注意
    //1. 此处不能添加@ResponseBody, 如果加了该注入, 会直接将page.jsp当字符串返回前端
    //2. 方法需要返回String
    public String toJumpPage(){
        System.out.println("跳转页面");
        return "page.jsp";
    }

}
```

2. 返回文本数据 (了解)

```
@Controller
public class UserController {

    @RequestMapping("/toText")
    //注意此处该注解就不能省略, 如果省略了, 会把response text当前页面名称去查找, 如果没有回报404错误
    @ResponseBody
    public String toText(){
        System.out.println("返回纯文本数据");
        return "response text";
    }

}
```

3. 响应JSON数据 (重点)

加上@ResponseBody注解, 并将返回类型设置为实体类。

```
@Controller
public class UserController {

    @RequestMapping("/toJsonPOJO")
    @ResponseBody
    public User toJsonPOJO(){
        System.out.println("返回json对象数据");
        User user = new User();
        user.setName("itcast");
        user.setAge(15);
        return user;
    }
}
```

```
}
```

```
}
```

4. 响应JSON集合数据（重点）

和响应JSON数据方法相同。

```
@Controller
public class UserController {

    @RequestMapping("/toJsonList")
    @ResponseBody
    public List<User> toJsonList(){
        System.out.println("返回json集合数据");
        User user1 = new User();
        user1.setName("传智播客");
        user1.setAge(15);

        User user2 = new User();
        user2.setName("黑马程序员");
        user2.setAge(12);

        List<User> userList = new ArrayList<User>();
        userList.add(user1);
        userList.add(user2);

        return userList;
    }

}
```

5. @ResponseBody

名称	@ResponseBody
类型	方法\类注解
位置	SpringMVC控制器方法定义上方和控制类上
作用	设置当前控制器返回值作为响应体， 写在类上，该类的所有方法都有该注解功能
相关属性	pattern: 指定日期时间格式字符串

当方法上有@ResponseBody注解后

- 方法的返回值为字符串，会将其作为文本内容直接响应给前端。
- 方法的返回值为对象，会将对象转换成JSON响应给前端。

REST风格

简介

1. REST简介

REST (Representational State Transfer) , 表现形式状态转换，它是一种软件架构风格。

当我们想表示一个网络资源的时候，可以使用两种方式：

传统风格资源描述形式

- `http://localhost/user/getById?id=1` 查询id为1的用户信息
- `http://localhost/user/saveUser` 保存用户信息

REST风格描述形式（对应上面两个url的功能）

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

因此REST风格的优点为：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；简化了书写。

2. REST行为动作

按照不同的请求方式代表不同的操作类型。

- 发送GET请求是用来做查询
- 发送POST请求是用来做新增
- 发送PUT请求是用来做修改
- 发送DELETE请求是用来做删除

3. RESTful：根据REST风格来访问资源。

RESTful开发

1. 开发方法

- (1) 对于不同的请求方式，用不同的注解来代表。
- (2) 各个方法的公共路径提取到类上，再将@ResponseBody注解和@Controller注解合并成一个注解@RestController。

```
@RestController // @Controller + ReponseBody  
@RequestMapping("/books")
```

```
public class BookController {  
  
    // @RequestMapping(method = RequestMethod.POST)  
    @PostMapping  
    public String save(@RequestBody Book book){  
        System.out.println("book save..." + book);  
        return "{ 'module': 'book save'}";  
    }  
  
    // @RequestMapping(value = "/{id}", method =  
    RequestMethod.DELETE)  
    @DeleteMapping("/{id}")  
    public String delete(@PathVariable Integer id){  
        System.out.println("book delete..." + id);  
        return "{ 'module': 'book delete'}";  
    }  
  
    // @RequestMapping(method = RequestMethod.PUT)  
    @PutMapping  
    public String update(@RequestBody Book book){  
        System.out.println("book update..." + book);  
        return "{ 'module': 'book update'}";  
    }  
  
    // @RequestMapping(value = "/{id}", method =  
    RequestMethod.GET)  
    @GetMapping("/{id}")  
    public String getById(@PathVariable Integer id){  
        System.out.println("book getById..." + id);  
        return "{ 'module': 'book getById'}";  
    }  
  
    // @RequestMapping(method = RequestMethod.GET)  
    @GetMapping  
    public String getAll(){  
        System.out.println("book getAll...");  
        return "{ 'module': 'book getAll'}";  
    }  
}
```

2. 相关注解

@PathVariable

名称	@PathVariable
类型	==形参注解==
位置	SpringMVC控制器方法形参定义前面
作用	绑定路径参数与处理器方法形参间的关系，要求路径参数名与形参名一一对应

@RestController

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

@GetMapping @PostMapping @PutMapping @DeleteMapping

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
相关属性	value (默认) : 请求访问路径

三个接收参数注解对比

@RequestBody、@RequestParam、@PathVariable三个注解之间的区别和应用分别是什么？

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。

SSM整合

整合配置

1. 流程分析

SpringConfig

- 标识该类为配置类 @Configuration
- 扫描Service所在的包 @ComponentScan
- 在Service层要管理事务 @EnableTransactionManagement
- 读取外部的properties配置文件 @PropertySource
- 整合Mybatis需要引入Mybatis相关配置类 @Import
 - 第三方数据源配置类 JdbcConfig
 - 构建DataSource数据源，DruidDataSource,需要注入数据库连接四要素，@Bean @Value
 - 构建平台事务管理器，DataSourceTransactionManager,@Bean
 - Mybatis配置类 MybatisConfig
 - 构建SqlSessionFactoryBean并设置别名扫描与数据源，@Bean
 - 构建MapperScannerConfigurer并设置DAO层的包扫描

SpringMvcConfig

- 标识该类为配置类 @Configuration
- 扫描Controller所在的包 @ComponentScan
- 开启SpringMVC注解支持 @EnableWebMvc

2. 具体流程

(1) 步骤1：创建web项目、添加依赖、创建项目结构。

(2) 步骤2：SpringConfig配置类

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

(3) 步骤3：创建JdbcConfig配置类

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

(4) 步骤4：创建MybatisConfig配置类

```

public class MyBatisConfig {
    @Bean
    public SqlSessionFactoryBean sqlSessionFactory(DataSource
datasource){
        SqlSessionFactoryBean factoryBean = new
SqlSessionFactoryBean();
        factoryBean.setDataSource(dataSource);

        factoryBean.setTypeAliasesPackage("com.itheima.domain");
        return factoryBean;
    }

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

```

(5) 步骤5：创建jdbc.properties

在resources下提供jdbc.properties,设置数据库连接四要素

```

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

```

(6) 步骤6：创建SpringMVC配置类

```

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

```

(7) 步骤7：创建Web项目入口配置类

```

public class ServletConfig extends
AbstractAnnotationConfigDispatcherServletInitializer {
    //加载Spring配置类
    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{SpringConfig.class};
    }
}

```

```

    }
    //加载SpringMVC配置类
    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{SpringMvcConfig.class};
    }
    //设置SpringMVC请求地址拦截规则
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }
    //设置post请求中文乱码过滤器
    @Override
    protected Filter[] getServletFilters() {
        CharacterEncodingFilter filter = new
CharacterEncodingFilter();
        filter.setEncoding("utf-8");
        return new Filter[]{filter};
    }
}

```

统一结果封装

1. 表现层与前端数据传输协议定义

为了将返回结果的数据进行统一，具体如何来做，大体的思路为：

- 为了封装返回的结果数据：创建结果模型类，封装数据到data属性中。
- 为了封装返回的数据是何种操作及是否操作成功：封装操作结果到code属性中。
- 操作失败后为了封装返回的错误信息：封装特殊消息到message(msg)属性中。

根据分析，设置统一数据返回结果类

```

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

```

2. 表现层与前端数据传输协议实现

(1) 步骤1：创建Result类

```

public class Result {
    //描述统一格式中的数据
}

```

```

private Object data;
//描述统一格式中的编码，用于区分操作，可以简化配置0或1表示成功失败
private Integer code;
//描述统一格式中的消息，可选属性
private String msg;

public Result() {
}

//构造方法是方便对象的创建
public Result(Integer code, Object data) {
    this.data = data;
    this.code = code;
}

//构造方法是方便对象的创建
public Result(Integer code, Object data, String msg) {
    this.data = data;
    this.code = code;
    this.msg = msg;
}

//setter...getter...省略
}

```

(2) 步骤2：定义返回码Code类

```

//状态码
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;
}

```

code类中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等。

(3) 步骤3：修改Controller类的返回值

```

//统一每一个控制器方法返回值
@RestController

```

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

    @Autowired
    private BookService bookService;

    @PostMapping
    public Result save(@RequestBody Book book) {
        boolean flag = bookService.save(book);
        return new Result(flag ?
Code.SAVE_OK:Code.SAVE_ERR,flag);
    }

    @PutMapping
    public Result update(@RequestBody Book book) {
        boolean flag = bookService.update(book);
        return new Result(flag ?
Code.UPDATE_OK:Code.UPDATE_ERR,flag);
    }

    @DeleteMapping("/{id}")
    public Result delete(@PathVariable Integer id) {
        boolean flag = bookService.delete(id);
        return new Result(flag ?
Code.DELETE_OK:Code.DELETE_ERR,flag);
    }

    @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);
    }

    @GetMapping
    public Result getAll() {
        List<Book> bookList = bookService.getAll();
        Integer code = bookList != null ? Code.GET_OK :
Code.GET_ERR;
        String msg = bookList != null ? "" : "数据查询失败, 请重试!";
        return new Result(code,bookList,msg);
    }
}
```

```
}
```

统一异常处理

1. 问题描述

当后端程序出现异常，需要先将异常处理完再交给前端。

为了处理异常，思路为：

- 各个层级都会出现异常，统一将所有异常抛出到表现层进行处理。
- 异常种类很多，表现层需要对异常进行分类。
- 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，使用AOP思想解决。

2. 异常处理器

SpringMVC提供了异常处理器，可以集中的、统一的处理项目中出现的异常。

测试使用步骤：

(1) 步骤1：创建异常处理器类

```
//@RestControllerAdvice用于标识当前类为REST风格对应的异常处理器
@RestControllerAdvice
public class ProjectExceptionAdvice {
    //除了自定义的异常处理器，保留对Exception类型的异常处理，用于处理非预期的异常
    @ExceptionHandler(Exception.class) //异常处理标志，告知方法处理何种异常
    public void doException(Exception ex){ //处理异常的方法，拦截到的异常会作为形参ex传入
        System.out.println("嘿嘿，异常你哪里跑！")
    }
}
```

PS：要确保SpringMvcConfig能够扫描到异常处理器类。

(2) 步骤2：让程序抛出异常

修改BookController的getById方法，添加int i = 1/0.

```

@GetMapping("/{id}")
public Result getById(@PathVariable Integer id) {
    int i = 1/0;
    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);
}

```

(3) 步骤3：运行程序，测试

控制台成功打印，说明已经拦截到了程序中的异常。

(4) 步骤4：修改异常处理器方法

由于此时信息只在控制台打印，前端未收到任何关于异常的信息，所以需要异常处理器类返回结果给前端。

```

//@RestControllerAdvice用于标识当前类为REST风格对应的异常处理器
@RestControllerAdvice
public class ProjectExceptionAdvice {
    //除了自定义的异常处理器，保留对Exception类型的异常处理，用于处理非预期的异常
    @ExceptionHandler(Exception.class)
    public Result doException(Exception ex){
        System.out.println("嘿嘿，异常你哪里跑！");
        return new Result(666,null,"嘿嘿，异常你哪里跑！");
    }
}

```

3. 相关注解

@RestControllerAdvice

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

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

@ExceptionHandler

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	设置处理哪类异常，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

项目异常处理

1. 异常分类

由于异常很多，不可能为每个异常都写一个控制器方法。因此需要为异常分类，这样处理的仅为分完类的异常。

业务异常 (BusinessException)

- 规范的用户行为产生的异常
 - 如用户在页面输入内容的时候未按照指定格式进行数据填写，如在年龄框输入的是字符串
- 不规范的用户行为操作产生的异常
 - 如用户故意传递错误数据
- 系统异常 (SystemException)
 - 项目运行过程中可预计但无法避免的异常
 - 比如数据库或服务器宕机
- 其他异常 (Exception)
 - 编程人员未预期到的异常，如：用到的文件不存在

将异常分类以后，针对不同类型的异常，要提供具体的解决方案。

2. 异常解决方案

业务异常 (BusinessException)

- 发送对应消息传递给用户，提醒规范操作
 - 大家常见的就是提示用户名已存在或密码格式不正确等

系统异常 (SystemException)

- 发送固定消息传递给用户，安抚用户
 - 系统繁忙，请稍后再试
 - 系统正在维护升级，请稍后再试

- 系统出问题，请联系系统管理员等
- 发送特定消息给运维人员，提醒维护
 - 可以发送短信、邮箱或者是公司内部通信软件
- 记录日志
 - 发消息和记录日志对用户来说是不可见的，属于后台程序

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）
 - 一般是程序没有考虑全，比如未做非空校验等
- 记录日志

3. 具体实现

(1) 步骤1：自定义异常，主要是为异常提供不同的构造器

```
//自定义异常处理器，用于封装异常信息，对异常进行分类
public class SystemException extends RuntimeException{
    private Integer code;

    public Integer getCode() {
        return code;
    }

    public void setCode(Integer code) {
        this.code = 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;
    }

}

//自定义异常处理器，用于封装异常信息，对异常进行分类
public class BusinessException extends RuntimeException{
    private Integer code;
```

```
public Integer getCode() {
    return code;
}

public void setCode(Integer code) {
    this.code = 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;
}

}
```

(2) 步骤2：将其他异常包装成自定义异常

具体的包装方式有：

- 方式一：try{}catch(){ } 在catch中重新throw我们自定义异常即可。
- 方式二：直接throw自定义异常即可。

```

public Book getById(Integer id) {
    //模拟业务异常，包装成自定义异常
    if(id == 1){
        throw new BusinessException(Code.BUSINESS_ERR,"请不要使用你的技术挑战我的耐性!");
    }
    //模拟系统异常，将可能出现的异常进行包装，转换成自定义异常
    try{
        int i = 1/0;
    }catch (Exception e){
        throw new SystemException(Code.SYSTEM_TIMEOUT_ERR,"服务器访问超时，请重试!",e);
    }
    return bookDao.getById(id);
}

```

(3) 步骤3：在异常处理器里处理自定义异常

```

//@RestControllerAdvice用于标识当前类为REST风格对应的异常处理器
@RestControllerAdvice
public class ProjectExceptionAdvice {
    //ExceptionHandler用于设置当前处理器类对应的异常类型
    @ExceptionHandler(SystemException.class)
    public Result doSystemException(SystemException ex){
        //记录日志
        //发送消息给运维
        //发送邮件给开发人员，ex对象发送给开发人员
        return new Result(ex.getCode(),null,ex.getMessage());
    }

    @ExceptionHandler(BusinessException.class)
    public Result doBusinessException(BusinessException ex){
        return new Result(ex.getCode(),null,ex.getMessage());
    }

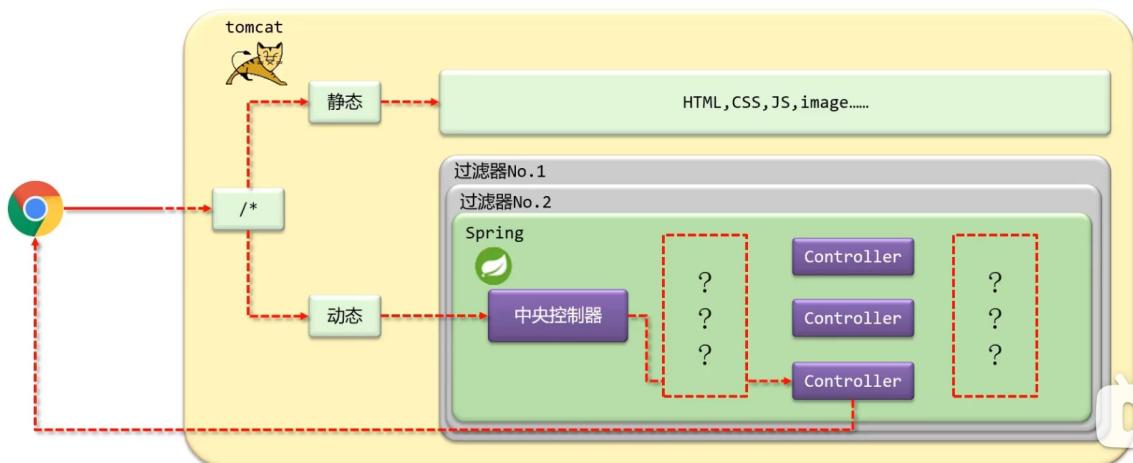
    //除了自定义的异常处理器，保留对Exception类型的异常处理，用于处理非预期的异常
    @ExceptionHandler(Exception.class)
    public Result doOtherException(Exception ex){
        //记录日志
        //发送消息给运维
        //发送邮件给开发人员，ex对象发送给开发人员
        return new Result(Code.SYSTEM_UNKNOW_ERR,null,"系统繁忙，请稍后再试!");
    }
}

```

拦截器

概述与拦截器使用

1. 拦截器概念



浏览器发起请求后，Tomcat会拦截所有的请求。对于静态资源，Tomcat即可在目录下直接访问，进行处理。对于动态资源，要交给后端代码处理。动态资源的请求先经过过滤器，再进入到SpringMVC的中央控制器，再经过拦截器到达Controller。

因此，拦截器（Interceptor）是一种动态拦截方法调用的机制，在SpringMVC中动态拦截控制器方法的执行。

拦截器作用：

- (1) 在指定的方法调用前后执行预先设定的代码。
- (2) 阻止原始方法的执行。

拦截器和过滤器的区别：

- (1) 归属不同：Filter属于Servlet技术，Interceptor属于SpringMVC技术。
- (2) 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。SpringMVC的访问可以在配置类中设定，其范围可以小于所有访问。

```
c ServletContainersInitConfig.java x
1 package com.itheima.config;
2
3 import ...
4
5 public class ServletContainersInitConfig extends AbstractAnnotationConfigDispatcherServletInitializer {
6     protected Class<?>[] getRootConfigClasses() { return new Class[0]; }
7
8     protected Class<?>[] getServletConfigClasses() { return new Class[]{SpringMvcConfig.class}; }
9
10    protected String[] getServletMappings() { return new String[]{"/"}; }
11
12    //乱码处理
13    @Override
14    protected Filter[] getServletFilters() {
15        CharacterEncodingFilter filter = new CharacterEncodingFilter();
16        filter.setEncoding("UTF-8");
17        return new Filter[]{filter};
18    }
19
20 }
21
22 }
```

2. 拦截器使用

(1) 步骤1：创建拦截器类

让类实现HandlerInterceptor接口，重写接口中的三个方法。

```
@Component
//定义拦截器类，实现HandlerInterceptor接口
//注意当前类必须受Spring容器控制
public class ProjectInterceptor implements HandlerInterceptor {
    @Override
    //原始方法调用前执行的内容
    public boolean preHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler) throws Exception {
        System.out.println("preHandle...");
        return true; //设置为true才会放行
    }

    @Override
    //原始方法调用后执行的内容
    public void postHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("postHandle...");
    }

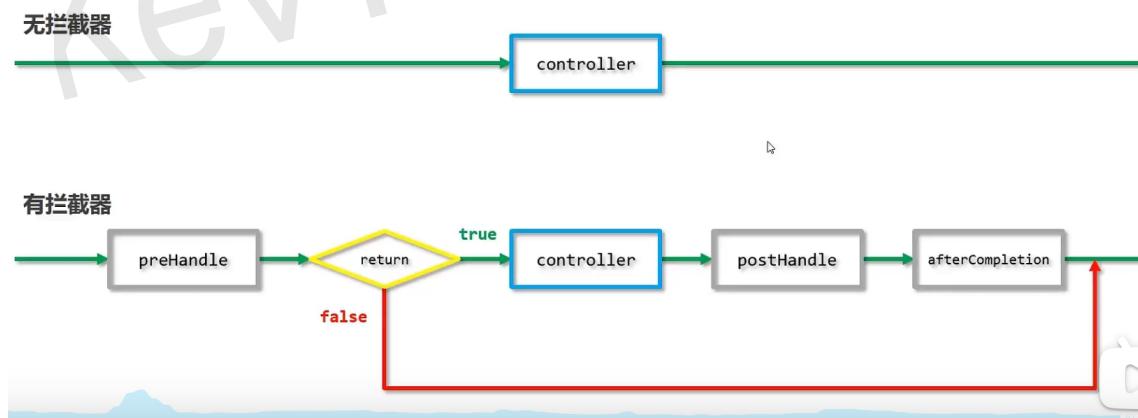
    @Override
    //原始方法调用完成后执行的内容
    public void afterCompletion(HttpServletRequest request,
    HttpServletResponse response, Object handler, Exception ex)
    throws Exception {
        System.out.println("afterCompletion...");
    }
}
```

```
    }  
}
```

(2) 步骤2：配置拦截器

```
@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; //注入创建的拦截器  
  
    @Override  
    public void addInterceptors(InterceptorRegistry registry)  
    {  
        //配置多拦截器  
  
        registry.addInterceptor(projectInterceptor).addPathPatterns(""  
        "/books","/books/*");  
    }  
}
```

3. 拦截器执行流程



拦截器参数

1. 前置处理方法

原始方法之前运行preHandle

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

- 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，对反射中的Method对象进行了再包装

使用request对象可以获取请求数据中的内容，如获取请求头的Content-Type

```
public boolean preHandle(HttpServletRequest request,
                         HttpServletResponse response, Object handler) throws Exception {
    String contentType = request.getHeader("Content-Type");
    System.out.println("preHandle..."+contentType);
    return true;
}
```

使用handler参数，可以获取方法的相关信息

```
public boolean preHandle(HttpServletRequest request,
                         HttpServletResponse response, Object handler) throws Exception {
    HandlerMethod hm = (HandlerMethod)handler;
    String methodName = hm.getMethod().getName(); //可以获取方法的名称
    System.out.println("preHandle..."+methodName);
    return true;
}
```

2. 后置处理方法

原始方法运行后运行，如果原始方法被拦截，则不执行

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

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。

3. 完成处理方法

拦截器最后执行的方法，无论原始方法是否执行

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

ex: 如果处理器执行过程中出现异常对象，可以在原始方法执行之后拿到这个异常对象。

拦截器链配置

1. 拦截器链

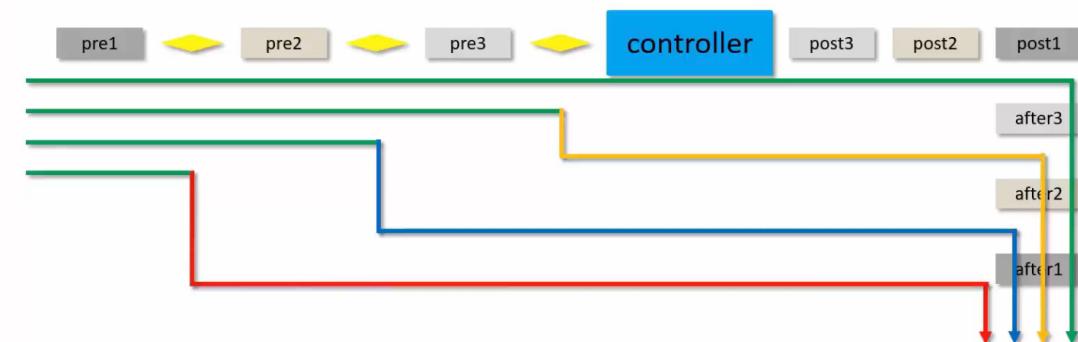
配置了多个拦截器，即形成了拦截器链。

拦截器链执行顺序：

- 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（即某个拦截器设置prehandle返回false），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。
如下图，若pre3设置为false，运行中断，会直接执行前两个拦截器的afterCompletion操作。

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

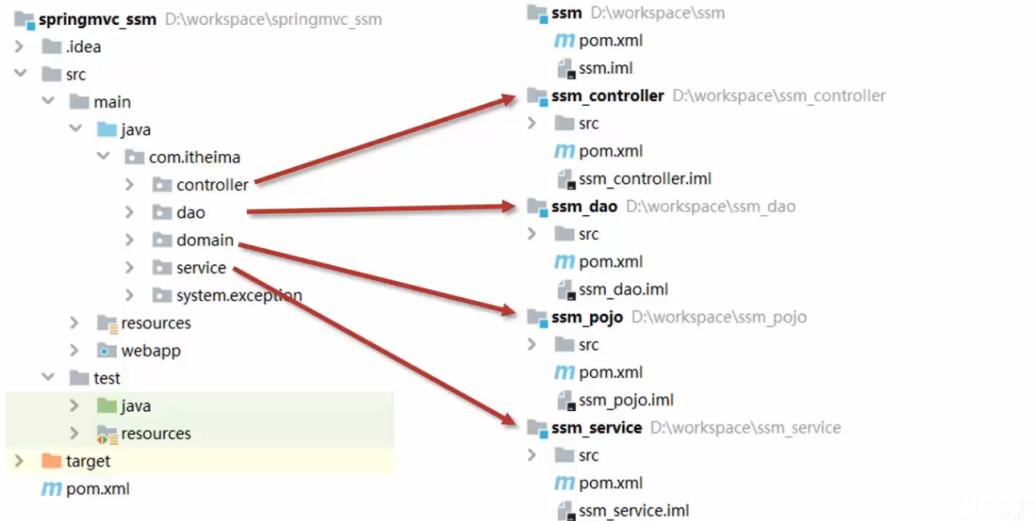


Maven高级

分模块开发

1. 分模块开发的意义

将原始模块按照功能拆分成若干个子模块，方便模块间的相互调用，接口共享



如上图，当进行分模块开发，项目中的每一层都可以单独维护。且由于抽取成一个独立的模块，其他项目中的模块要想使用时可以像添加第三方jar包依赖一样来使用我们自己抽取的模块。

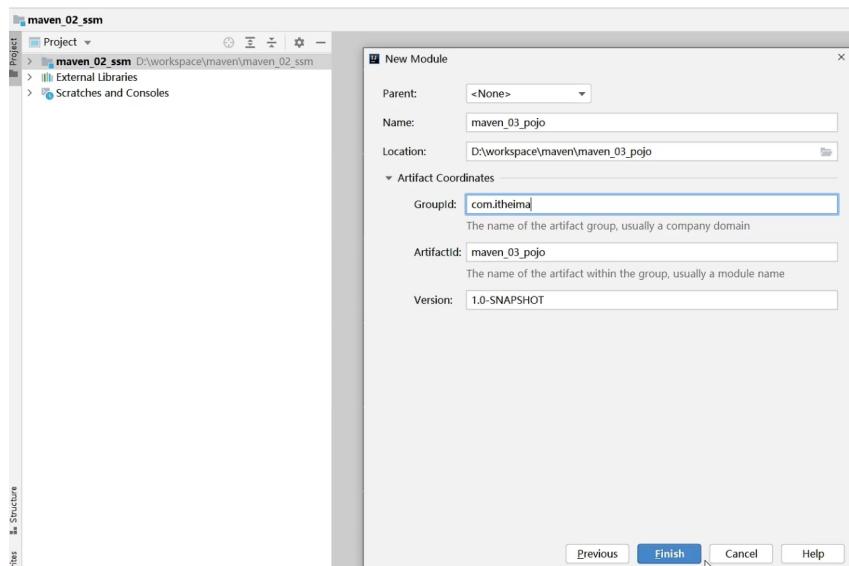
比如电商项目中有订单和商品两个模块，订单中需要商品的模型类，商品中也需要商品的模型类。将商品模块进行分模块开发，抽取出的domain层就可以引到订单模块中，避免重复开发。

2. 分模块开发的实现

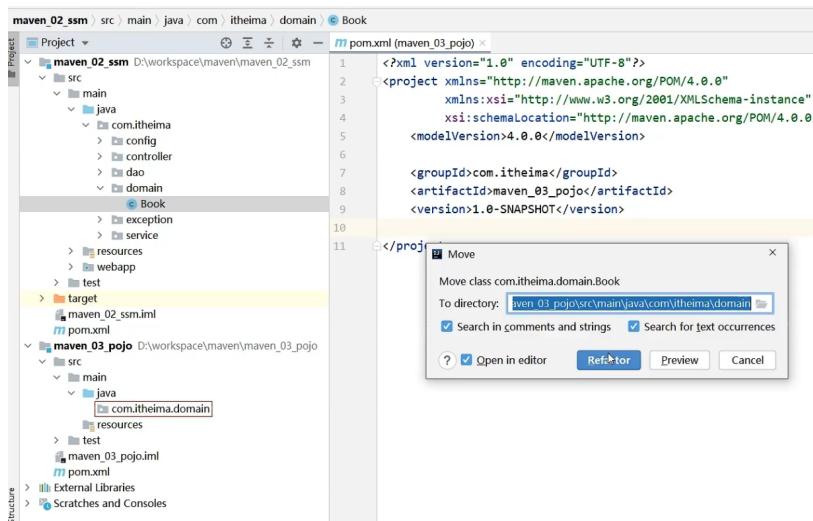
以抽取domain和dao为例。

抽取domain：

(1) 步骤1：创建新模块



(2) 步骤2：项目中创建domain包，并将原模块中domain包下的程序复制到新模块中



(3) 步骤3：删除原项目中的domain包

(4) 步骤4：建立依赖关系

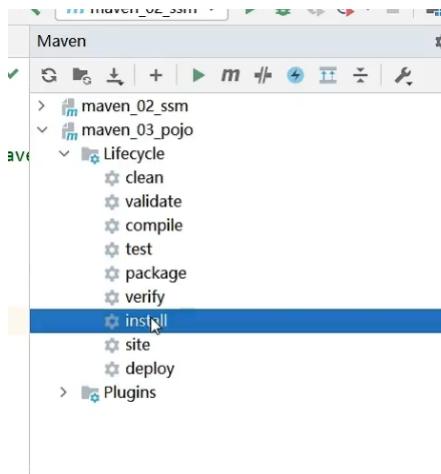
经历步骤3后，模块maven_02_ssm会报错，因为里面用到了Book对象，而Book对象此时在maven_03.pojo模块中。

在maven_02_ssm项目的pom.xml添加maven_03.pojo的依赖（从maven_03.pojo的pom.xml复制）。

```
<dependency>
    <groupId>com.itheima</groupId>
    <artifactId>maven_03.pojo</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

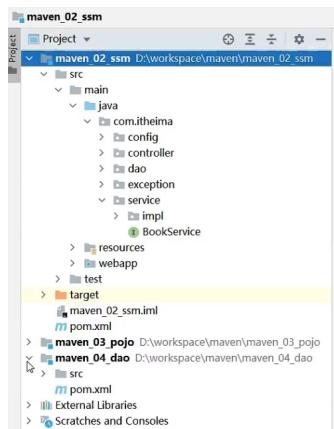
(5) 步骤5：将项目安装到本地仓库

经历步骤4后，虽然在IDEA中不报错，但项目无法编译运行，因为maven编译会从本地找jar包，而此时jar包只是在IDEA成功导入，本地仓库并不存在该jar包。因此需要将新项目安装到本地仓库。

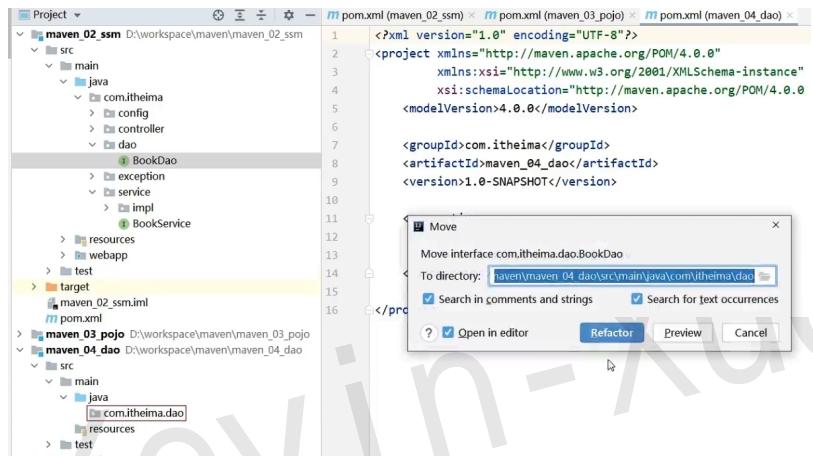


抽取dao：

(1) 步骤1：创建新模块



(2) 步骤2：项目中创建dao包，并将原模块中dao包下的程序复制到新模块中



(3) 步骤3：删除原项目中的dao包

(4) 步骤4：建立依赖关系

经历步骤3后，模块maven_04_ssm会报错，因为里面用到了Book对象，而Book对象此时在maven_03_pojo模块中。

在maven_04_ssm项目的pom.xml添加maven_03_pojo的依赖（从maven_03_pojo的pom.xml复制）。

同时由于dao层用了mybatis，还需要引入mybatis的依赖。

(5) 步骤5：将项目安装到本地仓库

依赖管理

依赖传递与依赖冲突

1. 依赖

依赖指当前项目运行所需的jar，一个项目可以设置多个依赖。

格式为：

```

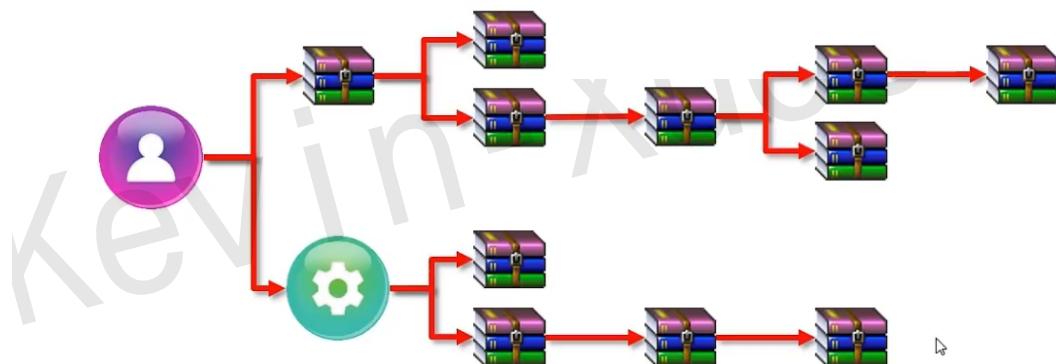
<!--设置当前项目所依赖的所有jar-->
<dependencies>
    <!--设置具体的依赖-->
    <dependency>
        <!--依赖所属群组id-->
        <groupId>org.springframework</groupId>
        <!--依赖所属项目id-->
        <artifactId>spring-webmvc</artifactId>
        <!--依赖版本号-->
        <version>5.2.10.RELEASE</version>
    </dependency>
</dependencies>

```

2. 依赖传递

依赖具有传递性，分为直接依赖和间接依赖。项目运行可以使用直接依赖和间接依赖。

- 直接依赖：在当前项目中通过依赖配置建立的依赖关系
- 间接依赖：被资源的资源如果依赖其他资源，当前项目间接依赖其他资源



3. 依赖冲突

依赖传递的存在，会导致jar包在依赖的过程中出现冲突问题，即项目依赖的某一个jar包，有多个不同的版本，因而造成类包版本冲突。

依赖冲突时应遵循的原则：

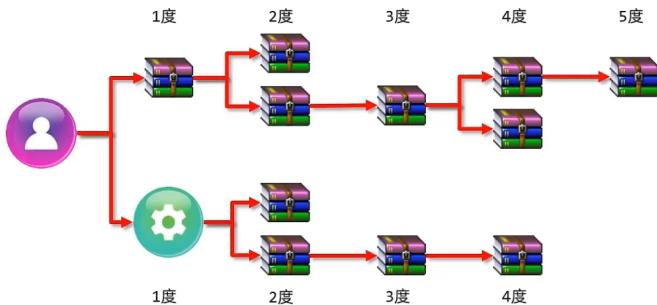
- (1) 特殊优先：当同级配置了相同资源的不同版本，后配置的覆盖先配置的。
如下图，同时配置Junit两个版本，使用的是后配置的版本。

```

4   xsi:schemaLocation="http://maven.apache.org/POX/4.0.0 http://maven.apache.org/POX/4.0.0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6
7   <groupId>com.itheima</groupId>
8   <artifactId>maven_02_ssm</artifactId>
9   <version>1.0-SNAPSHOT</version>
10  <packaging>war</packaging>
11
12  <dependencies>
13
14    <dependency>
15      <groupId>junit</groupId>
16      <artifactId>junit</artifactId>
17      <version>4.12</version>
18      <scope>test</scope>
19    </dependency>
20
21    <dependency>
22      <groupId>junit</groupId>
23      <artifactId>junit</artifactId>
24      <version>4.11</version>
25      <scope>test</scope>
26    </dependency>

```

(2) 路径优先：当依赖中出现相同的资源时，层级越深，优先级越低，层级越浅，优先级越高。



(3) 声明优先：当资源在相同层级被依赖时，配置顺序靠前的覆盖配置顺序靠后的。

假设A通过B间接依赖到D1，A通过C间接依赖到D2，D1和D2都是两度，这个时候就不能按照层级来选择，需要按照声明来，谁先声明用谁，也就是说B在C之前声明，这个时候使用的是D1，反之则为D2。

依赖关系查看：Maven的 Dependencies 面板中可以查看依赖关系。

可选依赖与排除依赖

1. 可选依赖

可选依赖是隐藏当前工程所依赖的资源，隐藏后对应资源将不具有依赖传递。

在依赖中添加optional标签，设置为true即表示可选依赖，此时其他模块就算导入该jar包也无法使用该资源。

```

<dependency>
  <groupId>com.itheima</groupId>
  <artifactId>maven_03_pojo</artifactId>
  <version>1.0-SNAPSHOT</version>
  <!-- 可选依赖是隐藏当前工程所依赖的资源，隐藏后对应资源将不具有依赖传递-->
  <optional>true</optional>
</dependency>

```

此时maven_02_ssm项目将无法使用maven_03_pojo的资源。

2. 排除依赖

排除依赖指主动断开依赖的资源，被排除的资源无需指定版本。

通俗的说，可选依赖表示我不想被别人用，排除依赖指的是我不想用别人的。

```
<dependency>
    <groupId>com.itheima</groupId>
    <artifactId>maven_04_dao</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!--排除依赖是隐藏当前资源对应的依赖关系-->
    <exclusions>
        <exclusion>
            <groupId>com.itheima</groupId>
            <artifactId>maven_03_pojo</artifactId>
        </exclusion>
        <exclusion>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

聚合和继承

聚合

1. 聚合概述

当进行分模块开发后会出现一个问题，如下图，当ssm_pojo修改后，由于其他模块依赖ssm_pojo，需要重新构建其他所有模块。因此，想要有一个用于管理的工程，同时管理这四个工程，当发生了更新的操作，同时对这四个工程进行构建。



聚合：将多个模块组织成一个整体，同时进行项目构建的过程称为聚合。

聚合工程：通常是一个不具有业务功能的“空”工程（有且仅有一个pom文件）。

作用：使用聚合工程可以将多个工程编组，通过对聚合工程进行构建，实现对所包含的模块进行同步构建，解决批量模块同步构建的问题。

2. 聚合实现步骤

- (1) 步骤1：创建一个空的maven项目
- (2) 步骤2：pom.xml中将聚合工程的打包方式改为pom

```
<?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>

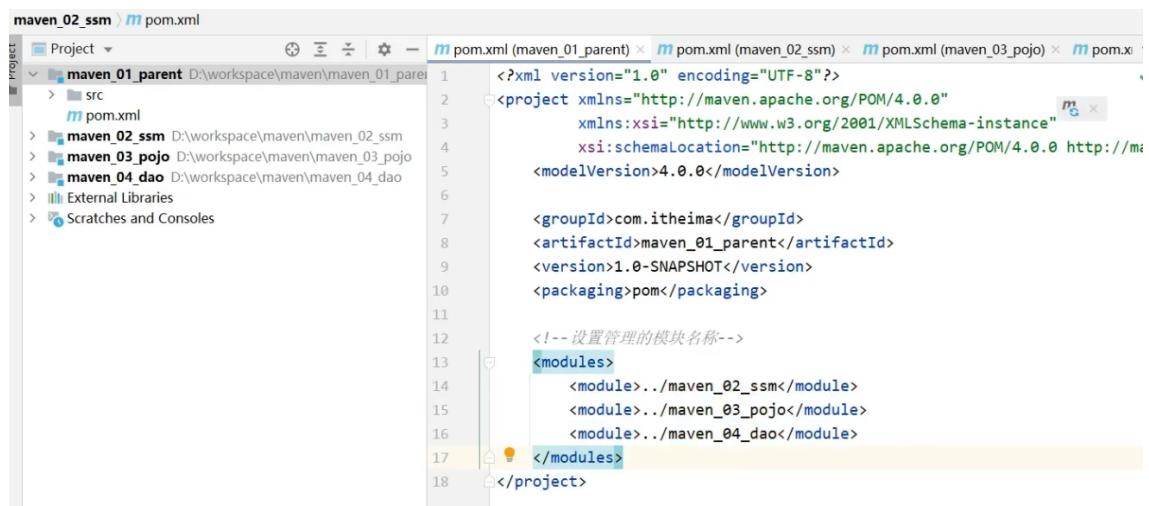
    <groupId>com.itheima</groupId>
    <artifactId>maven_01_parent</artifactId>
    <version>1.0-RELEASE</version>
    <packaging>pom</packaging> <!-- 打包方式-->

</project>
```

项目的打包方式，接触到的有三种，分别是

- jar：默认情况，说明该项目为java项目
- war：说明该项目为web项目
- pom：说明该项目为聚合或继承项目

- (3) 步骤3：pom.xml中设置管理的模块名称



- (4) 步骤4：统一管理项目

只要构建聚合工程，就会同时构建其管理的模块。构建的顺序取决于各模块的依赖关系，比如A依赖B，那么一定会先构建B。

继承

1. 继承概述

继承：描述的是两个工程间的关系，与java中的继承相似，子工程可以继承父工程中的配置信息，常见于依赖关系的继承。

作用：简化配置，减少版本冲突。

2. 继承实现

在子项目的pom文件中设置其父工程即可。

```
<!--配置当前工程继承自parent工程-->
<parent>
    <groupId>com.itheima</groupId>
    <artifactId>maven_01_parent</artifactId>
    <version>1.0-RELEASE</version>
    <!--设置父项目pom.xml位置路径-->
    <relativePath>../maven_01_parent/pom.xml</relativePath>
</parent>
```

3. 优化子项目依赖版本问题

当把所有jar包都引到父工程，会导致很多子工程继承自己不需要的jar包，此时需要进行优化。

优化步骤：

(1) 步骤1：在父工程的pom.xml定义依赖管理

代表并不强制要求子工程继承的依赖，而只是为子工程提供了可选择的依赖资源。这些依赖不会被自动继承，需要子工程重写才能继承。

```
<!--定义依赖管理-->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

(2) 步骤2：在子工程中配置使用父工程中可选依赖的坐标

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <scope>test</scope>
</dependency>
```

子工程中就不需要添加版本了，这样做的好处就是当父工程 dependencyManagement 标签中的版本发生变化后，子项目中的依赖版本也会跟着发生变化。

4. 聚合和继承的区别

两种之间的作用：

- 聚合用于快速构建项目，对项目进行管理
- 继承用于快速配置和管理子项目中所使用jar包的版本

聚合和继承的相同点：

- 聚合与继承的 pom.xml 文件打包方式均为 pom，可以将两种关系制作到同一个 pom 文件中
- 聚合与继承均属于设计型模块，并无实际的模块内容

聚合和继承的不同点：

- 聚合是在当前模块中配置关系，聚合可以感知到参与聚合的模块有哪些
- 继承是在子模块中配置关系，父模块无法感知哪些子模块继承了自己

属性

1. 属性配置与使用

属性：可以理解为 pom.xml 中的变量。声明一个变量，在其他地方使用该变量，当变量的值发生变化后，所有使用变量的地方，就会跟着修改。使用属性可以对 pom.xml 中的信息做集中管理。

属性的使用：

- (1) 步骤1：父工程中定义属性

使用标签，属性名随意声明。

```
<properties>
    <spring.version>5.2.10.RELEASE</spring.version>
    <junit.version>4.12</junit.version>
    <mybatis-spring.version>1.3.0</mybatis-spring.version>
</properties>
```

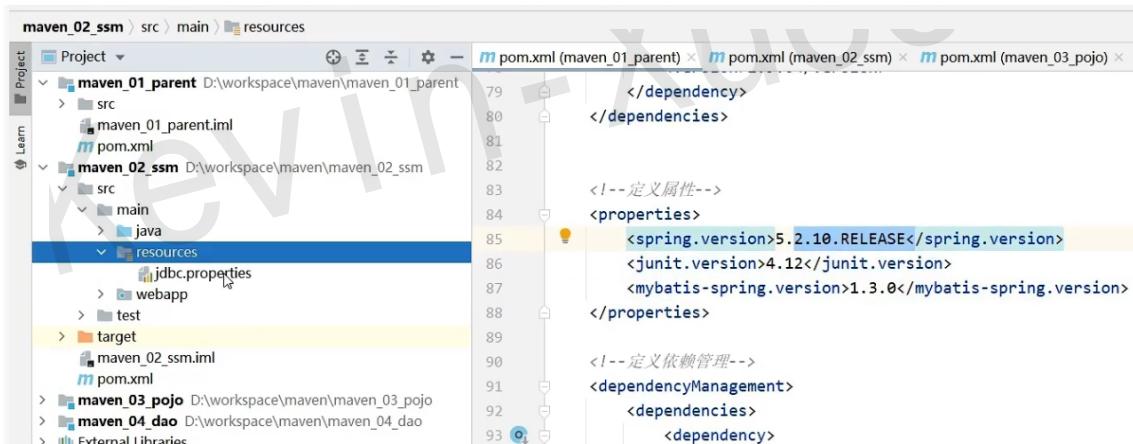
- (2) 步骤2：修改依赖的 version

在需要使用该属性的地方引用属性即可。

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${spring.version}</version>
</dependency>
```

2. 配置文件加载属性

除了pom.xml中的信息，maven还是管理其他配置文件中的信息，如下图，可以在maven中设置jdbc.properties文件的属性。



使用步骤：

(1) 步骤1：父工程定义属性

```
<properties>
    <jdbc.url>jdbc:mysql://127.1.1.1:3306/ssm_db</jdbc.url>
</properties>
```

(2) 步骤2：jdbc.properties文件中引用属性

在jdbc.properties，将jdbc.url的值直接获取Maven配置的属性

```
jdbc.driver=com.mysql.jdbc.Driver  
jdbc.url=${jdbc.url}  
jdbc.username=root  
jdbc.password=root
```

(3) 步骤3：设置maven过滤文件范围

```
<build>  
  <resources>  
    <!--  
      ${project.basedir}: 当前项目所在目录，子项目继承了父项目，  
      相当于所有的子项目都添加了资源目录的过滤  
    -->  
    <resource>  
  
      <directory>${project.basedir}/src/main/resources</directory>  
      <filtering>true</filtering>  
    </resource>  
  </resources>  
</build>
```

(4) 步骤4：配置maven打war包时，忽略web.xml检查（在子工程的pom文件中设置）

```
<build>  
  <plugins>  
    <plugin>  
      <groupId>org.apache.maven.plugins</groupId>  
      <artifactId>maven-war-plugin</artifactId>  
      <version>3.2.3</version>  
      <configuration>  
  
        <failOnMissingWebXml>false</failOnMissingWebXml>  
      </configuration>  
    </plugin>  
  </plugins>  
</build>
```

多环境配置与应用

多环境开发

1. 多环境开发

由于本地、生产、测试等环境使用的配置可能不同，需要在pom.xml文件中对多个环境作不同的配置。通俗的说，就是把一系列属性编个组，设成一个名称，根据要启动的环境选择名称。

使用步骤：

(1) 步骤1：父工程配置多个环境，并指定默认激活环境

```
<profiles>
    <!--开发环境-->
    <profile>
        <id>env_dep</id>
        <properties>

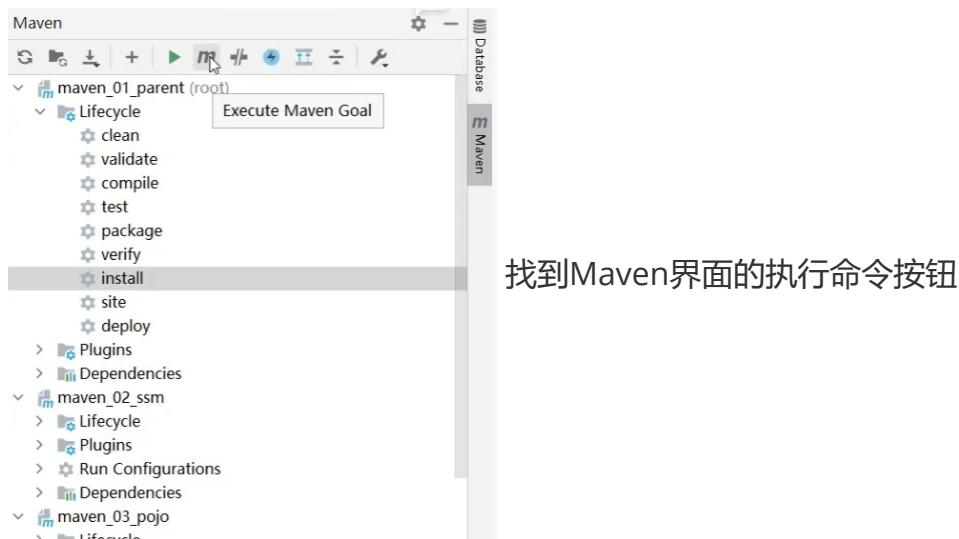
            <jdbc.url>jdbc:mysql://127.1.1.1:3306/ssm_db</jdbc.url>
            </properties>
            <!--设定是否为默认启动环境-->
            <activation>
                <activeByDefault>true</activeByDefault>
            </activation>
        </profile>
        <!--生产环境-->
        <profile>
            <id>env_pro</id>
            <properties>

                <jdbc.url>jdbc:mysql://127.2.2.2:3306/ssm_db</jdbc.url>
                </properties>
            </profile>
            <!--测试环境-->
            <profile>
                <id>env_test</id>
                <properties>

                    <jdbc.url>jdbc:mysql://127.3.3.3:3306/ssm_db</jdbc.url>
                    </properties>
                </profile>
            </profiles>
```

(2) 步骤2：用命令进行构建，选择使用的环境

用这个步骤可以避免每次使用不同环境时，需要去pom中修改默认启动环境。



执行命令实现构建

```
mvn install -P 环境id(步骤1设定)
```

跳过测试

1. 背景

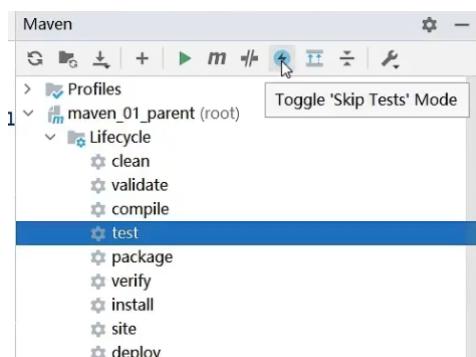
执行install指令的时候，Maven都会按照顺序从上往下依次执行，默认都会执行test。

但是遇到功能更新中且没有开发完毕、想要快速打包等情况，我们往往想跳过测试，减少耗费的时间。

2. 跳过测试的方法

(1) IDEA工具实现

这种方式最简单，但是有点“暴力”，会把所有的测试都跳过，如果我们想更精细的控制哪些跳过哪些不跳过，就需要使用配置插件的方式。



(2) 配置插件实现跳过测试

在父工程中的pom.xml中添加测试插件配置。

```
<build>
  <plugins>
```

```

<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.12.4</version>
    <configuration>
        <skipTests>false</skipTests>
        <!--排除掉不参与测试的内容-->
        <excludes>
            <exclude>**/BookServiceTest.java</exclude>
        </excludes>
    </configuration>
</plugin>
</plugins>
</build>

```

skipTests：如果为true，则跳过所有测试，如果为false，则不跳过测试。

excludes：哪些测试类不参与测试，即排除，针对skipTests为false来设置的。

includes：哪些测试类要参与测试，即包含，针对skipTests为true来设置的。

(3) 命令行跳过测试

```

mvn install -D skipTests
install可以是其他指令，如package、compile

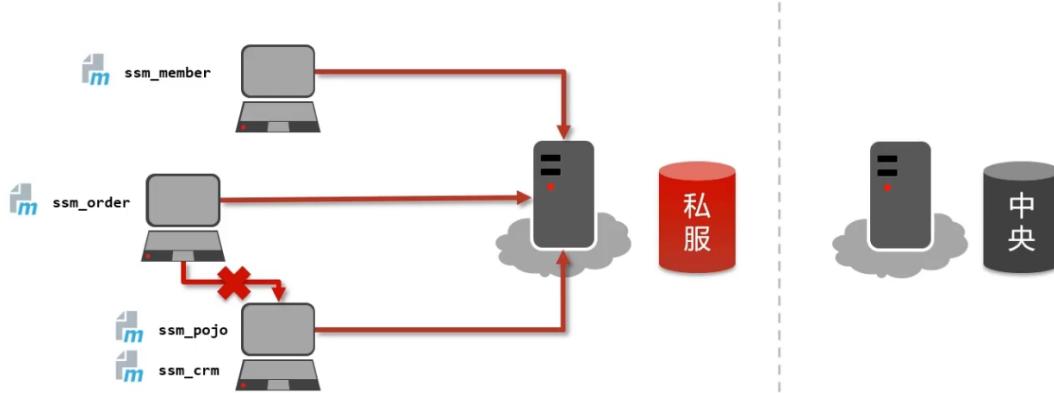
```

私服

1. 私服简介

私服是一台独立的服务器，用于解决团队内部的资源共享与资源同步问题。

如下图，在此前的介绍中，能让ssm_pojo调用ssm_crm，但是无法让ssm_order调用ssm_crm。通过搭建私服，将所有资源上传到私服，能实现小范围里的资源共享。



2. 私服仓库分类

宿主仓库hosted

- 保存无法从中央仓库获取的资源
 - 自主研发
 - 第三方非开源项目,比如Oracle,因为是付费产品,所以中央仓库没有代理仓库proxy
- 代理远程仓库,通过nexus访问其他公共仓库,例如中央仓库仓库组group
 - 将若干个仓库组成一个群组,简化配置
 - 仓库组不能保存资源,属于设计型仓库

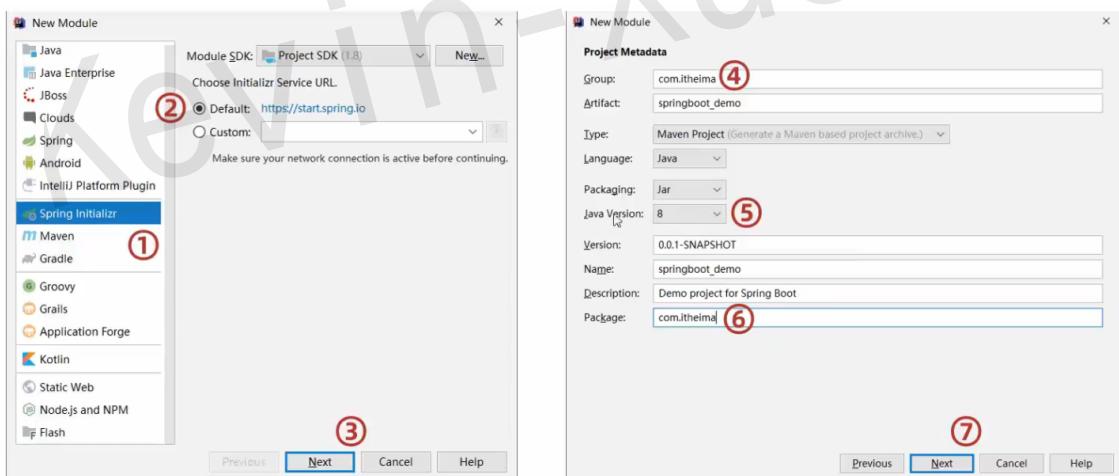
SpringBoot

SpringBoot简介

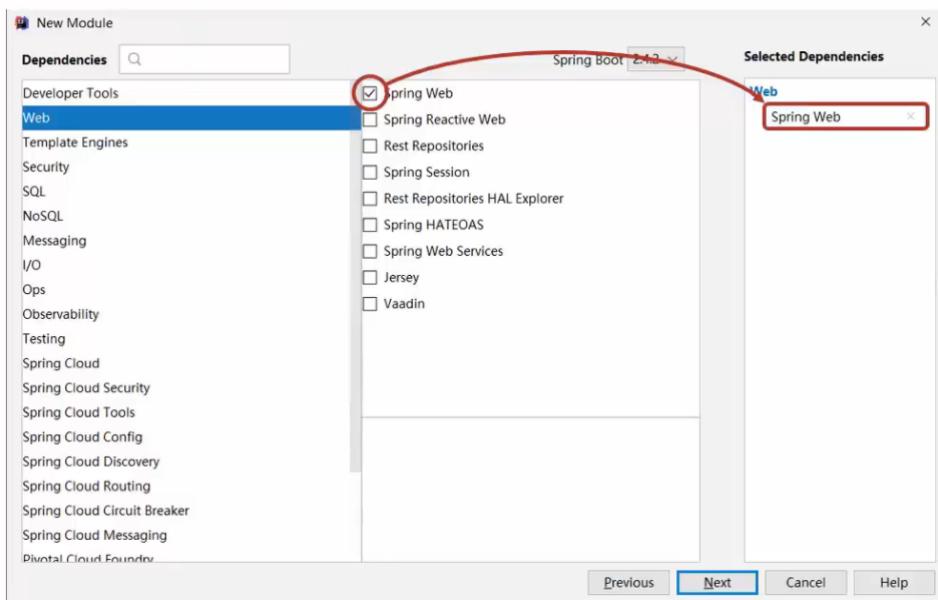
开发与启动

1. SpringBoot简单开发步骤

(1) 步骤1: 创建新模块,选择Spring初始化,并配置模块相关信息



(2) 步骤2: 选择当前模块需要使用的技术集



(3) 步骤3：直接开发Controller层

(4) 步骤4：运行SpringBoot自动生成的Application类

2. 官网构建工程

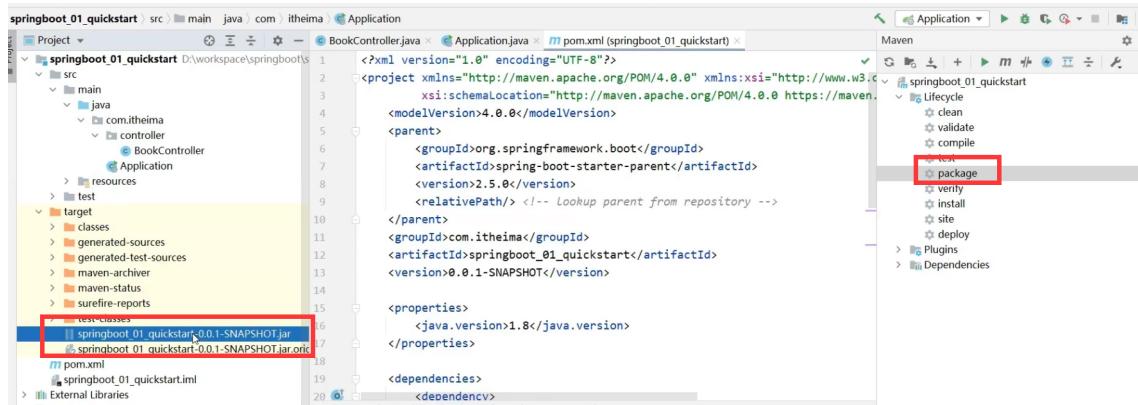
The screenshot shows the Spring Initializr website. It has several configuration sections: 'Project' (Maven Project selected), 'Language' (Java selected), 'Spring Boot' (2.5.0 selected), 'Project Metadata' (Group: com.itheima, Artifact: springboot_01_quickstart, Name: springboot_01_quickstart, Description: Demo project for Spring Boot, Package name: com.itheima, Packaging: Jar), and 'Dependencies' (Spring Web selected). At the bottom are buttons for 'GENERATE' (CTRL + D), 'EXPLORE' (CTRL + SPACE), and 'SHARE...'.

3. SpringBoot项目快速启动

SpringBoot程序可以不依赖Tomcat、IDEA等工具，直接运行。用快速启动的方式，使得前端也能运行SpringBoot程序。

启动步骤：

(1) 步骤1：使用Maven的package指令，会在target目录下生成jar包



要注意，打jar包之前，需要确定pom.xml中有如下的插件

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

(2) 步骤2：启动

进入步骤1中jar包所在的位置，在那个路径下打开cmd，输入指令：

`jar -jar jar包名称`

举例：

`jar -jar springboot_01_quickstart-0.0.1-SNAPSHOT.jar`

概述

1. 起步依赖

SpringBoot的启动主要依靠两部分的依赖。

```
<?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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.0</version>
    </parent>
    <groupId>com.itheima</groupId>
    <artifactId>springboot-01-quickstart</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>
</project>
```

第一部分的starter继承于父工程parent，其父工程如下，parent又继承自更上一层的父工程dependencies。

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-dependencies</artifactId>
    <version>2.5.0</version>
  </parent>
  <artifactId>spring-boot-starter-parent</artifactId>
  <packaging>pom</packaging>
  ...
</project>

```

dependencies如下，其中包含两部分，分别是properties（包含很多版本信息）和很多依赖管理。

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>2.5.0</version>
  <packaging>pom</packaging>
  <properties>
    <servlet-api.version>4.0.1</servlet-api.version>
    ...
  </properties>
</project>

```

```

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>${junit.version}</version>
</dependency>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>${servlet-api.version}</version>
</dependency>

```

第二部分的starter也包含很多依赖，如Tomcat服务器等启动web需要的依赖。

2. 小结

starter: SpringBoot中常见项目名称，定义了当前项目使用的所有项目坐标，以达到减少依赖配置的目的。

parent: 所有SpringBoot项目要继承的项目，定义了若干个坐标版本号（依赖管理，而非依赖），以达到减少依赖冲突的目的。

实际开发中使用任意坐标时，仅书写GAV中的G和A，V由SpringBoot提供。

3. 启动类

SpringBoot的引导类是项目的入口，运行main方法就可以启动项目。

配置文件

配置文件格式

1. 文件格式

SpringBoot中配置文件格式分为三类：

(1) application.properties

```
server.port=80
```

(2) application.yml

```
server:
  port: 81
```

(3) application.yaml

```
server:  
  port: 82
```

SpringBoot程序配置文件名必须是application。yml和yaml文件写法相同，只是后缀名不同。

2. 三种配置文件的优先级

application.properties > application.yml > application.yaml

yaml

1. 格式与语法规则

yaml是一种数据序列化格式，以数据为核心，重数据轻格式。

yaml可以以.yml为后缀，也可以以.yaml为后缀。

语法格式：

```
#1. 只允许加空格，不允许叫tab  
#2. 属性值前面添加空格  
#3. 数组数据在数据书写位置的下方使用减号作为数据开始符号，每行书写一个数据  
enterprise:  
  name: itcast  
  age: 16  
  tel: 4006184000  
  subject:  
    - Java  
    - 前端  
    - 大数据
```

2. yaml配置文件数据读取

指的是在程序中读取yaml配置文件中的数据。假设要读取的配置文件如下：

```
lesson: SpringBoot
```

```
server:  
  port: 80  
  
enterprise:  
  name: itcast  
  age: 16  
  tel: 4006184000  
  subject:  
    - Java  
    - 前端  
    - 大数据
```

(1) 方法1：使用@Value注解直接读取

注解中用于读取属性名引用方式是：\${一级属性名.二级属性名.....}

```
@RestController  
@RequestMapping("/books")  
public class BookController {  
  
    @Value("${lesson}")  
    private String lesson;  
    @Value("${server.port}")  
    private Integer port;  
    @Value("${enterprise.subject[0]}")  
    private String subject_00;  
  
    @GetMapping("/{id}")  
    public String getById(@PathVariable Integer id){  
        System.out.println(lesson);  
        System.out.println(port);  
        System.out.println(subject_00);  
        return "hello , spring boot!";  
    }  
}
```

(2) 方法2：使用Environment对象读取（实际开发不常用）

Environment对象封装了配置文件中的所有数据，需要哪个数据就使用Environment对象的getProperty(String name)方法获取。

```
@RestController  
@RequestMapping("/books")
```

```

public class BookController {

    @Autowired
    private Environment env;

    @GetMapping("/{id}")
    public String getById(@PathVariable Integer id){
        System.out.println(env.getProperty("lesson"));

        System.out.println(env.getProperty("enterprise.name"));

        System.out.println(env.getProperty("enterprise.subject[0]"));
        return "hello , spring boot!";
    }
}

```

(3) 方法3：自定义对象接收和读取（最常用）

yaml是进行数据配置的，该方法就是将数据配置转换为数据对象。

假设定义一个实体类来接收配置文件中的enterprise属性

```

@Component //将实体类的bean交给Spring管理
@ConfigurationProperties(prefix = "enterprise") //表示加载配置文件，prefix属性指定加载配置文件中哪个属性
public class Enterprise {
    private String name;
    private int age;
    private String tel;
    private String[] subject;

    //get set toString方法 省略
}

```

```

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

    @Autowired
    private Enterprise enterprise;

    @GetMapping("/{id}")
    public String getById(@PathVariable Integer id){
        System.out.println(enterprise.getName());
        System.out.println(enterprise.getAge());
        System.out.println(enterprise.getSubject());
    }
}

```

```
        System.out.println(enterprise.getTel());
        System.out.println(enterprise.getSubject()[0]);
        return "hello , spring boot!";
    }
}
```

如果使用方法3在实体类上出现警告，需要在pom.xml中添加如下依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-
processor</artifactId>
    <optional>true</optional>
</dependency>
```

多环境配置&配置文件分类

1. yaml多环境配置

```
#设置启用的环境
spring:
  profiles:
    active: dev #表示使用的是开发环境的配置
#开发
spring:
  config:
    activate:
      on-profile: dev
server:
  port: 80

#生产
spring:
  config:
    activate:
      on-profile: pro
server:
  port: 81

#生产
spring:
  config:
    activate:
      on-profile: test
server:
```

port: 82

2. properties多文件配置

需要用多个文件分别配置。

- 主启动配置文件application.properties

```
spring.profiles.active=pro
```

- 环境分类配置文件application-pro.properties

```
server.port=80
```

- 环境分类配置文件application-dev.properties

```
server.port=81
```

- 环境分类配置文件application-test.properties

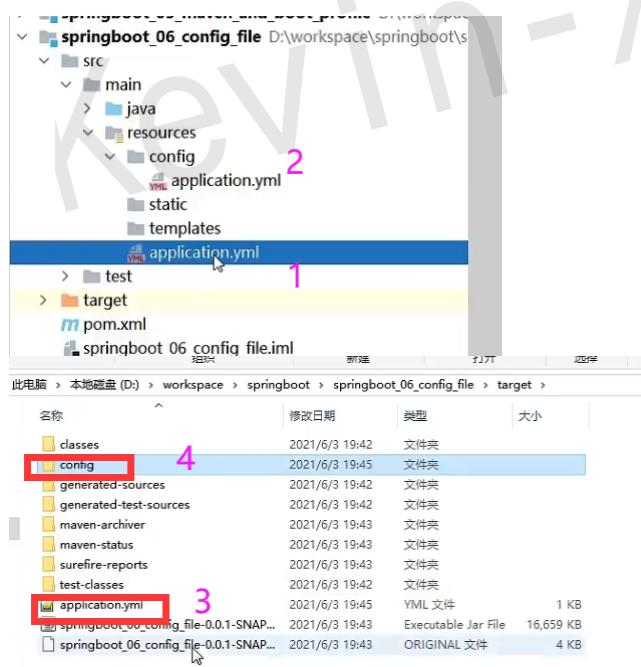
```
server.port=82
```

3. 命令行启动参数设置

在SpringBoot简介中，讲解了SpringBoot的启动。在用命令行进行启动时，还可以设置参数指定环境与端口。

```
java -jar springboot.jar --server.port=88 --  
spring.profiles.active=test
```

4. 配置文件分类



上述四处都可以写配置文件，优先级为4>3>2>1。

SpringBoot整合

1. 整合Junit

使用@SpringBootTest注解

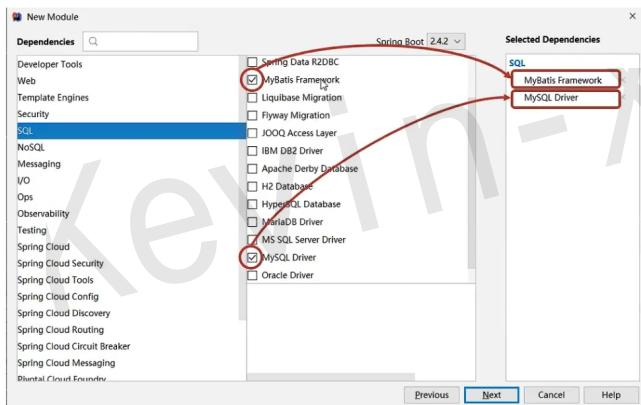
```
@SpringBootTest
class Springboot07TestApplicationTests {

    @Autowired
    private BookService bookService;

    @Test
    public void save() {
        bookService.save();
    }
}
```

2. 整合Mybatis

(1) 步骤1：选择要使用的技术集



(2) 步骤2：设置数据源参数

在 `application.yml` 配置文件中配置如下内容

```
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ssm_db
    username: root
    password: root
```

(3) 步骤3：定义数据层接口与映射配置

要加上@Mapper注解。

```

@Mapper
public interface BookDao {
    @Select("select * from tb1_book where id = #{id}")
    public Book getById(Integer id);
}

```

MyBatisPlus

概述与入门使用

1. 简介

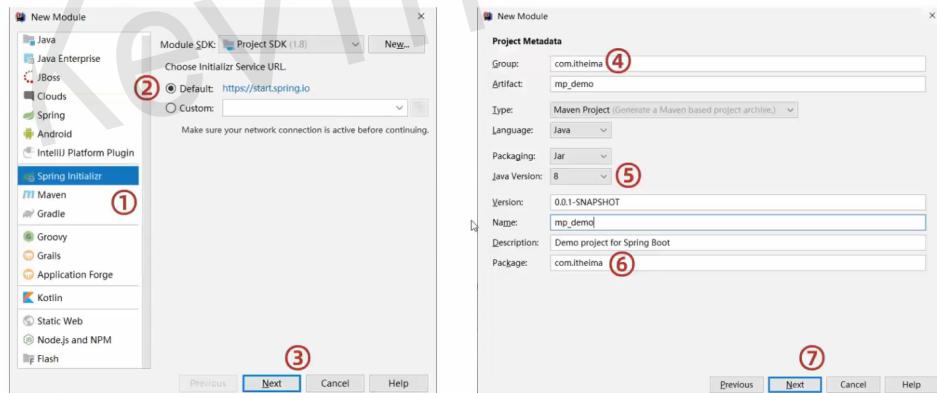
MyBatisPlus（简称MP）是基于MyBatis框架基础上开发的增强型工具，旨在简化开发、提高效率。

MP是MyBatis的一套增强工具，它是在MyBatis的基础上进行开发的，我们虽然使用MP但是底层依然是MyBatis的东西，也就是说我们也可以在MP中写MyBatis的内容。

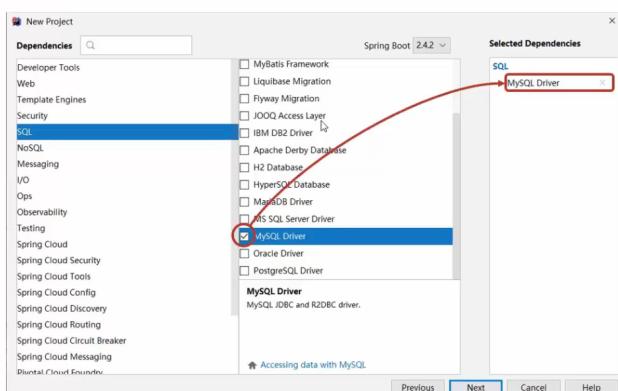
2. 入门使用

(1) 步骤1：创建新模块

①：创建新模块，选择Spring初始化，并配置模块相关基础信息



(2) 步骤2：选择技术集（仅保留JDBC，不保留MyBatis）



(3) 步骤3：手动添加MP起步依赖

```
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.1</version>
</dependency>
```

(4) 步骤4：设置JDBC参数

④：设置Jdbc参数 (`application.yml`)

```
spring:
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/mybatisplus_db?serverTimezone=UTC
    username: root
    password: root
```

数据库名称

(5) 步骤5：制作实体类与表结构（注意！注意！注意！实体类名要与表名相同，属性名要和字段名相同）

```
CREATE TABLE `user` (
  `id`      bigint(20) NOT NULL AUTO_INCREMENT COMMENT '编号',
  `name`     varchar(32) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '用户名',
  `password` varchar(32) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '密码',
  `age`      int(3) NOT NULL COMMENT '龄年',
  `tel`      varchar(32) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '电话',
  PRIMARY KEY (`id`)
)
```

```
public class User {
    private Long id;
    private String name;
    private String password;
    private Integer age;
    private String tel;
}
```

(6) 步骤6：定义数据接口，继承BaseMapper<实体类>

```
@Mapper
public interface UserDao extends BaseMapper<User> { }
```

(7) 步骤7：注入Dao接口进行功能开发

标准数据层开发

标准CRUD与lombok

1. 标准CRUD制作

MP中自带了很多方法，能帮助我们完成基础的CRUD。下图展示了一些基本的方法。

功能	自定义接口	MP接口
新增	boolean save(T t)	int insert(T t)
删除	boolean delete(int id) ↳	int deleteById(Serializable id)
修改	boolean update(T t)	int updateById(T t)
根据id查询	T getById(int id)	T selectById(Serializable id)
查询全部	List<T> getAll()	List<T> selectList()
分页查询	PageInfo<T> getAll(int page, int size)	IPage<T> selectPage(IPage<T> page)
按条件查询	List<T> getAll(Condition condition)	IPage<T> selectPage(Wrapper<T> queryWrapper)

```

@SpringBootTest
class Mybatisplus01QuickstartApplicationTests {

    @Autowired
    private UserDao userDao;

    //新增，返回值Int，新增成功返回1，不成功返回0
    @Test
    void testSave() {
        User user = new User();
        user.setName("黑马程序员");
        user.setPassword("itheima");
        user.setAge(12);
        user.setTel("4006184000");
        userDao.insert(user);
    }

    //删除，入参是Serializable，能接收所有可以作为主键的类型
    @Test
    void testDelete() {
        userDao.deleteById(1401856123725713409L);
    }

    //修改，MP的修改可以自定义修改哪些属性，不设置的属性默认不会被修改
    @Test
    void testUpdate() {
        User user = new User();
        user.setId(1L);
        user.setName("Tom888");
        user.setPassword("tom888");
        userDao.updateById(user);
    }

    //根据ID查询
    @Test
    void test GetById() {
}

```

```
User user = userDao.selectById(2L);
System.out.println(user);
}

//查询所有
@Test
void testGetAll() {
    List<User> userList = userDao.selectList(null);
    System.out.println(userList);
}
}
```

2. Lombok

Lombok是一个Java类库，提供了一组注解，简化POJO实体类开发。

使用步骤：

(1) 步骤1：添加Lombok依赖

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <!--<version>1.18.12</version>-->
</dependency>
```

(2) 步骤2：安装Lombok插件（新版本IDEA已内置）

(3) 步骤3：使用Lombok提供的注解

Lombok常见的注解有：

- @Setter:为模型类的属性提供setter方法
- @Getter:为模型类的属性提供getter方法
- @ToString:为模型类的属性提供toString方法
- @EqualsAndHashCode:为模型类的属性提供equals和hashcode方法
- @Data:是个组合注解，包含上面的注解的功能
- @NoArgsConstructor:提供一个无参构造函数
- @AllArgsConstructor:提供一个包含所有参数的构造函数

```
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class User {  
    private Long id;  
    private String name;  
    private String password;  
    private Integer age;  
    private String tel;  
}
```

说明：Lombok只是简化模型类的编写，我们之前的方法也能用，比如只想要有name和password的构造函数，也是允许的：

```
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class User {  
    private Long id;  
    private String name;  
    private String password;  
    private Integer age;  
    private String tel;  
  
    public User(String name, String password) {  
        this.name = name;  
        this.password = password;  
    }  
}
```

标准分页功能制作

1. 分页功能介绍

分页指在数据库查询后，将查询结果分为不同页，我们可以指定看哪页、该页的多少条数据。

分页查询使用的方法是：

```
IPage<T> selectPage(IPage<T> page, wrapper<T> querywrapper)
```

- IPage：用来构建分页查询条件
- Wrapper：用来构建条件查询的条件，目前我们没有可直接传为Null
- IPage：返回值，构建分页条件和方法的返回值都是IPage

IPage是一个接口，其实现类为Page。

2. 使用步骤

(1) 步骤1：设置分页拦截器作为Spring管理的bean

设置分页拦截器后，分页功能才会启动。

```
@Configuration
public class MybatisPlusConfig {

    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor(){
        //1 创建MybatisPlusInterceptor拦截器对象
        MybatisPlusInterceptor mpInterceptor=new
        MybatisPlusInterceptor();
        //2 添加分页拦截器
        mpInterceptor.addInnerInterceptor(new
        PaginationInnerInterceptor());
        return mpInterceptor;
    }
}
```

(2) 步骤2：执行分页查询

```
@SpringBootTest
class Mybatisplus01QuickstartApplicationTests {

    @Autowired
    private UserDao userDao;

    //分页查询
    @Test
    void testSelectPage(){
        //1 创建IPage分页对象，设置分页参数，1为选择要显示的是哪页的数据,
        3为每页显示的记录数
        IPage<User> page=new Page<>(1,3);
        //2 执行分页查询
        userDao.selectPage(page,null);
        //3 获取分页结果
        System.out.println("当前页码值: "+page.getCurrent());
        System.out.println("每页显示数: "+page.getPageSize());
        System.out.println("一共多少页: "+page.getPages());
        System.out.println("一共多少条数据: "+page.getTotal());
        System.out.println("数据: "+page.getRecords());
    }
}
```

```
}
```

DQL编程控制

查询条件格式

1. 构建条件查询

MP中用条件构造器Wrapper来进行条件构造。

(1) 方式1: QueryWrapper

```
@SpringBootTest
class Mybatisplus02DqlApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void testGetAll(){
        QueryWrapper qw = new QueryWrapper();
        qw.lt("age",18);
        List<User> userList = userDao.selectList(qw); //依然用
        MP自带方法，里面可以放置参数为wrapper
        System.out.println(userList);
    }
}
```

(2) 方式2: QueryWrapper基础上使用Lambda

```
@SpringBootTest
class Mybatisplus02DqlApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void testGetAll(){
        QueryWrapper<User> qw = new QueryWrapper<User>();
        qw.lambda().lt(User::getAge, 10); //添加条件
        List<User> userList = userDao.selectList(qw);
        System.out.println(userList);
    }
}
```

(3) 方式3: 直接使用LambdaQueryWrapper

```

@SpringBootTest
class Mybatisplus02DqlApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void testGetAll(){
        LambdaQueryWrapper<User> lqw = new
        LambdaQueryWrapper<User>();
        lqw.lt(User::getAge, 10);
        List<User> userList = userDao.selectList(lqw);
        System.out.println(userList);
    }
}

```

2. 多条件构造

使用条件构造器时，支持链式编程。

(1) and

```

LambdaQueryWrapper<User> lqw = new LambdaQueryWrapper<User>();
lqw.lt(User::getAge, 30).gt(User::getAge, 10);
List<User> userList = userDao.selectList(lqw);
System.out.println(userList);

```

(2) or

```

LambdaQueryWrapper<User> lqw = new LambdaQueryWrapper<User>();
lqw.lt(User::getAge, 10).or().gt(User::getAge, 30);
List<User> userList = userDao.selectList(lqw);
System.out.println(userList);

```

3. null判定

当条件构造器里的参数为null时，将查询不出任何数据，因此在查询时需要对null进行判定。

lt()、gt()等方法均有两种入参，支持对null进行判定。

```

@SpringBootTest
class Mybatisplus02DqlApplicationTests {

    @Autowired
    private UserDao userDao;

```

```

    @Test
    void testGetAll(){
        //模拟页面传递过来的查询数据
        UserQuery uq = new UserQuery();
        uq.setAge(10);
        uq.setAge2(30);

        LambdaQueryWrapper<User> lqw = new
        LambdaQueryWrapper<User>();
        lqw.lt(null!=uq.getAge2(),User::getAge, uq.getAge2());
        lqw.gt(null!=uq.getAge(),User::getAge, uq.getAge());
        List<User> userList = userDao.selectList(lqw);
        System.out.println(userList);
    }
}

```

查询投影

查询投影可以对应SQL语句中select后面的部分

1. 查询指定字段

(1) LambdaQueryWrapper

```

@SpringBootTest
class Mybatisplus02DqlApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void testGetAll(){
        LambdaQueryWrapper<User> lqw = new
        LambdaQueryWrapper<User>();
        lqw.select(User::getId,User::getName,User::getAge);
        List<User> userList = userDao.selectList(lqw);
        System.out.println(userList);
    }
}

```

(2) QueryWrapper (需指定字段)

```

@SpringBootTest
class Mybatisplus02DqlApplicationTests {

    @Autowired

```

```

private UserDao userDao;

@Test
void testGetAll(){
    QueryWrapper<User> lqw = new QueryWrapper<User>();
    lqw.select("id", "name", "age", "tel");
    List<User> userList = userDao.selectList(lqw);
    System.out.println(userList);
}

```

2. 聚合函数查询

即完成count、 max、 min、 arg、 sum的查询，聚合和分组无法使用Lambda完成。

```

@SpringBootTest
class Mybatisplus02DqlApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void testGetAll(){
        QueryWrapper<User> lqw = new QueryWrapper<User>();
        //lqw.select("count(*) as count");
        //SELECT count(*) as count FROM user
        //lqw.select("max(age) as maxAge");
        //SELECT max(age) as maxAge FROM user
        //lqw.select("min(age) as minAge");
        //SELECT min(age) as minAge FROM user
        //lqw.select("sum(age) as sumAge");
        //SELECT sum(age) as sumAge FROM user
        lqw.select("avg(age) as avgAge");
        //SELECT avg(age) as avgAge FROM user

        //需要用Map来接收，因为聚合结果是多出来的字段
        List<Map<String, Object>> userList =
        userDao.selectMaps(lqw);
        System.out.println(userList);
    }
}

```

3. 分组查询

即完成group by的功能。

```

@SpringBootTest
class Mybatisplus02DqlApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void testGetAll(){
        QueryWrapper<User> lqw = new QueryWrapper<User>();
        lqw.select("count(*) as count,tel");
        lqw.groupBy("tel");
        List<Map<String, Object>> list =
        userDao.selectMaps(lqw);
        System.out.println(list);
    }
}

```

```
SELECT count(*) as count, tel FROM user GROUP BY tel
```

查询条件设置

1. 等值条件

eq()方法，selectList的查询结果为多个或者单个，selectOne的查询结果为单个。

```

@SpringBootTest
class Mybatisplus02DqlApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void testGetAll(){
        LambdaQueryWrapper<User> lqw = new
        LambdaQueryWrapper<User>();
        lqw.eq(User::getName, "Jerry").eq(User::getPassword,
        "jerry");
        User loginUser = userDao.selectOne(lqw);
        System.out.println(loginUser);
    }
}

```

2. 范围查询

- gt(): 大于(>)
- ge(): 大于等于(>=)
- lt(): 小于(<)
- lte(): 小于等于(<=)
- between(): between ? and ?

3. 模糊查询

- like(): 前后加百分号, 如 %j%
- likeLeft(): 前面加百分号, 如 %j
- likeRight(): 后面加百分号, 如 j%

4. 排序查询

```

@SpringBootTest
class Mybatisplus02DqlApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void testGetAll(){
        LambdaQueryWrapper<User> lqw = new
        LambdaQueryWrapper<>();
        /**
         * condition : 条件, 返回boolean,
         * 当condition为true, 进行排序, 如果为false, 则不排序
         * * isAsc: 是否为升序, true为升序, false为降序
         * * columns: 需要操作的列, 可以是多列
         */
        //orderby(condition,isAsc,columns)
        lqw.orderBy(true, false, User::getId);

        userDao.selectList(lw
    }
}

```

映射匹配兼容性

1. 表名与实体类不一致

解决方法：在实体类上加上@TableName注解，value属性写数据库中的表名。

```
@Data  
@TableName("tbl_user")  
public class User {  
    private Long id;  
    private String name;  
    private String password;  
    private Integer age;  
    private String tel;  
}
```

2. 表中字段与实体类中属性不一致

解决方法：在属性上加上@TableField注解，value属性写数据库字段的名称，使用该注解可以实现模型类属性名和表的列名之间的映射关系。

```
@Data  
@TableName("tbl_user")  
public class User {  
    private Long id;  
    private String name;  
    @TableField(value="pwd")  
    private String password;  
    private Integer age;  
    private String tel;  
}
```

3. 实体类定义了数据库中未定义的属性

由于代码的需要，有时要在实体类中加上新的属性。但在执行select等语句时，会select到表中没有的新属性导致错误。

解决方法：@TableField注解的exist属性，设置该字段是否在数据库表中存在，如果设置为false则不存在，生成sql语句查询的时候，就不会再查询该字段了。

```
@Data  
@TableName("tbl_user")  
public class User {  
    private Long id;  
    private String name;  
    @TableField(value="pwd")  
    private String password;  
    private Integer age;  
    private String tel;  
    @TableField(exist=false)  
    private Integer online;  
}
```

4. 投影的字段可能隐秘

查询表中所有的列的数据，就可能把一些敏感数据查询到返回给前端，这个时候我们就需要限制哪些字段默认不要进行查询。

解决方案：@TableField注解的一个属性叫select，该属性设置默认是否需要查询该字段的值，true(默认值)表示默认查询该字段，false表示默认不查询该字段。此时该属性仍能当做条件，但不会被投影。

```
@Data  
@TableName("tbl_user")  
public class User {  
    private Long id;  
    private String name;  
    @TableField(value="pwd", select=false)  
    private String password;  
    private Integer age;  
    private String tel;  
    @TableField(exist=false)  
    private Integer online;  
}
```

5. 相关注解

@TableField

名称	@TableField
类型	==属性注解==
位置	模型类属性定义上方
作用	设置当前属性对应的数据表中的字段关系
相关属性	value(默认): 设置数据库表字段名称 exist:设置属性在数据库表字段中是否存在，默认为true，此属性不能与value合并使用 select:设置属性是否参与查询，此属性与select()映射配置不冲突

@TableName

名称	@TableName
类型	==类注解==
位置	模型类定义上方

名称	@TableName
作用	设置当前类对应于数据库表关系
相关属性	value(默认): 设置数据库表名称

DML编程控制

id生成策略

1. 四种ID生成策略

(1) AUTO策略

AUTO的作用是使用数据库ID自增，在使用该策略的时候一定要确保对应的数据表设置了ID主键自增，否则无效。

```
@Data  
@TableName("tbl_user")  
public class User {  
    @TableId(type = IdType.AUTO)  
    private Long id;  
    private String name;  
    @TableField(value="pwd", select=false)  
    private String password;  
    private Integer age;  
    private String tel;  
    @TableField(exist=false)  
    private Integer online;  
}
```

(2) INPUT策略

需要将表的自增策略删除掉，同时在插入数据时设置主键ID的值。

```
@Data  
@TableName("tbl_user")  
public class User {  
    @TableId(type = IdType.INPUT)  
    private Long id;  
    private String name;  
    @TableField(value="pwd",select=false)  
    private String password;  
    private Integer age;  
    private String tel;  
    @TableField(exist=false)  
    private Integer online;  
}
```

```
@SpringBootTest  
class Mybatisplus03DqlApplicationTests {  
  
    @Autowired  
    private UserDao userDao;  
  
    @Test  
    void testSave(){  
        User user = new User();  
        //设置主键ID的值  
        user.setId(666L);  
        user.setName("黑马程序员");  
        user.setPassword("itheima");  
        user.setAge(12);  
        user.setTel("4006184000");  
        userDao.insert(user);  
    }  
}
```

(3) ASSIGN_ID策略

默认的策略，使用雪花算法生成ID。生成的ID是一个Long类型的数据。

```

@Data
@TableName("tbl_user")
public class User {
    @TableId(type = IdType.ASSIGN_ID)
    private Long id;
    private String name;
    @TableField(value="pwd", select=false)
    private String password;
    private Integer age;
    private String tel;
    @TableField(exist=false)
    private Integer online;
}

```

(4) ASSIGN_UUID策略

数据库中主键的类型需要设置为varchar，因为UUID策略生成的ID是32位的String。

2. 雪花算法

生成的是64位大小的整数

占位符: 0	0 00100110111011010101100001101010011000110 10000 10001 000000000010		
	时间戳(41)	机器码(5+5)	序列号(12)

3. ID生成策略对比

- NONE: 不设置id生成策略，MP不自动生成，约等于INPUT，所以这两种方式都需要用户手动设置，但是手动设置第一个问题是容易出现相同的ID造成主键冲突，为了保证主键不冲突就需要做很多判定，实现起来比较复杂。
- AUTO: 数据库ID自增，这种策略适合在数据库服务器只有1台的情况下使用，不可作为分布式ID使用。
- ASSIGN_UUID: 可以在分布式的情况下使用，而且能够保证唯一，但是生成的主键是32位的字符串，长度过长占用空间而且还不能排序，查询性能也慢。
- ASSIGN_ID: 可以在分布式的情况下使用，生成的是Long类型的数字，可以排序性能也高，但是生成的策略和服务器时间有关，如果修改了系统时间就有可能导致出现重复主键。

4. 简化配置

如果想要为每个实体类设置统一的ID生成策略，在配置文件中添加如下内容：

```
mybatis-plus:  
  global-config:  
    db-config:  
      id-type: assign_id
```

如果数据库表都以某前缀开头，如tb1_，可以设置统一前缀，这样在实体类上@TableName中就不需要加前缀：

```
mybatis-plus:  
  global-config:  
    db-config:  
      table-prefix: tb1_
```

5. @TableId

名称	@TableId
类型	==属性注解==
位置	模型类中用于表示主键的属性定义上方
作用	设置当前类中主键属性的生成策略
相关属性	value(默认): 设置数据库表主键名称 type:设置主键属性的生成策略，值查照IdType的枚举值

批量删除与查询

API

```
int deleteBatchIds(@Param(Constants.COLLECTION) Collection<? extends Serializable> idList);  
  
List<T> selectBatchIds(@Param(Constants.COLLECTION) Collection<? extends Serializable> idList);
```

将ID组成的集合传入，可实现批量删除这些ID。

```
@SpringBootTest  
class Mybatisplus03DqlApplicationTests {  
  
  @Autowired  
  private UserDao userDao;  
  
  @Test  
  void testDelete(){
```

```

//删除指定多条数据
List<Long> list = new ArrayList<>();
list.add(1402551342481838081L);
list.add(1402553134049501186L);
list.add(1402553619611430913L);
userDao.deleteBatchIds(list);
userDao.selectBatchIds(list);
}
}

```

逻辑删除

1. 概念

逻辑删除：为数据设置是否可用状态字段，删除时设置状态字段为不可用状态，数据保留在数据库中。在MP使用delete命令，但实际的sql语句执行的是update操作。

进行逻辑删除的数据，无法通过MP的select查询出来，需要自己写sql语句进行查询。

2. 逻辑删除步骤

(1) 步骤1：修改数据库表，添加一列作为标记字段

需要在数据库中为该字段设置默认值。

字段	索引	外键	触发器	选项	注释	SQL 预览	
名							
id							
name							
pwd							
age							
tel							
* deleted							

(2) 步骤2：实体类添加属性，并用@TableLogic设定其为逻辑删除标识字段

添加与数据库表的列对应的一个属性名，名称可以任意，如果和数据表列名对不上，可以使用@TableField进行关系映射，如果一致，则会自动对应。

使用@TableLogic标识新增的字段为逻辑删除字段。

```

@Data
//@TableName("tbl_user") 可以不写是因为配置了全局配置
public class User {
    @TableId(type = IdType.ASSIGN_UUID)
    private String id;
    private String name;
    @TableField(value="pwd", select=false)
    private String password;
}

```

```
private Integer age;
private String tel;
@TableField(exist=false)
private Integer online;
@TableLogic(value="0",delval="1")
//value为正常数据的值, delval为删除数据的值
private Integer deleted;
}
```

(3) 步骤3：运行删除方法

从代码上看，执行的是delete命令；但查看其sql语句，发现实际执行的是update。

```
@SpringBootTest
class Mybatisplus03DqlApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void testDelete(){
        userDao.deleteById(1L);
    }
}
```

3. 逻辑删除后的查询

逻辑删除后的数据无法用MP进行查询，需要重写sql语句

```
@Mapper
public interface UserDao extends BaseMapper<User> {
    //查询所有数据包含已经被删除的数据
    @Select("select * from tbl_user")
    public List<User> selectAll();
}
```

4. 为所有表配置逻辑删除

如果每个表都要有逻辑删除，可以在配置文件中添加全局配置，这样就不需要在每个实体类的属性上添加@TableLogic注解。

```

mybatis-plus:
  global-config:
    db-config:
      # 逻辑删除字段名，需要每张表的逻辑删除标识字段都设置为deleted
      logic-delete-field: deleted
      # 逻辑删除字面值：未删除为0
      logic-not-delete-value: 0
      # 逻辑删除字面值：删除为1
      logic-delete-value: 1

```

5. @TableLogic

名称	@TableLogic
类型	==属性注解==
位置	模型类中用于表示删除字段的属性定义上方
作用	标识该字段为进行逻辑删除的字段
相关属性	value: 逻辑未删除值 delval:逻辑删除值

乐观锁

1. 概念

在秒杀问题中，如果仅剩一件物品而有很多人抢购时，若没有锁会导致商品数量出现负数，此时需要加锁。用乐观锁能处理少量的并发处理。

2. 实现步骤

- (1) 步骤1：数据库表添加一列，设置默认值

字段	索引	外键	触发器	选项	注释	SQL 预览	
名	类型	长度	小数点	不是 null	虚拟	键	注释
id	bigint	20	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	 1	
name	varchar	32	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
pwd	varchar	32	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
age	int	3	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
tel	varchar	32	0	<input type="checkbox"/>	<input type="checkbox"/>		
deleted	int	1	0	<input type="checkbox"/>	<input type="checkbox"/>		
version	int	11	0	<input type="checkbox"/>	<input type="checkbox"/>		

添加新列



设置默认值

默认:	<input type="text" value="1"/>
<input type="checkbox"/> 自动递增	
<input type="checkbox"/> 无符号	
<input type="checkbox"/> 填充零	

(2) 步骤2：在实体类中添加对应的属性，并加上@Version注解表示是个乐观锁

```

@Data
//@TableName("tbl_user") 可以不写是因为配置了全局配置
public class User {
    @TableId(type = IdType.ASSIGN_UUID)
    private String id;
    private String name;
    @TableField(value="pwd", select=false)
    private String password;
    private Integer age;
    private String tel;
    @TableField(exist=false)
    private Integer online;
    private Integer deleted;
    @Version
    private Integer version;
}

```

(3) 步骤3：添加乐观锁的拦截器

```

@Configuration
public class MpConfig {
    @Bean
    public MybatisPlusInterceptor mpInterceptor() {
        //1. 定义Mp拦截器
        MybatisPlusInterceptor mpInterceptor = new
MybatisPlusInterceptor();
        //2. 添加乐观锁拦截器
        mpInterceptor.addInnerInterceptor(new
OptimisticLockerInnerInterceptor());
        return mpInterceptor;
    }
}

```

(4) 步骤4：执行更新，要注意使用乐观锁机制在修改前必须先获取到对应数据的version

```

@SpringBootTest
class Mybatisplus03DqlApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void testUpdate(){
        //1. 先通过要修改的数据id将当前数据查询出来
        User user = userDao.selectById(3L);          //version=3
        User user2 = userDao.selectById(3L);          //version=3
        user2.setName("Jock aaa");
        userDao.updateById(user2);                   //version=>4
        user.setName("Jock bbb");
        userDao.updateById(user);                   //version=3? 条件
还成立吗？
    }
}

```

其原理是设置@Version后，底层sql语句执行更新时会检查version字段。

执行修改前先执行查询语句：

```
SELECT id,name,age,tel,deleted,version FROM tbl_user WHERE id=?
```

执行修改时使用version字段作为乐观锁检查依据

```
UPDATE tbl_user SET name=?, age=?, tel=?, version=? WHERE id=? AND version=?
```

IService

之前的接口和实现类为：

```
public interface UserService{
    public List<User> findAll();
}

@Service
public class UserServiceImpl implements UserService{
    @Autowired
    private UserDao userDao;

    public List<User> findAll(){
        return userDao.selectList(null);
    }
}
```

MP提供了一个Service接口和实现类，分别是IService和ServiceImpl，后者是对前者的一个具体实现。

```
public interface UserService extends IService<User>{

}

@Service
public class UserServiceImpl extends ServiceImpl<UserDao, User>
implements UserService{

}
```

修改以后的好处是，MP已经帮我们把业务层的一些基础的增删改查都已经实现了，可以直接进行使用。

编写测试类进行测试：

```
@SpringBootTest
class Mybatisplus04GeneratorApplicationTests {

    private IUserService userService;

    @Test
    void testFindAll() {
        List<User> list = userService.list();
        System.out.println(list);
    }

}
```

Kevin-Xun