快捷键

自动补全变量名

```
ctrl + alt + v
```

查找类名

```
shfit + shift
```

查找接口的方法

鼠标点击 想查询的 接口 使用 ctrl + h进行查询

```
ctrl+H
```

spring

bean配置

bean基础配置

```
<bean id = "BookDao" class="com.itheima.dao.impl.BookDaoImpl"></bean>
id 表示 bean的 id class表示 bean 的 类型
```

bean别名配置

```
bean 的别名 设置
使用name 设置

在 xml 文件中设置 name 属性
<bean id = "BookDao" class="com.itheima.dao.impl.BookDaoImpl" name="dao"></bean>

在 main 函数中 就能 使用 别名
BookDao bookDao = (BookDao) ctx.getBean("dao");
```

bean作用范围配置

默认设置 bean的作用范围是 singleton

```
<bean id = "BookDao" class="com.itheima.dao.impl.BookDaoImpl" name="dao"
scope="singleton"></bean>
```

创建出来之后的是一个单例效果

```
package com.itheima;

import com.itheima.dao.BookDao;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

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

//        Spring 默认给我们创建的事 一个 单例
//        可以自己设定 成一个 非单例

        BookDao bookdao1 = (BookDao) ctx.getBean("BookDao");
        BookDao bookdao2 = (BookDao) ctx.getBean("BookDao");
        System.out.println(bookdao1);
        System.out.println(bookdao2);
    }
}
```

地址显示 明显是相同的

```
com.itheima.dao.impl.BookDaoImpl@2d928643
com.itheima.dao.impl.BookDaoImpl@2d928643
```

配置bean的作用范围为 可变 prototype

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

```
package com.itheima;

import com.itheima.dao.BookDao;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class AppforScoped {
    public static void main(String [] args){
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
//        Spring 默认给我们创建的事 一个 单例
//        可以自己设定 成一个 非单例

        BookDao bookdao1 = (BookDao) ctx.getBean("BookDao");
```

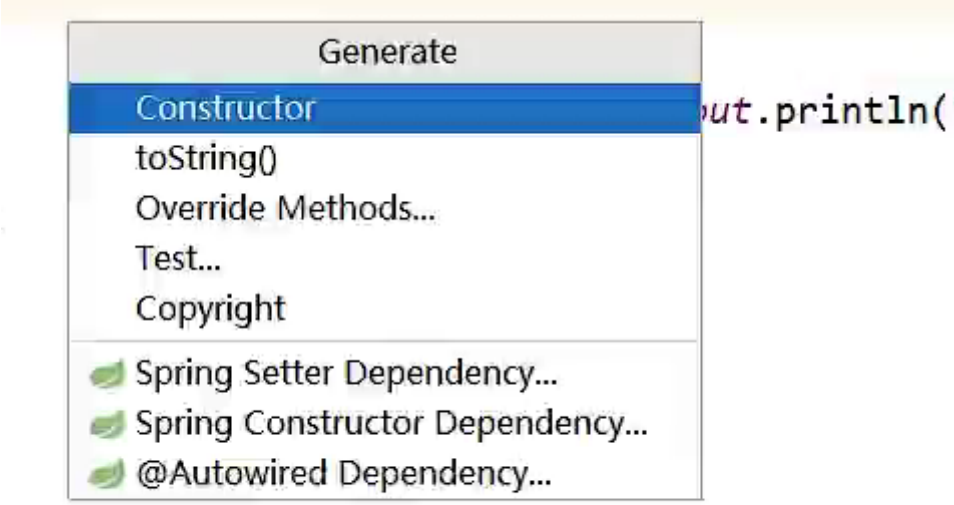
```
        BookDao bookdao2 = (BookDao) ctx.getBean("BookDao");
        System.out.println(bookdao1);
        System.out.println(bookdao2);
    }
}
```

地址显示

```
com.itheima.dao.impl.BookDaoImpl@2d928643
com.itheima.dao.impl.BookDaoImpl@5025a98f
```

bean的构造方法

1. 介绍idea一个快捷键 alt + insert 生成



2. 创建出一个 无参构造函数

```
    public BookDaoImpl() {
        System.out.println("bookdao constructor is running...");
    }
```

3. 生成结果

```
bookdao constructor is running...
book dao save ...
```

4. 注意一点的是 Spring创建构方法的时候 创建的是 无参构造方法，当使用有参构造方法的时候 会报错
5. 使用工厂的形式进行配置
    1. 创建一个OrderDao接口 声明一个 save() 方法

```
package com.itheima.dao;

public interface OrderDao {

    public void save();

}
```

2. 在impl中创建一个类继承OrderDao的接口

```
package com.itheima.dao.impl;

import com.itheima.dao.OrderDao;

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

3. 生成一个工厂 在xml中使用

```
package com.itheima.factory;

import com.itheima.dao.OrderDao;
import com.itheima.dao.impl.OrderDaoImpl;

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

4. 创建一个工厂

```
package com.itheima;

import com.itheima.dao.OrderDao;
import com.itheima.factory.OrderDaoFactory;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class AppForInstanceOrder {

    public static void main(String[] args){

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

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

```
        orderDao.save();

    }
}
```

5. 在xml 中进行配置 factory-method 表示造对象的方法

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

6. 第四种配置 使用factoryBean 进行配置
    1. 在xml 进行配置

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

    2. 创建 UserDaoFactoryBean类

```
package com.itheima.factory;

import com.itheima.dao.UserDao;
import com.itheima.dao.impl.UserDaoImpl;
import org.springframework.beans.factory.FactoryBean;

public class UserDaoFactoryBean implements FactoryBean<UserDao> {

//    代替原始实例工厂中创建对象的方法
//    返回bean实例
    public UserDao getObject() throws Exception {
        return new UserDaoImpl();
    }
//    返回bean指定类型
    public Class<?> getObjectType() {
//        返回UserDao的字节码 UserDao.class
        return UserDao.class;
    }

    public boolean isSingleton() {
//        返回 true 表示 单例 return false 表示 非单例
        return false;
    }
}
```

    3. 创建主函数

```
package com.itheima;

import com.itheima.dao.UserDao;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
public class AppForInstanceUser {

    public static  void main(String[] args){

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

        UserDao userDao1 = (UserDao) ctx.getBean("userDao");
        UserDao userDao2 = (UserDao) ctx.getBean("userDao");

        System.out.println(userDao1);
        System.out.println(userDao2);
//        userDao.save();

    }
}
```

7. bean 的 声明周期

    1. 声明实现 创建和销毁的方法

```
//   对bean进行初始化操作
    public void init(){
        System.out.println("init...");
    }
//    对bean进行销毁操作
    public void destroy(){
        System.out.println("destroy...");
    }
```

    2. 在xml中对 初始化方法和销毁方法进行声明定义

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

    3. 在主函数执行时可以观测到 初始化方法

```
package com.itheima.service;

import com.itheima.dao.BookDao;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class BookServiceImpl {

    public static void main(String[] args){

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

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

        bookDao.save();
    }
```

```
}
```

4. 在虚拟机退出时关闭容器 调用close 方法
   值得注意的一点时 使用close方法的时候 使用的接口是ClassPathXmlApplicationContext 并不是 ApplicationContext 因为
   ApplicationContext 并没用close这个方法

```
  ctx.close();
```

5. 设置关闭钩子

```
ctx.registerShutdownHook()
```

8. 使用接口设置声明周期

```
public class BookDaoImpl implements BookDao , InitializingBean,
DisposableBean {

//    将这个构造方法给写出来
    public BookDaoImpl() {
        System.out.println("bookdao constructor is running...");
    }

    //    实现接口的关键字
    public void save(){
        System.out.println("book dao is save ... ");
    }

    public void destroy() throws Exception {
        System.out.println("service destroy");
    }
//  在属性设置之后
    public void afterPropertiesSet() throws Exception {
        System.out.println("service init");
    }
}
```

依赖注入方式

setter

1. 声明一个接口

```
package com.itheima.dao;

//定义一个接口 BookDao

public interface BookDao {

//     在接口中定义一个函数 这个函数没有实现
    public void save();

}
```

2. 声明一个类继承接口

```
package com.itheima.dao.impl;

import com.itheima.dao.BookDao;

public class BookDaoImpl implements BookDao  {

    public void save(){
        System.out.println("BookDao is running..." + databaseName + "," +
connectionNum);
    }
}
```

3. 使用setter方法

```
package com.itheima.dao.impl;

import com.itheima.dao.BookDao;

public class BookDaoImpl implements BookDao  {

    private int connectionNum;
    private String databaseName;

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

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

    public void save(){
        System.out.println("BookDao is running..." + databaseName + "," +
connectionNum);
    }
}
```

4. 在其它类中调用 引入setter

```
package com.itheima.service.Impl;
```

```
import com.itheima.dao.BookDao;
import com.itheima.dao.UserDao;
import com.itheima.service.BookService;

public class BookServiceImpl implements BookService {

//    定义一个接口 bookDao

    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();
    }


}
```

5. 配置xml文件

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

6. 在相应的实现接口配置xml文件 name 来源于类 private BookDao bookDao 当中的类型 ref对应的是id

```
 <bean id="BookService" class="com.itheima.service.Impl.BookServiceImpl">
        <property name="BookDao" ref="BookDao"></property>
    </bean>
```

7. 在主函数中运行

```
package com.itheima;

import com.itheima.service.BookService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

    public static void main(String[] args){
```

```
        System.out.println("start");

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

        BookService bookService = (BookService) ctx.getBean("BookService");

        bookService.save();
    }
}
```

构造器输入construcor

1. 定义一个接口类

```
package com.itheima.dao;

//定义一个接口 BookDao

public interface BookDao {

//    在接口中定义一个函数 这个函数没有实现
    public void save();
}
```

2. 实现接口类

```
package com.itheima.dao.impl;

import com.itheima.dao.BookDao;

public class BookDaoImpl implements BookDao  {

    private int connectionNum;
    private String databaseName;
//    使用构造器 实现传参
//    最典型写法
    public BookDaoImpl(int connectionNum, String databaseName) {
        this.connectionNum = connectionNum;
        this.databaseName = databaseName;
    }

    public void save(){
        System.out.println("BookDao is running..." + databaseName +","+
connectionNum);
    }
}
```

3. 实现传递

```
package com.itheima.service.Impl;

import com.itheima.dao.BookDao;
```

```
import com.itheima.dao.UserDao;
import com.itheima.service.BookService;

public class BookServiceImpl implements BookService {

//    定义一个接口 bookDao

    private BookDao bookDao;
    private UserDao userDao;

//    使用构造器构造


    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();

    }
}
```

4. 主函数实现调用功能

```
package com.itheima;

import com.itheima.dao.BookDao;
import com.itheima.service.BookService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

    public static void main(String[] args){

        System.out.println("start");

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

        BookService bookService = (BookService) ctx.getBean("BookService");

        bookService.save();
    }
}
```

5. xml配置

    1. 定义BookDao的bean 在bean里面定义构造器传参

```
 <bean id="BookDao" class="com.itheima.dao.impl.BookDaoImpl">
        <constructor-arg name = "databaseName" value = "mysql">
</constructor-arg>
        <constructor-arg name = "connectionNum" value = "10">
</constructor-arg>
    </bean>
```

2. 定义UserDao的bean

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

3. 定义传递中的xml name 是形参名 ref 表示的 指定bean的id名

```
    <bean id="BookService"
class="com.itheima.service.Impl.BookServiceImpl">
<!--         name 是 形参的 变量名-->
       <constructor-arg name = "bookDao" ref = "BookDao"></constructor-
arg>
        <constructor-arg name = "userDao" ref = "UserDao"></constructor-
arg>
    </bean>
```

自动装配

1. 修改xml配置 原来的xml配置

```
<bean id="BookDao" class="com.itheima.dao.impl.BookDaoImpl">
    <constructor-arg index = "1" value = "mysql"></constructor-arg>
    <constructor-arg index="0" value = "10"></constructor-arg>
</bean>

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

<bean id="BookService" class="com.itheima.service.Impl.BookServiceImpl">
    <!--         name 是 形参的 变量名-->
    <constructor-arg name = "bookDao" ref = "BookDao"></constructor-arg>
    <constructor-arg name = "userDao" ref = "UserDao"></constructor-arg>
</bean>
```

2. 修改成自动装配 name是和 private BookDao bookDao; BookDao 的首字母小写
3. autowire 主要使用byName 或者 byType

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

<bean id="BookService" class="com.itheima.service.Impl.BookServiceImpl"
autowire="byName">
</bean>
```

集合注入

1. 创建一个BookDao接口

```
package com.itheima.dao;

//定义一个接口 BookDao

public interface BookDao {

//     在接口中定义一个函数 这个函数没有实现
    public void save();

}
```

2. 实现接口类

```
package com.itheima.dao.impl;

import com.itheima.dao.BookDao;

import java.util.*;

public class BookDaoImpl implements BookDao  {

    private int[] array;

    private List<String> list;

    private Set<String> set;

    private Map<String,String> map;

    private Properties properties;

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

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

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

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

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

```
    public void save(){
        System.out.println("BookDao is running..." );

        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);

    }
}
```

3. xml中配置 数组 List set map等

```
<?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">
<!--配置Dao-->
    <bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl">
<!--         表示注入一个数组-->
<!--        以下的name 都是 和变量名匹配-->
        <property name="array">
            <array>
                <value>100</value>
                <value>200</value>
                <value>300</value>
            </array>
        </property>

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

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

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

```
                <entry key="city" value="kaifeng"></entry>
            </map>
        </property>

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

4. 主函数调用

```
package com.itheima;

import com.itheima.dao.BookDao;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

    public static void main(String[] args){

        System.out.println("start");

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

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

        bookDao.save();

    }
}
```

数据源管理对象

idea查找当前方法的实现 快捷键 alt + 7

1. 创建一个接口类

```java
package com.itheima.dao;

//定义一个接口 BookDao

public interface BookDao {

//    在接口中定义一个函数 这个函数没有实现
    public void save();

}
```

2. 继承接口并且实现类

```java
package com.itheima.dao.impl;

import com.itheima.dao.BookDao;

import java.util.*;

public class BookDaoImpl implements BookDao  {

    public void save(){

        System.out.println("BookDao is running..." );

    }
}
```

3. 在xml文件中配置相应的坐标

```xml
<!--        导入 druid 坐标-->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.16</version>
        </dependency>
        <!--导入c3p0 坐标-->
        <dependency>
            <groupId>c3p0</groupId>
            <artifactId>c3p0</artifactId>
            <version>0.9.1.2</version>
        </dependency>

<!--    导入 mysql 坐标-->
                <dependency>
                    <groupId>mysql</groupId>
                    <artifactId>mysql-connector-java</artifactId>
                    <version>5.1.16</version>
                </dependency>
```

4. 在bean中进行管理

```xml
<!--   管理DruidDataSource 对象-->
    <bean class="com.alibaba.druid.pool.DruidDataSource">
```

```
        <property name="driverClassName"
value="com.alibaba.mysql.jdbc.Driver"></property>
        <property name="url" value="jdbc:mysql://localhost:3306/spring_db">
</property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>


    </bean>

    <!--    管理Druid的连接池-->
    <bean id="dataSource"
 class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="com.mysql.jdbc.Driver">
</property>
        <property name="jdbcUrl"
value="jdbc:mysql://localhost:3306/spring_db"></property>
        <property name="password" value="root"></property>
        <property name="user" value = "root"></property>
```

5. 在main函数中进行测试

```
package com.itheima;

import com.itheima.dao.BookDao;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import javax.sql.DataSource;

public class App {

    public static void main(String[] args){

        System.out.println("start");

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

        DataSource dataSource = (DataSource) ctx.getBean("dataSource");

        System.out.println(dataSource);
    }
}
```

加载properties文件

1. 在properties中创建一些数据变量

```
driverClass = "com.mysql.jdbc.Driver"
jdbcUrl = "jdbc:mysql://localhost:3306/spring_db"
password = "root"
user = "root"
```

2. 在xml文件中那个配置 开辟一个新的空间

```
 xmlns:context="http://www.springframework.org/schema/context"
```

3. 加入相应的https

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

4. 指定相应的内容 classpath*表示的所有路径 *.properties表示所有的properties的文件

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

5. 向bookDao中传递参数 使用 ${}

```
    <bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl">
        <property name="name" value="${driverClass}"></property>
    </bean>
```

容器

加载类型

1. 文件类型加载

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

2. 加载盘符类型

```
 ApplicationContext ctx = new FileSystemXmlApplicationContext("E:\\大学两年的比赛和项目\\Spring\\spring09\\src\\main\\resources\\applicationContext.xml");
```

强行转换类型

1. 实现强转

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

2. 实现类转换

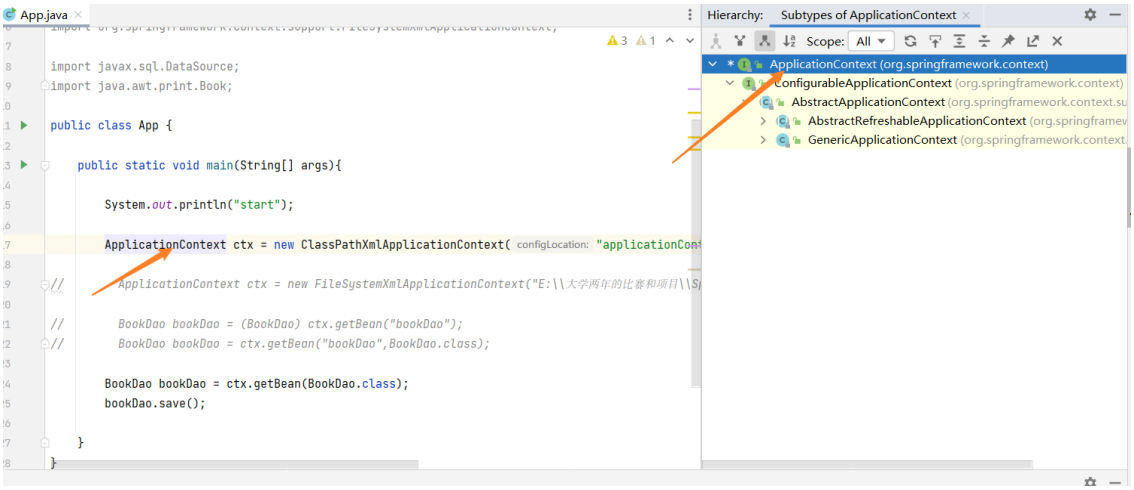
```
BookDao bookDao = ctx.getBean("bookDao",BookDao.class);
```

3. 指定类中 这种写法要保证文件中只有这么一个类

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

查看类接口

1. 使用快捷键 ctrl+H



2. 创建一个BeanFactory对象

```
package com.itheima;

import com.itheima.dao.BookDao;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;


public class AppFactory {

    public static void main(String[] args) {

        Resource resources = new
ClassPathResource("applicationContext.xml");

        BeanFactory bf = new XmlBeanFactory(resources);

        BookDao bookDao = (BookDao) bf.getBean("bookDao");

        bookDao.save();

    }
}
```

3. BeanFactory 和 ApplicationContext 区别

   在有构造方法的时候 BeanFactory 不会触发构造方法 但是 ApplicationContext 会触发构造方法

4. 解决两者这件的区别 在 bean的坐标中加上一个延迟加载就行了

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

bean的总结

```
<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
    />
```

注解开发定义bean

1. 注解开发提供了一个Component 注解取代 Component等价于下面代码

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

具体使用

```
    package com.itheima.dao.impl;

    import com.itheima.dao.BookDao;
    import org.springframework.stereotype.Component;
//  进行注解 （） 中表示名称 参数可以省去
    @Component("bookDao")

    public class BookDaoImpl implements BookDao  {
```

```
        public void save(){
            System.out.println("BookDao is running...");
        }
    }
```

2. 让 Spring 感知到这个 component 在xml中进行配置 让Spring感知到

   加上component-scan是可以扫描到Bean的

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

3. 在使用 Component 进行注解的时候
   - 进行命名

```
@Component("bookDao")
// 调用的时候 使用名称调用
BookDao bookDao = (BookDao) ctx.getBean("bookDao");

System.out.println(bookDao);
```

   - 不进行命名

```
@Component

// 不使用名称调用 使用默认参数的时候 就只能调用类的形式进行调用

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

System.out.println(bookService);
```

4. 使用衍生注解

- 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

```
@Repository("bookDao")
public class BookDaoImpl implements BookDao {
}
@Service
public class BookServiceImpl implements BookService {
}
```

纯注解开发bean

1. 使用SpringConfig直接使用类 将上面代码替换成下面代码

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       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

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

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

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

   等价于

```
@Configuration
```

2. 使用@ComponentScan 配制 Context文件

```
@ComponentScan("com.itheima") 代替 <context:component-scan base-
package="com.itheima"></context:component-scan>
@ComponentScan({"com.itheima.dao","com.itheima.service"})
```

3. 使用类的时侯 替换

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

```
 ApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);
```

bean的生命周期管理

控制类的单例和多例

```
    package com.itheima.dao.impl;

    import com.itheima.dao.BookDao;
    import org.springframework.context.annotation.Scope;
    import org.springframework.stereotype.Component;

    @Component("bookDao")
//  控制单例使用
//    @Scope("singleton") // 单例
//    @Scope("prototype") // 多例
    public class BookDaoImpl implements BookDao  {

        public void save(){

            System.out.println("BookDao is running...");

        }
    }
```

```
"C:\Program Files\Java\jdk1.8.0_151\bin\java.exe" ...
com.itheima.dao.impl.BookDaoImpl@1a052a00
com.itheima.dao.impl.BookDaoImpl@4d826d77
com.itheima.service.impl.BookServiceImpl@7f77e91b
```

生命周期函数 init destory

```
//        构造方法前
        @PostConstruct
        public void init(){
            System.out.println("init...");
        }
//        销毁方法后
        @PreDestroy

        public void destory(){
            System.out.println("destory...");
        }
    }
```

主函数调用

```
        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);

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

        System.out.println(bookDao);

        BookDao bookDao1 = (BookDao) ctx.getBean("bookDao");
```

```
        System.out.println(bookDao1);

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

        System.out.println(bookService);
//        关闭钩子
        ctx.close();
```

注解开发依赖注入

Autowired和Qualifier 配置注解

引入Autowired之后 setter 方法可以不写

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

```
//    使用类型装配
    @Autowired
//    指定配置 BookDao这个属性
    @Qualifier("bookDao")
    private BookDao bookDao;
//  使用了Autowired 之后 setter 方法可以不用
    public void setBookDao(BookDao bookDao) {
        this.bookDao = bookDao;
    }
```

value 传递参数

```
@Value("${name}")
private String name;

public void save(){

    System.out.println("BookDao is running...");
    System.out.println("Hello " + name);

    }
}
```

Springconfig配置properties文件

```
@PropertySource({"jdbc.properties"})
```

第三方bean管理 依赖注入

管理一个第三方的bean

在 SpringConfig中进行配置第三方的 Bean

```
//    1定义一个要管理的对象
//    2添加@Bean,表示当前返回值是一个bean
    @Bean("dataSource")
    public DataSource dataSource(){
        DruidDataSource ds = new DruidDataSource();

        ds.setDriverClassName("com.mysql.jdbc.Driver");

        ds.setUrl("jdbc:mysql://localhost:3306/spring_db");

        ds.setPassword("root");

        ds.setUsername("root");
        return ds;
    }
```

但是缺点是 数据都在这个文件中进行配置 会造成数据混乱现象

基于对应文件开发管理第三方文件

方式一 建立JdbcConfig文件

```
package com.itheima.config;

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

import javax.sql.DataSource;

@Configuration
public class JdbcConfig {


//    1定义一个要管理的对象
//    2添加@Bean,表示当前返回值是一个bean
    @Bean("dataSource")
    public DataSource dataSource(){
        DruidDataSource ds = new DruidDataSource();

        ds.setDriverClassName("com.mysql.jdbc.Driver");

        ds.setUrl("jdbc:mysql://localhost:3306/spring_db");

        ds.setPassword("root");

        ds.setUsername("root");
        return ds;
    }
}
```

```java
package com.itheima.config;

import com.alibaba.druid.pool.DruidDataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;

import javax.sql.DataSource;

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

}
```

缺点是在SpringConfig中无法观察配置文件的内容

导入式import

```java
//@ComponentScan("com.itheima.config")
@Import(JdbcConfig.class)
```

简单类型进一步封装

```java
    @Value("${com.mysql.jdbc.Driver}")
    private String dirver;
    @Value("${jdbc:mysql://localhost:3306/spring_db}")
    private String url;
    @Value("${root}")
    private String name;
    @Value("${root}")
    private String password;
```

引用类型

直接给一个形参 通过Bean 进行扫描得到的就行

```java
public DataSource dataSource(BookDao bookDao){

    System.out.println(bookDao);
```

1. 第三方bean管理
    - @Bean
2. 第三方bean依赖注入
    - 引用类型：方法形参
    - 简单类型：成员变量

Spring整合Mybatis

加入依赖

```
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.2.10.RELEASE</version>
        </dependency>
                <!--mybatis和spring的整合坐标-->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>1.3.0</version>
        </dependency>
```

1 创建JdbcConfig

1 JdbcConfig中主要是给 数据库中的变量赋值

```
    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
    private String password;
```

2 使用注解生成DataSource

```
    @Bean
    public DataSource dataSource(){

        DruidDataSource ds = new DruidDataSource();
        ds.setDriverClassName(driver);
        ds.setUrl(url);
        ds.setUsername(username);
        ds.setPassword(password);
        return ds;
    }
```

3 将这个文件导入到SpringConfig中

```
// 为了在 SrpingConfig中使用JdbcConfig
@Import({JdbcConfig.class})
// 使用Jdbc.properties中的数据
@PropertySource("classpath:jdbc.properties")
```

2 创建MybatisConfig文件

1 创建SqlSessionFactoryBean对象

```
    @Bean
    public SqlSessionFactoryBean sqlSessionFactoryBean(DataSource dataSource){

        SqlSessionFactoryBean ssfb = new SqlSessionFactoryBean();

        ssfb.setTypeAliasesPackage("com.itheima.domain");

        ssfb.setDataSource(dataSource);

        return ssfb;

    }
```

1 代替变量名 ssfb.setTypeAliasesPackage("com.itheima.domain");

```
<typeAliases>
        <package name="com.itheima.domain"/>
</typeAliases>
```

2 代替dataSource对象 ssfb.setDataSource(dataSource);

```
<dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"></property>
                <property name="url" value="${jdbc.url}"></property>
                <property name="username" value="${jdbc.username}"></property>
                <property name="password" value="${jdbc.password}"></property>
</dataSource>
```

2 MapperScannerConfigurer对象替代mappers

```
<mappers>
    <package name="com.itheima.dao"></package>
</mappers>
```

```
    @Bean
    public MapperScannerConfigurer mapperScannerConfigurer(){

        MapperScannerConfigurer mapperScannerConfigurer = new
MapperScannerConfigurer();

        mapperScannerConfigurer.setBasePackage("com.itheima.dao");

        return mapperScannerConfigurer;

    }
```

3 将MybatisConfig导入到SpringConfig中

```
@Import({MybatisConfig.class})
```

3 SpringConfig文件

```
//使用 @Configuration 代替<beans></beans>
@Configuration
// 使用@ComponentScan 进行扫描文件
@ComponentScan("com.itheima")
// 使用Jdbc.properties中的数据
@PropertySource("classpath:jdbc.properties")
// 为了在 SrpingConfig中使用JdbcConfig
@Import({JdbcConfig.class, MybatisConfig.class})
public class SpringConfig {
}
```

4 创建dao文件创建AccountDao的接口

```
public interface AccountDao {
    @Select("select * from tb_user where id = #{id}")
    Account findById(int id);
}
```

5 创建domain文件创建Account类

```
    package com.itheima.domain;
    /**
    *@ClassName Account
    *@Description  TODO
    *@Author lenovo
    *@Date 2022/8/16 11:12
    *@Version 1.0
    */
```

```java
public class Account {

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

    public Integer getId() {
        return id;
    }

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

    public String getUsename() {
        return username;
    }

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

    public String getPassword() {
        return password;
    }

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

    public String getGender() {
        return gender;
    }

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

    public String getAddr() {
        return addr;
    }

    public void setAddr(String addr) {
        this.addr = addr;
    }

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

```
        }
    }
```

6 编写测试类

```
 @Test

    public void Test1() throws Exception {

        ApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);

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

        AccountDao accountDao = ctx.getBean(AccountDao.class);

        Account ac = accountDao.findById(2);

        System.out.println(ac);
    }
```

Junit整合

设置专用的类运行器

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = SpringConfig.class)
```

```
@Autowired
private AccountService accountService
```

等价于

```
    @Autowired
    @Qualifier("accountService")
    private AccountService accountService;

    public void setAccountService(AccountService accountService) {
        this.accountService = accountService;
    }
```

测试输出

```
@Test
public void testFindById(){

    System.out.println(accountService.findById(2));

}
```

AOP

AOP 面向切面编程 OOP面向对象编程

作用：在原来基础上为其增加功能

这个方法可以输出当前时间

```
    public void save(){
        // 将时间输出单独抽出去
        System.out.println(System.currentTimeMillis());
        System.out.println("book dao save...");

    }
```

这个方法不能输出当前时间

```
public void update(){

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

使用AOP方法实现并且不改边 update方法但是可是实现输出时间

```
public void method(){
     // 将时间输出单独抽出去
     System.out.println(System.currentTimeMillis());
}
```

将上面的方法单独抽出来

连接点：JoinPoint：方法的执行

切入点：PointCut：匹配连接点的式子

切面：Aspect：描述通知和切入点的对应关系

AOP入门案例

1 导入坐标（pom.xml）

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

2 制作连接点方法 (Dao接口和实现类)

```
@Repository
public class BookDaoImpl implements BookDao {

    public void save(){
        // 将时间输出单独抽出去
        System.out.println(System.currentTimeMillis());
        System.out.println("book dao save...");

    }


    public void update(){

        System.out.println("book dao update...");
    }

}
```

```
public interface BookDao {

    public void save();

    public void update();

}
```

3 制作共性功能 （通知类与通知）

创建文件aop

在文件中创建类MyAdvice类实现共性方法

定义好通知

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

4 定义切入点

定义一个空壳函数 public void 函数名(){}

在函数上方写入

@PointCut("execution(void (类型) com.itheima.dao.BookDao.update())")

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

5 绑定切入点与通知关系 （切面）

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

6 @Aspect说明AOP扫描切入点和通知 @Component生成Bean

```
@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 在SpringConfig文件中加入@EnableAspectJAutoProxy

@EnableAspectJAutoProxy 连接 @Aspect

```
//使用 @Configuration 代替<beans></beans>
@Configuration
// 使用@ComponentScan 进行扫描文件
@ComponentScan("com.itheima")
@EnableAspectJAutoProxy
public class SpringConfig {
}
```

AOP切入点表达式

切入点：要增强的方法

切入点表达式：要进行增强的方法的描述方式

描述方法形式有多种：1 描述接口中的方法 2 描述实现类的方法

1 语法格式

动作关键字（访问修饰符 返回值 包名.类 / 接口名.方法名（参数））

动作关键字：通常是execution

访问修饰符 public private

2 通配符

AOP通知类型

1 前置通知

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

2 后置通知

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

3 环绕通知

```
@Around("pt1()")
    public Object around_select(ProceedingJoinPoint pjp) throws Throwable {

        System.out.println("around before advice...");
        //表示对原始操作的调用
        Integer ret = (Integer) pjp.proceed();

        System.out.println("around after advice...");

        return ret + 566;
    }
```

4 afterReturning

```
// 有类型返回就会触发
   public void afterReturning() {
       System.out.println("afterReturning advice...");
   }
```

5 afterThrowing

```
    // 这个产生异常就会触发
    public void afterThrowing() {
        System.out.println("afterThrowing advice...");
    }
```

AOP完成执行案例

```
package com.itheima.aop;

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

/**
 * @ClassName ProjectAdvice
 * @Description TODO
 * @Author lenovo
 * @Date 2022/8/21 19:52
 * @Version 1.0
 */
@Component
@Aspect
public class ProjectAdvice {

    @Pointcut("execution(* com.itheima.dao.AccountDao.findById(..))")
    private void servicePt(){}

    @Around("ProjectAdvice.servicePt()")
    public void runSpeed(ProceedingJoinPoint pjp) throws Throwable {
        // 获取对象的方法
        Signature signature = pjp.getSignature();
        // 获得执行对象的包名
        String declaringTypeName = signature.getDeclaringTypeName();
        // 获得执行对象的方法名
        String name = signature.getName();

        System.out.println(declaringTypeName);

        System.out.println(name);

        long stat = System.currentTimeMillis();

        for(int i = 0; i < 10000; i++){

            pjp.proceed();

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

```
        System.out.println("万次连接操作时间:" + (end - stat) + "ms");
    }


}
```

AOP通知获取数据

获取参数 before after 使用JoinPoint

```
    @Before("ProjectAdvice.servicePt()")
     public void before(JoinPoint joinPoint) {

        Object[] args = joinPoint.getArgs();

        System.out.println(Arrays.toString(args));

        System.out.println("before...");
     }

     @After("ProjectAdvice.servicePt()")
     public void after(JoinPoint joinPoint) {

         Object[] args = joinPoint.getArgs();

         System.out.println(Arrays.toString(args));

         System.out.println("after...");
     }
```

Around 使用ProceedingJoinPoint

```
    @Around("ProjectAdvice.servicePt()")
     public Object around(ProceedingJoinPoint pjp) throws Throwable {

         Object[] args = pjp.getArgs();

         for(int i = 0 ; i < args.length;i++){
             System.out.println(args[i]);
         }

         System.out.println("before around...");

         System.out.println("after around...");

         // 修改传过来的参数

         args[0] = 666;

         args[1] = "zhangsan" ;
         // 将修改获得args 传回原函数 返回回去
         Object proceed = pjp.proceed(args);

         return proceed;
```

```
    }
```

AfterReturning

```
@AfterReturning(value = "ProjectAdvice.servicePt()", returning = "ret")
    public void afterReturning(Object ret) {
        System.out.println("afterReturning..." + ret);
    }
```

SpingMVC

1 SpringMVC入门案例

1 导入坐标

```
  <dependencies>
  <!--  导入坐标 javax 和 servlet-->
      <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
        <scope>provided</scope>
      </dependency>

  <!--  springwebmvc-->

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



  </dependencies>
<!--  配置 tomcat 插件-->

  <build>
    <plugins>

      <!--tomcat插件-->
      <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.2</version>
      </plugin>
    </plugins>
  </build>
```

2 创建Servlet启动类 加载spring文件

```
package com.itheima.config;

import org.springframework.web.context.WebApplicationContext;
import
org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import
org.springframework.web.servlet.support.AbstractDispatcherServletInitializer;

/**
 * @ClassName ServletContainerinitConfig
 * @Description TODO
 * @Author lenovo
 * @Date 2022/8/30 11:44
 * @Version 1.0
 */

// 定义一个Servlet 启动类 加载里面的spring 文件
public class ServletContainerinitConfig extends
AbstractDispatcherServletInitializer {

    // 加载springmvc容器配置
    @Override
    protected WebApplicationContext createServletApplicationContext() {

        AnnotationConfigWebApplicationContext ctx = new
AnnotationConfigWebApplicationContext();

        ctx.register(SpringMvcConfig.class);

        return ctx;
    }

    //设置 那些请求归属springmvc处理
    @Override
    protected String[] getServletMappings() {
        // 设置所有请求归spring 请求
        return new String[]{"/"};
    }

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

方法一: 加载springmvc容器配置

创建一个web容器

```
 // 加载springmvc容器配置
    @Override
    protected WebApplicationContext createServletApplicationContext() {

        AnnotationConfigWebApplicationContext ctx = new
AnnotationConfigWebApplicationContext();

        ctx.register(SpringMvcConfig.class);

        return ctx;
    }
```

方法二: 设置请求归属

将所有的方法设置为SpringMVC形式

```
  //设置 那些请求归属springmvc处理
    @Override
    protected String[] getServletMappings() {
        // 设置所有请求归spring 请求
        return new String[]{"/"};
    }
```

方法三: 加载spring形式

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

3 定义一个SpringMVC注解

扫描指定文件 将文件加载到 web容器当中去

```
// 进行扫描
@Configuration
// 扫描 controller
@ComponentScan("com.itheima.controller")

public class SpringMvcConfig {
}
```

4 定义一个controller类

```
// 定义Controller 定义bean 这个是springmvc 特定的
@Controller
public class UserController {

    // 定义当前的访问路径
    @RequestMapping("/save")

    //设置当前操作的返回类型
    @ResponseBody

    // 定义一个处理请求的 方法
    public String save() {

        System.out.println(" user save ");

        return "'{'moudle' : 'springmvc'}'";

    }
}
```

@Controller 是定义bean是使用的 是 springmvc中特定的

@RequestMapping("/save") 是同@RequestMapping 指向访问路径 以为getServletMappings()全部拦截为springmvc中

@ResponseBody 表示返回类型 是json形式

@ResponseBody

1、概念

注解 @ResponseBody，使用在控制层（controller）的方法上。

2、作用

将方法的返回值，以特定的格式写入到response的body区域，进而将数据返回给客户端。

当方法上面没有写ResponseBody,底层会将方法的返回值封装为ModelAndView对象。

如果返回值是字符串，那么直接将字符串写到客户端；如果是一个对象，会将对象转化为json串，然后写到客户端。

2 排除过滤器

```
// 排除过滤器 排除所有的 controller 的bean
@ComponentScan(value = "com.itheima",
        excludeFilters = @ComponentScan.Filter(
            // 类型方式 使用注解的方式
                type = FilterType.ANNOTATION,
            // 所有的代码类
                classes = Controller.class
        )
)
```

```
public class App {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);

        System.out.println(ctx.getBean(UserController.class));
    }
}
```

3 简易开发

```
public class ServletContainerinitConfig extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        //SpringMVC 开发
        return new Class[]{SpringMvcConfig.class};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {

        //Spring 开发


        return new Class[]{SpringConfig.class};
    }

    @Override
    protected String[] getServletMappings() {

        // 拦截所有的方法
        return new String[]{"/"};
    }
}
```

4 定义整个目录的前缀

```
@RequestMapping("/User")
```

5 传参数

1 简单参数

```
public String save(String name) {

    System.out.println("user save ");

    System.out.println(name);

    return "'{'moudle' : 'springmvc'}'";

}
```

2 post传参乱码处理

```
@Override
protected Filter[] getServletFilters(){

    CharacterEncodingFilter filter = new CharacterEncodingFilter();

    // 设置编码方式
    filter.setEncoding("UTF-8");

    return new Filter[]{filter};
}
```

3 传数组

```
 public String save1(String [] likes) {

        System.out.println("user save ");

        System.out.println(Arrays.toString(likes));

        return "'{'moudle' : 'springmvc'}'";

    }
```

4 传集合

```
// 定义当前的访问路径
@RequestMapping("/save2")

//设置当前操作的返回类型
@ResponseBody
```

```
// 定义一个处理请求的 方法
// 传集合
// 加上注解 @RequestParam 说明 传递形式是 集合形式
public String save2(@RequestParam List<String> list) {
    System.out.println(list);
    System.out.println("user save ");

    return "'{'moudle' : 'springmvc'}'";

}
```

5 传对象参数

创建user类 实现向user类中传入参数

```
    public String save(User user) {

        System.out.println("Book save");

        System.out.println(user.toString());

        System.out.println(user.getName());

        System.out.println(user.getAge());

        return "'{'moudle':'SpringMVC'}'";

    }
```

6 传对象嵌套参数

创建Address

```
package com.itheima.domain;

/**
 * @ClassName Address
 * @Description TODO
 * @Author lenovo
 * @Date 2022/8/30 21:28
 * @Version 1.0
 */

public class Address {

    String name;
    int age;

    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Address{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

创建User类 加入address

```
package com.itheima.domain;

/**
 * @ClassName User
 * @Description TODO
 * @Author lenovo
 * @Date 2022/8/30 21:03
 * @Version 1.0
 */

public class User {

    String name ;
    int age;

    // 嵌套使用类型
    private Address address;


    @Override
    public String toString() {
        return "User{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", address=" + address +
                '}';
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
```

```
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

实现方法

```
    // 定义当前的访问路径
    @RequestMapping("/save3")

    //设置当前操作的返回类型
    @ResponseBody

    // 定义一个处理请求的 方法
    // 传集合
    // 加上注解 @RequestParam 说明 传递形式是 集合形式
    public String save3(User user) {
        System.out.println("user save ");
        System.out.println(user);
        return "'{'moudle' : 'springmvc'}'";

    }
```

6 传json文件

1 导入json坐标

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

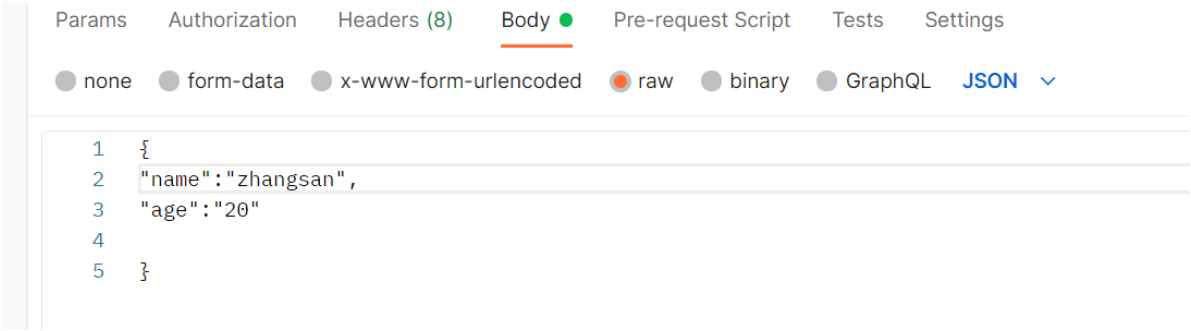
2 在config中加入识别json文件

```
// 开启注解支持JSON格式传输

@EnableWebMvc
```

3 编写传输数据

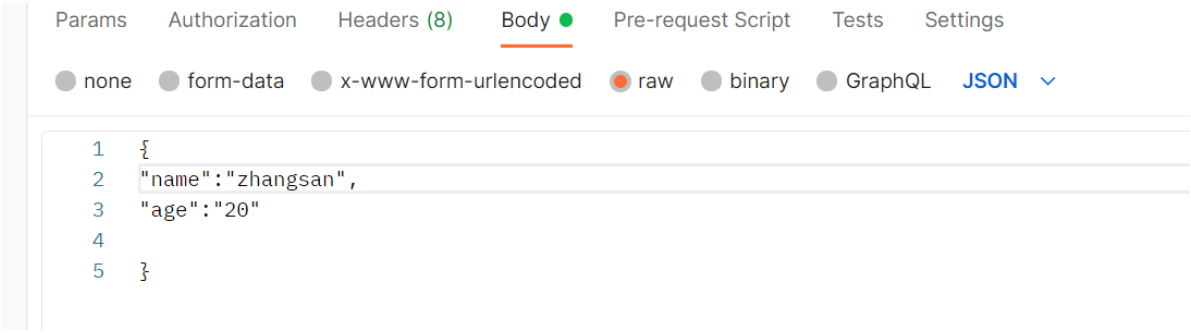
加入@RequestBody 识别传入数据类型是 json形式

```
//    定义一个以json格式传输数据

    @RequestMapping("/listParamForJson")
    @ResponseBody
    // 传入参数是json形式 加入注解形式说明
        public String listParamForJson(@RequestBody List<String> likes){

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

        System.out.println("likes ==>" + likes);

        return  "{'module':'list common for json param'}";

    }
```



加入对象形式传送数据

```
    //    定义一个以json格式传输数据

    @RequestMapping("/user/listParamForJson")
    @ResponseBody
    // 传入参数是json形式 加入注解形式说明
    public String UserlistParamForJson(@RequestBody User user){

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

        System.out.println(user.toString());

        return  "{'module':'list common for json param'}";

    }
```



传送数据为对象数组形式

```
// 传送数组形式传递

    @RequestMapping("/user/ListParamForJson")
    @ResponseBody
    // 传入参数是json形式 加入注解形式说明
    public String UserlistParamForJson(@RequestBody List<User> user){

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

        System.out.println("user == >" + user);

        return  "{'module':'list common for json param'}";

    }
```

```
[{
    "name":"zhangsan",
    "age":"200",
    "address":
    {
        "name":"jack",
        "age":"100"
    }
},
{
    "name":"lisi",
    "age":"20",
    "address":
    {
        "name":"jack_json",
        "age":"10"
    }
}]
```

@EnableWebMvc 作用是 开启SpringMVC多项辅助功能

@RequestBody与@RequestParam区别

@RequestBody 更多的是用来接受json数据 或者 application

@RequestParam区别 更多是用来接受表单传参 【application / x-www-form-urlencoded】

7 时间形式转换

1 方式1 2022/08/31

```
//      传递一个为日期的 形参
    @RequestMapping("/Data")
    @ResponseBody

    public String data(Date data){

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

        System.out.println(data);

        return "{'module':'data param'}";
    }
```

2 方式2 2022-08-31

使用 @DateTimeFormat 定义格式

定义成以 - 连接的形式 @DateTimeFormat(pattern = "yyyy-MM-dd") Date data1,

```
//      传递一个为日期的 形参
    @RequestMapping("/Data")
    @ResponseBody

    public String data(Date data,
                       @DateTimeFormat(pattern = "yyyy-MM-dd") Date data1,
                         @DateTimeFormat(pattern = "yyyy-MM-dd HH:MM:SS") Date
data2){

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

        System.out.println(data1);

        System.out.println(data);

        System.out.println(data2);

        return "{'module':'data param'}";
    }
```

注意点：

开启@**EnableWebMvc**

说明： 准换原理 是 字符之间格式的互相转换

响应

1 跳转页面

返回 page.jsp页面

```
@Controller
public class Page {

    @RequestMapping("/page1")
    public String page1(){

        System.out.println("page is running...");

        return "page.jsp";
    }
}
```

2 返回文本形式

```
package com.itheima.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

/**
 * @ClassName Text
 * @Description TODO
 * @Author lenovo
 * @Date 2022/8/31 17:43
 * @Version 1.0
 */

@Controller

public class Text {

    @RequestMapping("/Text")
    @ResponseBody

    public String Text(){

        System.out.println("Text is runing...");

        return "response.text";
    }

}
```

3 响应Json数据 和 集合

```
package com.itheima.controller;

import com.itheima.domain.User;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

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

/**
 * @ClassName Json
 * @Description TODO
 * @Author lenovo
 * @Date 2022/8/31 17:47
 * @Version 1.0
 */
@Controller
@RequestMapping("/Json")
public class Json {

    @RequestMapping("/json")

    @ResponseBody

    public User json(){

        System.out.println("json is running...");

        User user = new User();

        user.setName("itheima");

        user.setAge(20);

        System.out.println(
                user.toString()
        );

        return user;

    }



//    响应Pojo集合

    @RequestMapping("/json_collection")

    @ResponseBody

    public List<User> user(){

        System.out.println("user is running...");
```

```
        List<User> user = new ArrayList<User>();

        User user1  = new User();

        user1.setName("传智播客");
        user1.setAge(200);

        User user2 = new User();
        user2.setName("黑马程序员");
        user2.setAge(10);

        user.add(user1);
        user.add(user2);

        return user;

    }

}
```

Rest风格

指定查询功能

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

注解：

@RequestParam @PathVariable.@RequestParam

@RequestParam 请求参数是要以json形式发送

@PathVariable 传一个形参的时候，使用

@RequestParam 使用的时候是 一个或两个参数， 多用来 传表单数据 url地址等

使用rest 风格编写代码的时候　RequestMapping 变化

@RequestMapping 使用value 指定路径 指定使用方法 method

```
@RequestMapping(value = "/rest_save",method = RequestMethod.POST)

@ResponseBody
```

当时用参数传递的时候 加上 @PathVariable注解

```
@RequestMapping(value = "/delete/{id}",method = RequestMethod.DELETE)

@ResponseBody
```

```
public String delete(@PathVariable int id)
```

```
    /**
     *  使用rest风格 删除id 使用method方法 DELETE
     *  @PathVariable 定义地址值
     *
     * 删除操作
     * @param id
     * @return
     */
    @RequestMapping(value = "/delete/{id}",method = RequestMethod.DELETE)

    @ResponseBody

        public String delete(@PathVariable int id){

        System.out.println("delete");

        System.out.println(id);

        return "{'module':'user save'}";
    }

    /**
     * 跟新操作
     * @param user
     * @return
     */
    @RequestMapping(value = "/users",method = RequestMethod.PUT)
    @ResponseBody

    // 以json形式发送
    public String update(@RequestBody User user){
        System.out.println("user update");

        return "{'module':'user update'}";
    }

    /**
     * 查询操作
     * @param id
     * @return
     */

    @RequestMapping(value = "/users/{id}",method = RequestMethod.GET)
    @ResponseBody

    public String get(@PathVariable int id){
```

```
        System.out.println("user get...");

        System.out.println(id);

        return "{'module':'user get'}";
    }

    /**
     * 无参传递
     * 查询操作
     * @param
     * @return
     */

    @RequestMapping(value = "/users",method = RequestMethod.GET)
    @ResponseBody

    public String get(){

        System.out.println("user getAll...");

        return "{'module':'user get'}";
    }
```

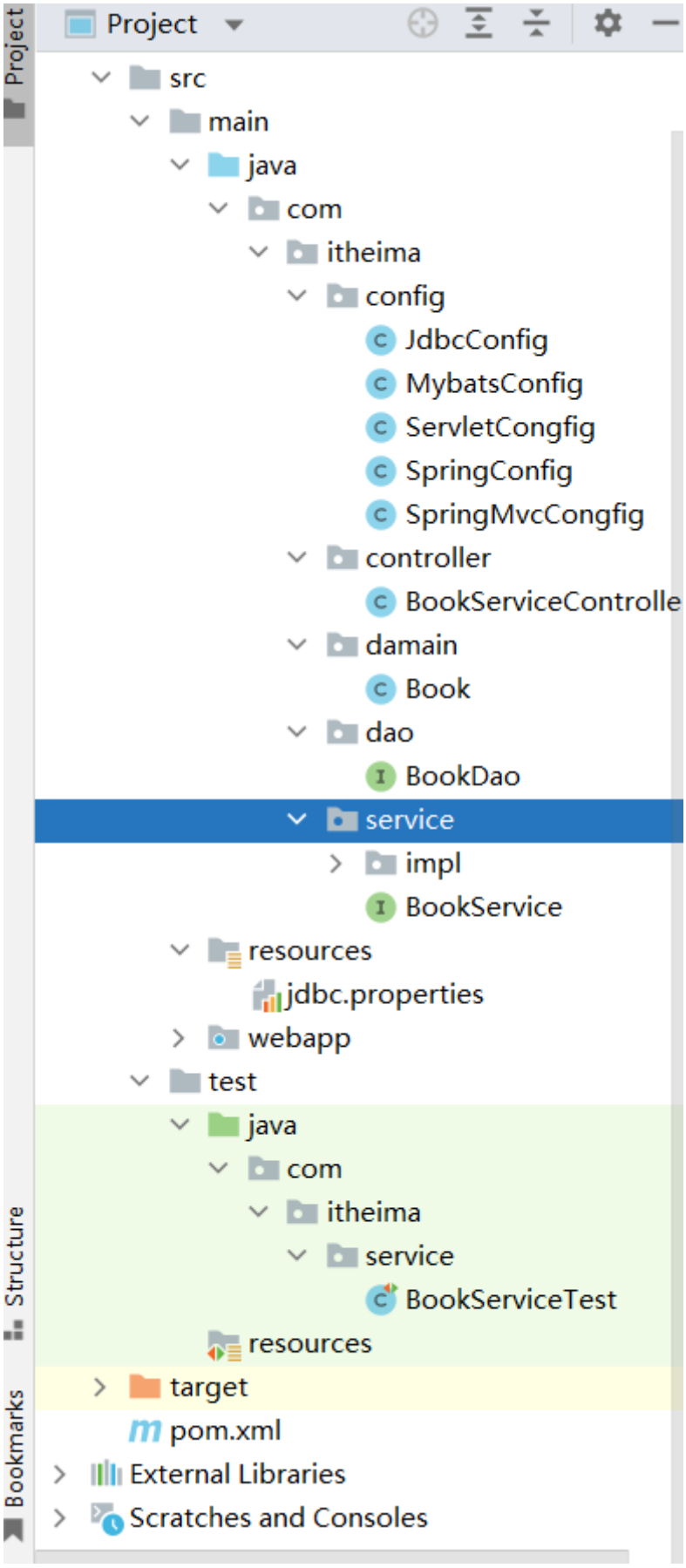
简易开发

```
package com.itheima.controller;

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

/**
 * @ClassName SP_BookController
 * @Description TODO
 * @Author lenovo
 * @Date 2022/9/1 9:47
 * @Version 1.0
 */

// @Controller
// @ResponseBody

//    使用 RestController 把 @Controller 和 @ResponseBody 合并
@RestController
@RequestMapping("/SP_books")
public class SP_BookController {
    /**
     *  add
     * @param book
     * @return
     */
    // @RequestMapping(method = RequestMethod.POST)
    @PostMapping
    public String add(@RequestBody Book book){
```

```
        System.out.println("book add..." + book.toString()
        );

        return "'{moudle}:add'";
    }

    /**
     * delete
     * @param id
     * @return
     */
    // @RequestMapping(value = "/{id}",method = RequestMethod.DELETE)
    @DeleteMapping("/{id}")
    public String delete(@PathVariable int id){

        System.out.println("book delete..." + id);

        return "'{moudle}:delete'";
    }

    /**
     * update
     * @param book
     * @return
     */
    // @RequestMapping(method = RequestMethod.PUT)
    @PutMapping
    public String update(@RequestBody Book book){

        System.out.println("book update..." + book.toString());

        return "'{moudle}:update'";
    }


    /**
     * get
     * @param id
     * @return
     */
    // @RequestMapping(value = "/{id}",method = RequestMethod.GET)
    @GetMapping("/{id}")
    public String get(@PathVariable int id){

        System.out.println("book get..." + id);

        return "'{moudle}:get'";
    }

}
```

SSM整合

1 文件层次目录


Project
src
main
java
com
itheima
config
JdbcConfig
MybatsConfig
ServletCongfig
SpringConfig
SpringMvcCongfig
controller
BookServiceControlle
damain
Book
dao
BookDao
service
impl
BookService
resources
jdbc.properties
webapp
test
java
com
itheima
service
BookServiceTest
resources
target
pom.xml
External Libraries
Scratches and Consoles
Project
Structure
Bookmarks

2 导入坐标

pom文件

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

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

  <groupId>org.example</groupId>
  <artifactId>ssm_01</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>


  <dependencies>

    <!--导入 druid 坐标-->
    <dependency>
      <groupId>com.alibaba</groupId>
      <artifactId>druid</artifactId>
      <version>1.1.16</version>
    </dependency>

    <!--导入 mybatis 坐标-->
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis</artifactId>
      <version>3.5.6</version>
    </dependency>

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

    <!-- 导入 mysql 连接 java坐标-->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.47</version>
    </dependency>

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

    <!--导入 spring-test 坐标-->
    <dependency>
```

```
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
      <version>5.2.1.RELEASE</version>
      <scope>test</scope>
    </dependency>

    <!-- 导入junit坐标-->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>

    <!--导入 spring-webmvc坐标-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>5.2.10.RELEASE</version>
    </dependency>

    <!--  导入坐标 javax 和 servlet-->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.1.0</version>
      <scope>provided</scope>
    </dependency>

    <!--导入json坐标-->
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.9.0</version>
    </dependency>

  </dependencies>


  <build>
    <plugins>
      <!--tomcat插件-->
      <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.2</version>
      </plugin>
    </plugins>
  </build>


</project>
```

3 resources文件中的jdbc 数据

jdbc.properties

```
jdbc.driver = com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatis?useSSL=false
jdbc.username=root
jdbc.password=admin123
```

4 创建一个config文件

1 创建一个 JdbcConfig文件

1 将数据库中的值赋值进去

```
@Value("${jdbc.driver}")
private String driver;
@Value("${jdbc.url}")
private String url;
@Value("${jdbc.username}")
private String username;
@Value("${jdbc.password}")
private String password;
```

2 造一个DataSource对象 声明称bean的形式

```
@Bean
    public DataSource dataSource(){

        DruidDataSource dataSource = new DruidDataSource();

        dataSource.setUsername(username);

        dataSource.setPassword(password);

        dataSource.setUrl(url);

        dataSource.setDriverClassName(driver);

        return dataSource;
    }
```

JdbcConfig

```
package com.itheima.config;

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

import javax.sql.DataSource;

/**
```

```
 * @ClassName JdbcConfig
 * @Description TODO
 * @Author lenovo
 * @Date 2022/9/1 11:31
 * @Version 1.0
 */

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.setUsername(username);

        dataSource.setPassword(password);

        dataSource.setUrl(url);

        dataSource.setDriverClassName(driver);

        return dataSource;
    }
}
```

2 创建MybatisConfig文件

在这个文件中创建 sqlsessionFactoryBean对象

创建 MapperScannerConfigurer 对象代替Mapper代理

1 sqlsessionFactoryBean

```
@Bean
    public SqlSessionFactoryBean sqlSessionFactory(DataSource dataSource){

        SqlSessionFactoryBean factoryBean = new SqlSessionFactoryBean();

        factoryBean.setDataSource(dataSource);

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

        return factoryBean;
    }
```

1 创建一个damain 文件 Book 文件 执行 返回形式

```
    package com.itheima.damain;

/**
 * @ClassName Bookd
 * @Description TODO
 * @Author lenovo
 * @Date 2022/9/1 21:29
 * @Version 1.0
 */

public class Book {

    private Integer id;
    private String type;
    private String name;
    private String description;

    @Override
    public String toString() {
        return "Book{" +
                "id=" + id +
                ", type='" + type + '\'' +
                ", name='" + name + '\'' +
                ", description='" + description + '\'' +
                '}';
    }

    public Integer getId() {
        return id;
    }

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

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

```
    public void setDescription(String description) {
        this.description = description;
    }
}
```

2 MapperScannerConfigurer

```
    @Bean
    public MapperScannerConfigurer mapperScannerConfigurer(){

        MapperScannerConfigurer msc = new MapperScannerConfigurer();

        msc.setBasePackage("com.itheima.dao");

        return msc;

    }
```

1 创建 dao 文件 下的 BookDao 文件执行sql语句查询的

```
package com.itheima.dao;

import com.itheima.damain.Book;
import org.apache.ibatis.annotations.Delete;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.annotations.Update;

import java.util.List;

/**
 * @ClassName
 * @Description TODO
 * @Author lenovo
 * @Date 2022/9/1 21:47
 * @Version 1.0
 */

public interface BookDao {

    @Insert("insert book values (#{id},#{type},#{name},#{description})")
    public Boolean add(Book book);

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

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

    // 查询单个
    @Select("select * from book where id = #{id}")
    public Book get(int id);
```

```
    //    查询全部的
    @Select("select * from book")
    public List<Book> getAll();

}
```

3 创建SpringConfig文件

将 JdbcConfig 文件和 MybatisConfig文件 导入 到 SpringCondig文件中去

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

4 创建一个SpringMvcConfig文件

扫描controller文件

```
package com.itheima.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Controller;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

/**
 * @ClassName SpringMvcCongfig
 * @Description TODO
 * @Author lenovo
 * @Date 2022/9/1 21:25
 * @Version 1.0
 */

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

创建一个 controller文件

实现网页查询

```
package com.itheima.controller;

import com.itheima.damain.Book;
import com.itheima.service.BookService;
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

import java.util.List;

/**
 * @ClassName BookServiceController
 * @Description TODO
 * @Author lenovo
 * @Date 2022/9/1 21:32
 * @Version 1.0
 */

@RestController
@RequestMapping("/Books")

public class BookServiceController {

    @Autowired
    private BookService bookService;

    @PostMapping
    public Boolean add(@RequestBody  Book book) {
        System.out.println("add...");
        return  bookService.add(book);

    }

    @DeleteMapping("/{id}")
    public Boolean delete(@PathVariable int id) {
        System.out.println("delete...");
        return bookService.delete(id);
    }

    @PutMapping
    public Boolean update(@RequestBody Book book) {
        System.out.println("update...");
        return bookService.update(book);
    }

    @GetMapping("/{id}")
    public Book get(@PathVariable int id) {
        System.out.println("get...");
        return bookService.get(id);
    }

    @GetMapping
    public List<Book> getAll() {
        System.out.println("getall...");
        return bookService.getAll();
    }

}
```

5 整合 Spring SpringMvc文件

```
package com.itheima.config;

import
org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServle
tInitializer;

/**
 * @ClassName ServletCongfig
 * @Description TODO
 * @Author lenovo
 * @Date 2022/9/1 21:23
 * @Version 1.0
 */

public class ServletCongfig extends
AbstractAnnotationConfigDispatcherServletInitializer {
    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{SpringConfig.class};
    }

    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{SpringMvcCongfig.class};
    }

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

6 实现 service 文件

```
package com.itheima.service.impl;

import com.itheima.damain.Book;
import com.itheima.dao.BookDao;
import com.itheima.service.BookService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

/**
 * @ClassName BookServiceImpl
 * @Description TODO
 * @Author lenovo
 * @Date 2022/9/1 21:31
 * @Version 1.0
 */

@Service
public class BookServiceImpl implements BookService {

    @Autowired
```

```
    private BookDao bookDao;

    public Boolean add(Book book) {

         bookDao.add(book);

         return true;

    }

    public Boolean delete(int id) {

        bookDao.delete(id);

        return true;

    }

    public Boolean update(Book book) {

        bookDao.update(book);

        return true;
    }

    public Book get(int id) {

        return bookDao.get(id);
    }

    public List<Book> getAll() {
       return bookDao.getAll();
    }

}
```

创建接口

```
package com.itheima.service;

import com.itheima.damain.Book;
import org.apache.ibatis.annotations.Delete;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.annotations.Update;

import java.util.List;

/**
 * @ClassName BookService
 * @Description TODO
 * @Author lenovo
 * @Date 2022/9/1 21:31
 * @Version 1.0
 */
```

```
public interface BookService {

    public Boolean add(Book book);

    public Boolean delete(int id);

    public Boolean update(Book book);

    public Book get(int id);

    public List<Book> getAll();

}
```

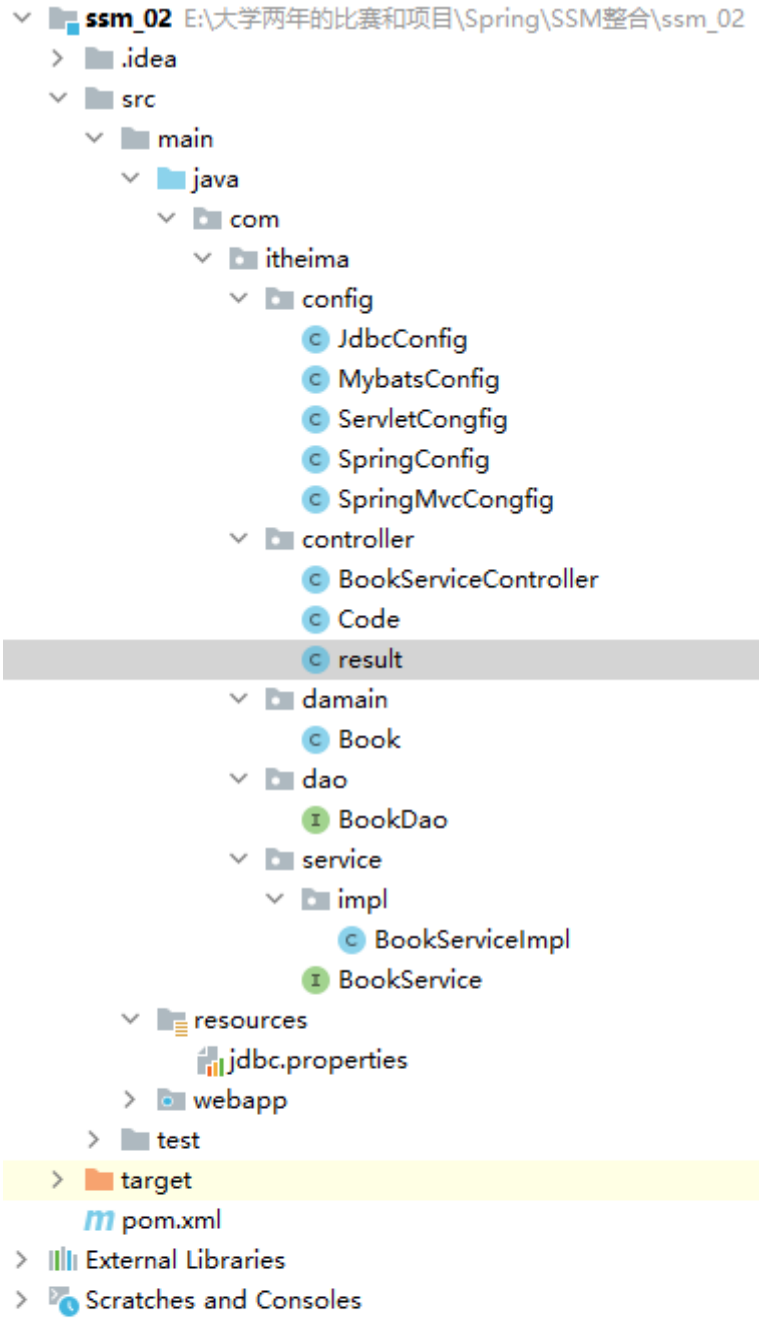
5 创建测试类

```
package com.itheima.service;

import com.itheima.config.SpringConfig;
import com.itheima.damain.Book;
import com.itheima.dao.BookDao;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import javax.swing.*;
import java.util.List;

/**
 * @ClassName BookServiceTest
 * @Description TODO
 * @Author lenovo
 * @Date 2022/9/2 11:05
 * @Version 1.0
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = SpringConfig.class)
public class BookServiceTest {


    @Autowired
    private BookService bookService;

    @Test
    public void testGetById(){

        Book book = bookService.get(1);

        System.out.println(book);
```

```
    }

    @Test
    public void testGetAll(){

        List<Book> book = bookService.getAll();

        System.out.println(book);

    }

    @Test
    public void TestGet(){

        ApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);

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

        Book book = bookDao.get(2);

        System.out.println(book);

    }

}
```

6 统一编码格式

1 文件 目录



2 创建code文件

在code文件中定义 后端返回的状态

统一编码格式

```
package com.itheima.controller;
```

```java
/**
 * @ClassName Code
 * @Description TODO
 * @Author lenovo
 * @Date 2022/9/3 9:19
 * @Version 1.0
 */

public class Code {

    // 成功标志
    public static final Integer   ADD_OK = 20011;
    public static final Integer   DELETE_OK = 20021;
    public static final Integer   UPDATE_OK = 20031;
    public static final Integer   GET_OK = 20041;

    // 失败标志
    public static final Integer   ADD_ERR = 20010;
    public static final Integer   DELETE_ERR = 20020;
    public static final Integer   UPDATE_ERR = 20030;
    public static final Integer   GET_ERR = 20040;

}
```

3 创建result文件

result 文件是 将 后端返回数据 做成一个标准的 格式

后端返回 数据 形式化 主要包含 code （状态码） data（数据源） message（返回信息）

返回数据格式

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

前端 发送请求之后 可以 根据 code 和 message 判定 返回的 状态

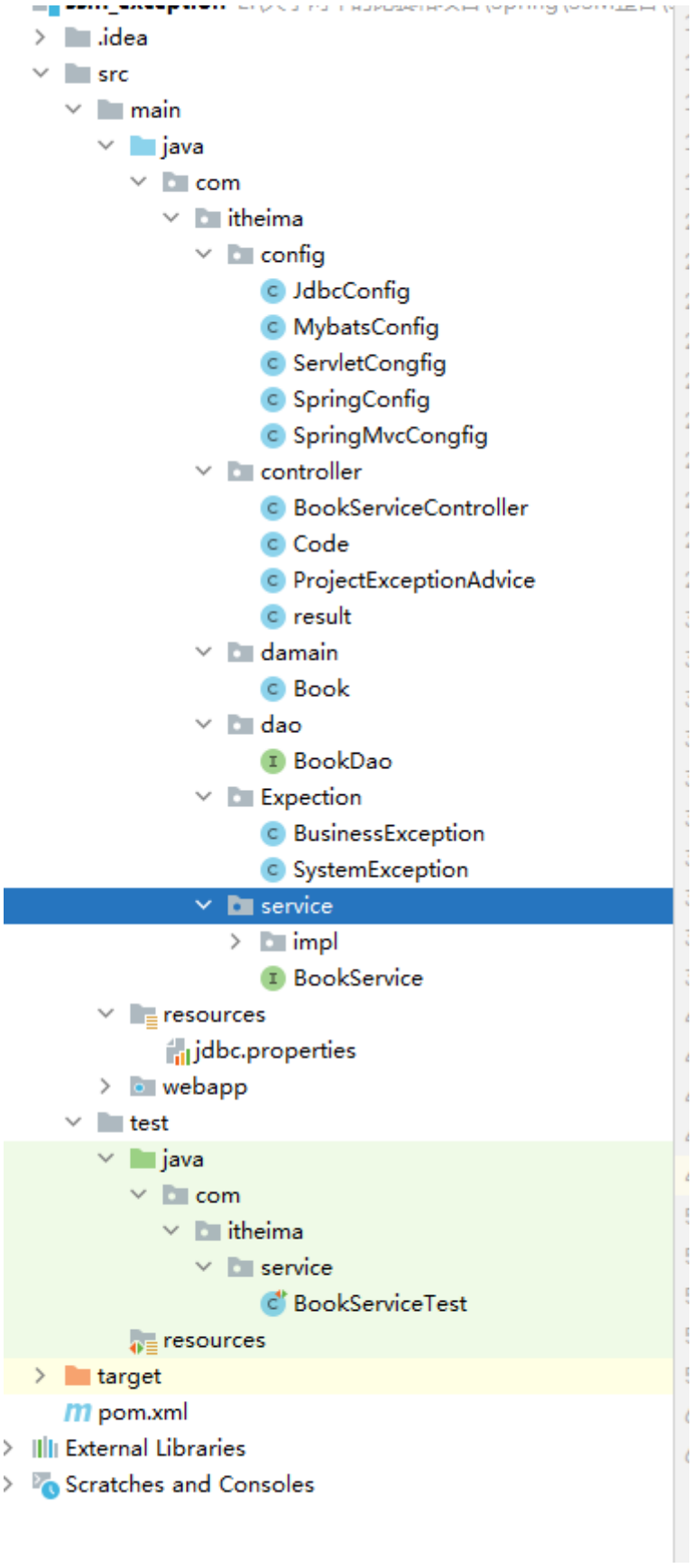
统一 拿数据 是从 data 中 拿取数据

```java
package com.itheima.controller;

/**
 * @ClassName result
```

```
 * @Description TODO
 * @Author lenovo
 * @Date 2022/9/3 9:15
 * @Version 1.0
 */

public class result {

    // 数据返回统一格式
    private Object data;

    //返回状态码
    private Integer code;

    //返回描述信息
    private String message;

    public result( Integer code,Object data) {
        this.data = data;
        this.code = code;
    }

    public result(Integer code, Object data, String message) {
        this.data = data;
        this.code = code;
        this.message = message;
    }

    // setter 方法是将 数据显示到 PostMan 页面上显示

    public Object getData() {

        return data;
    }

    public void setData(Object data) {
        this.data = data;
    }

    public Integer getCode() {
        return code;
    }

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

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

7 异常 处理

1 文件目录


.idea
src
main
java
com
itheima
config
JdbcConfig
MybatsConfig
ServletCongfig
SpringConfig
SpringMvcCongfig
controller
BookServiceController
Code
ProjectExceptionAdvice
result
damain
Book
dao
BookDao
Expection
BusinessException
SystemException
service
impl
BookService
resources
jdbc.properties
webapp
test
java
com
itheima
service
BookServiceTest
resources
target
pom.xml
External Libraries
Scratches and Consoles

2 在 controllerr 文件下 创建类 ProjectExceptionAdvice

使用 注解 @RestControllerAdvice @ExceptionHandler

所有的 异常 都会跑到 这个地方

```
package com.itheima.controller;

import com.itheima.Expection.BusinessException;
import com.itheima.Expection.SystemException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

/**
 * @ClassName ProjectExceptionAdvice
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/3 11:11
 * @Version 1.0
 */

@RestControllerAdvice

public class ProjectExceptionAdvice {

    @ExceptionHandler(Exception.class)
    public result doException(Exception exp){

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

        return new result(null,null,"error");

    }

}
```

3 将异常分类

构造方法 加上 set 和 get方法

1 BusinessException

```
package com.itheima.Expection;

/**
 * @ClassName BusinessException
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/3 15:13
 * @Version 1.0
 */

public class BusinessException extends RuntimeException{
```

```
    private Integer code;

    public BusinessException(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;

    }


    public Integer getCode() {

        return code;

    }

    public void setCode(Integer code) {

        this.code = code;

    }
}
```

2 SystemException

```
package com.itheima.Expection;

/**
 * @ClassName SystemException
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/3 15:13
 * @Version 1.0
 */

public class SystemException extends RuntimeException{

    private Integer code;

    public SystemException(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;
    }


    public Integer getCode() {
        return code;
    }

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

}
```

当有异常发生时 向 两个异常类抛出异常

```
if(id == 1){
    throw new BusinessException(Code.BUSINESS_ERR,"请不要使用你的技术挑战我的耐性!");
}

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);
}
```

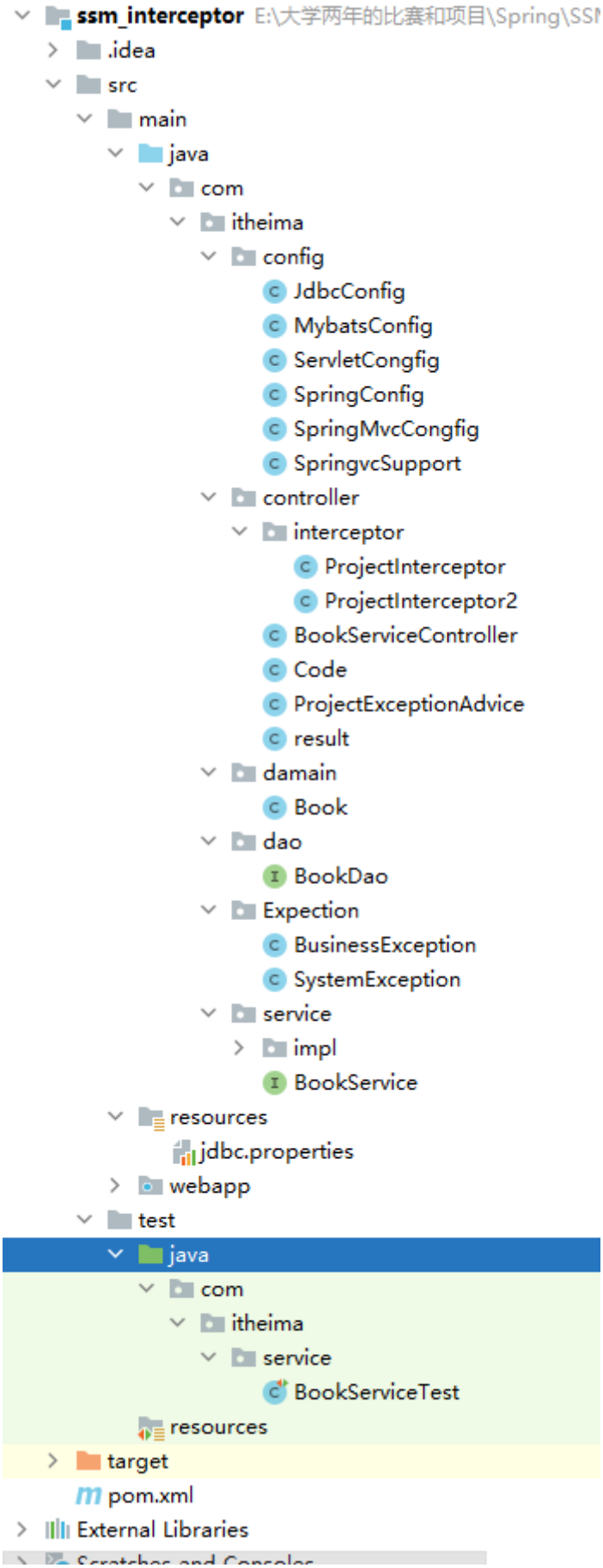
在 ProjectExceptionAdvice 中 接受 异常类 并且 向 result 返回信息

```
package com.itheima.controller;

import com.itheima.Expection.BusinessException;
import com.itheima.Expection.SystemException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

/**
 * @ClassName ProjectExceptionAdvice
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/3 11:11
 * @Version 1.0
```

```
 */

@RestControllerAdvice
public class ProjectExceptionAdvice {

    // 说明处理异常的种类
    @ExceptionHandler(SystemException.class)
    public result doSystemException(Exception exp){

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

        return new result(null,null,exp.getMessage());

    }


    @ExceptionHandler(BusinessException.class)
    public result doBusinessException(Exception exp){

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

        return new result(null,null,exp.getMessage());

    }

    @ExceptionHandler(Exception.class)
    public result doException(Exception exp){

        System.out.println("Exception is running...");

        return new result(null,null,"error");

    }
}
```

8 拦截器

1 目录文件


ssm_interceptor E:\大学两年的比赛和项目\Spring\SSM
.idea
src
main
java
com
itheima
config
JdbcConfig
MybatsConfig
ServletCongfig
SpringConfig
SpringMvcCongfig
SpringvcSupport
controller
interceptor
ProjectInterceptor
ProjectInterceptor2
BookServiceController
Code
ProjectExceptionAdvice
result
damain
Book
dao
BookDao
Expection
BusinessException
SystemException
service
impl
BookService
resources
jdbc.properties
webapp
test
java
com
itheima
service
BookServiceTest
resources
target
pom.xml
External Libraries

2在 controller 文件下创建 interceptor文件

在 该文件下 创建一个拦截器 ProjectInterceptor 类

```
package com.itheima.controller.interceptor;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * @ClassName ProjectInterceptor
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/3 16:38
 * @Version 1.0
 */

@Component
public class ProjectInterceptor implements HandlerInterceptor {

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

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

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

}
```

3 创建一个 SpringMvcSupport类

这个文件是加载到 SpringMvcConfig文件中去

配置 拦截地址

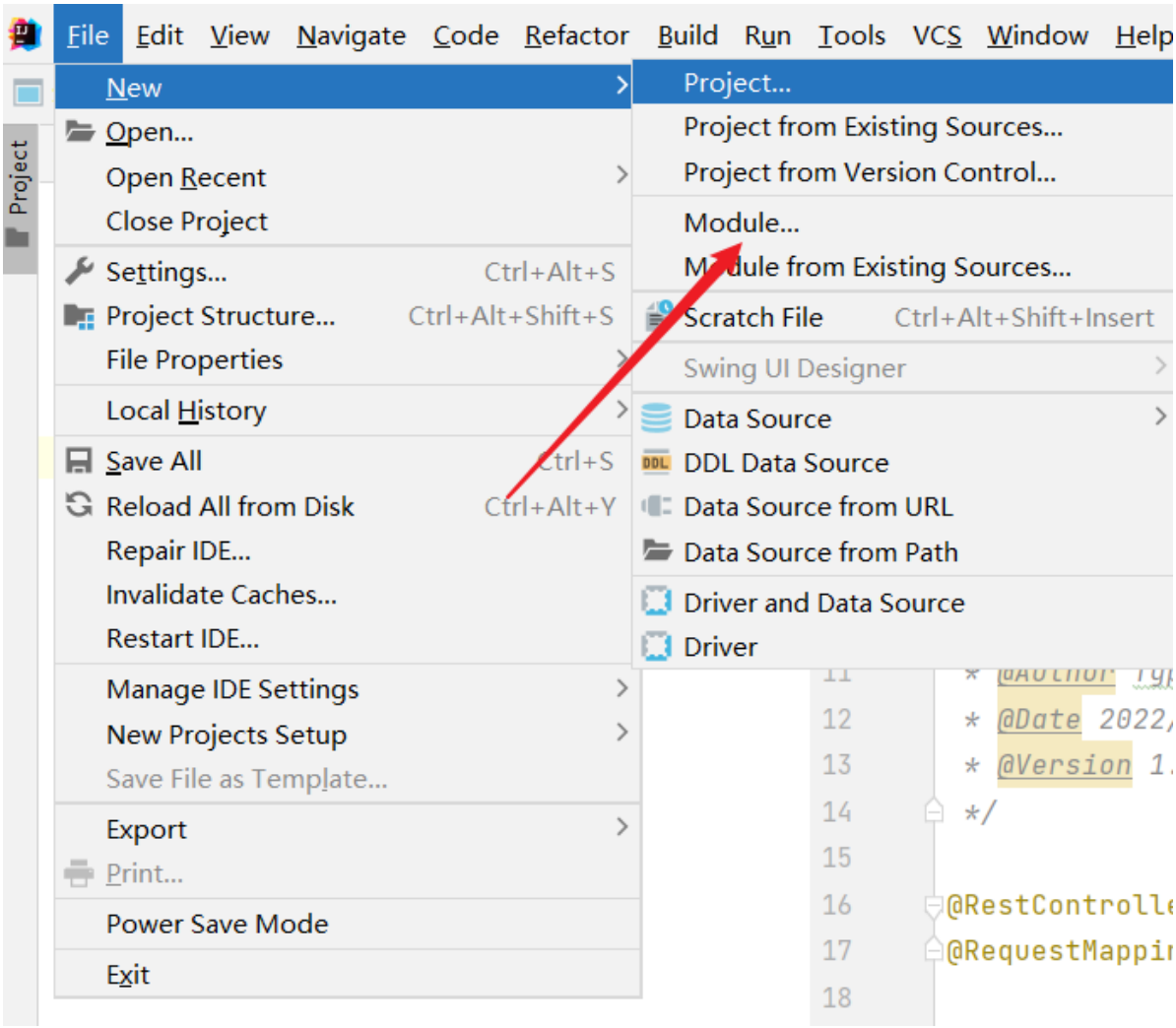
registry.addInterceptor(projectInterceptor).addPathPatterns("/Books","/Books/");**

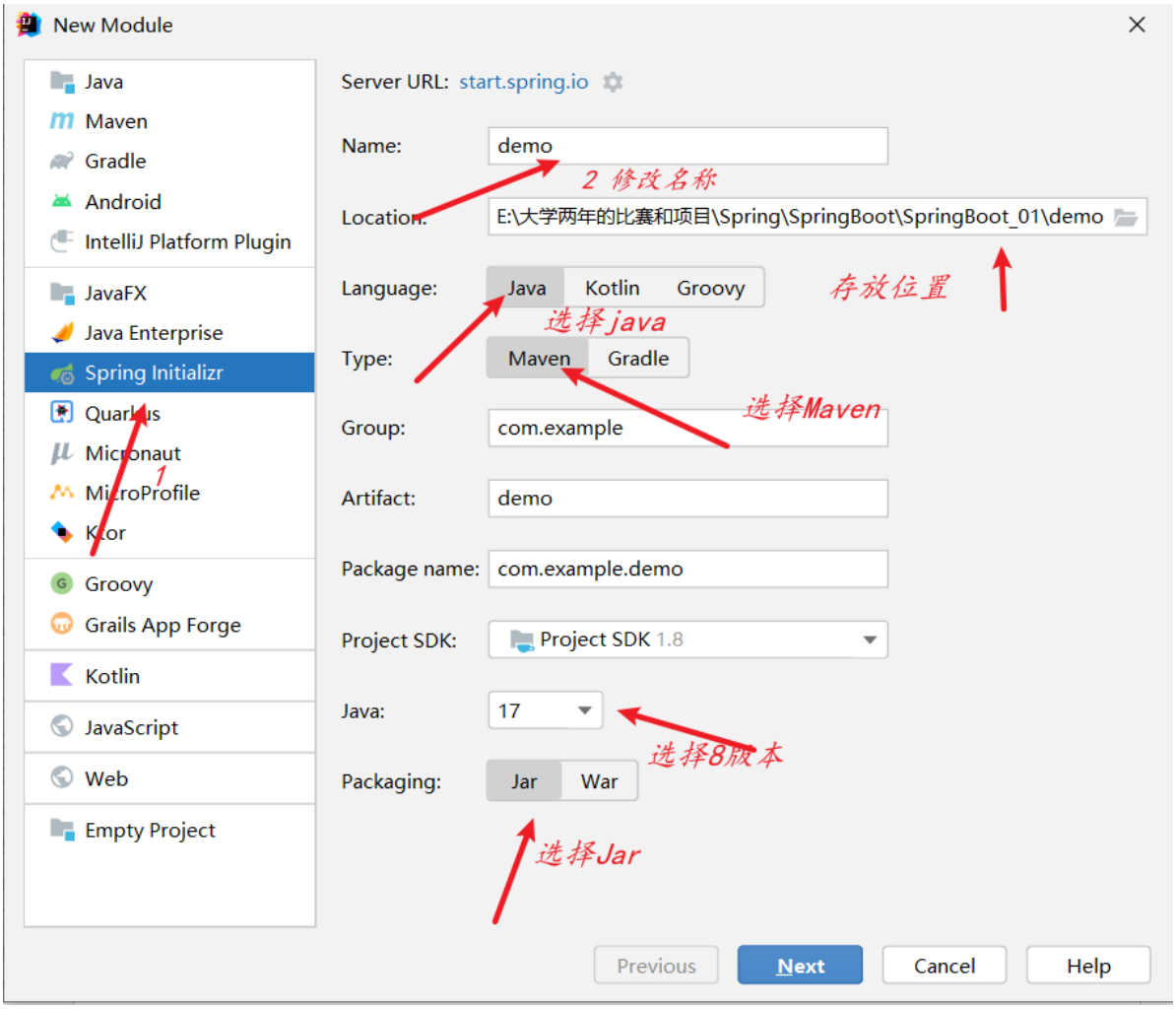
```
package com.itheima.config;
```

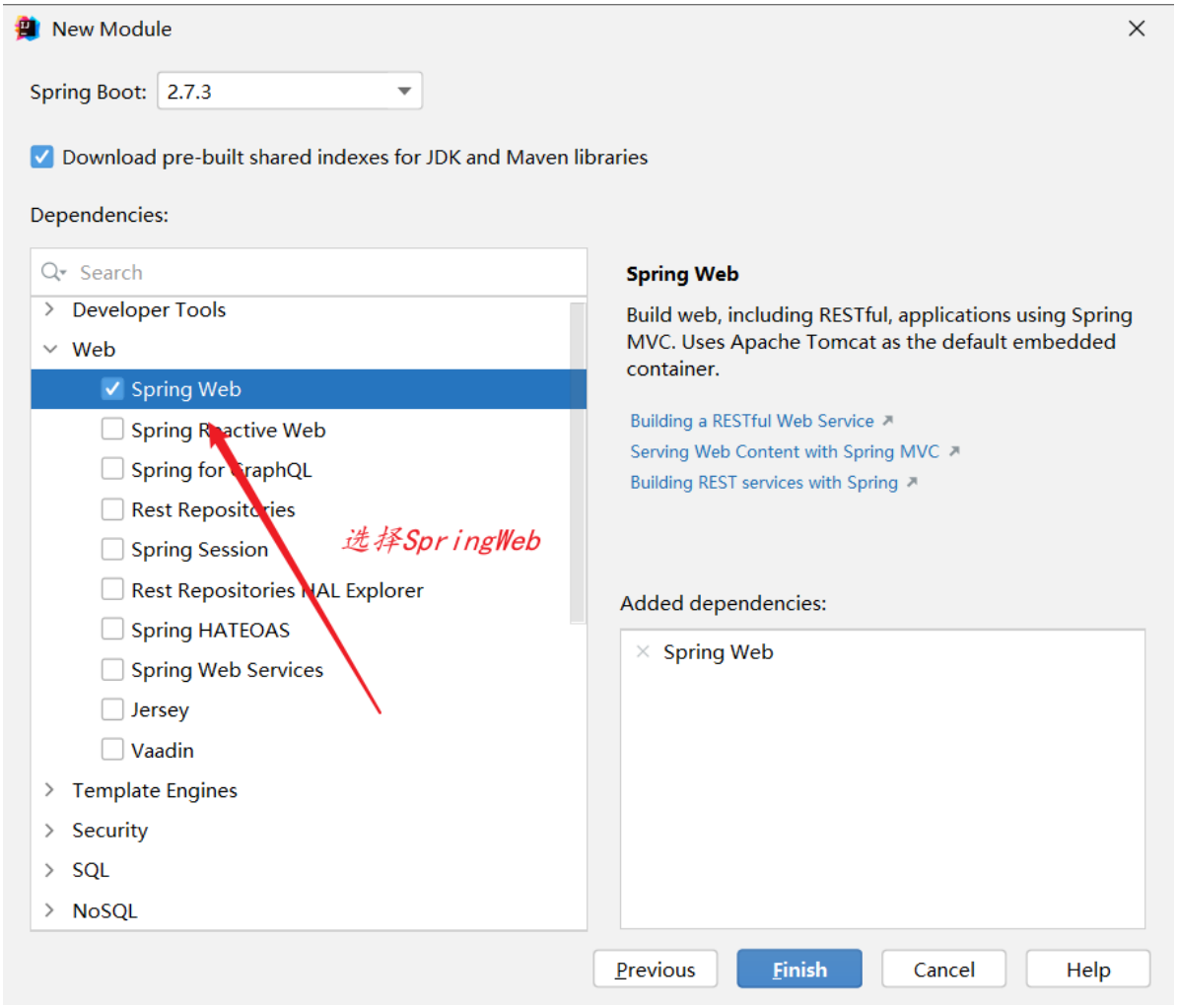
```
import com.itheima.controller.interceptor.ProjectInterceptor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import
org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;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

/**
 * @ClassName SpringvcSupport
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/3 16:41
 * @Version 1.0
 */

@Configuration
public class SpringMvcSupport extends WebMvcConfigurationSupport {

    @Autowired
    private ProjectInterceptor projectInterceptor;

    @Autowired
    // private ProjectInterceptor2 projectInterceptor2;

    @Override
    protected void addInterceptors(InterceptorRegistry registry) {

        // 配置多 拦截器


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

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

    @Override
    protected void addResourceHandlers(ResourceHandlerRegistry registry) {


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

    }

}
```

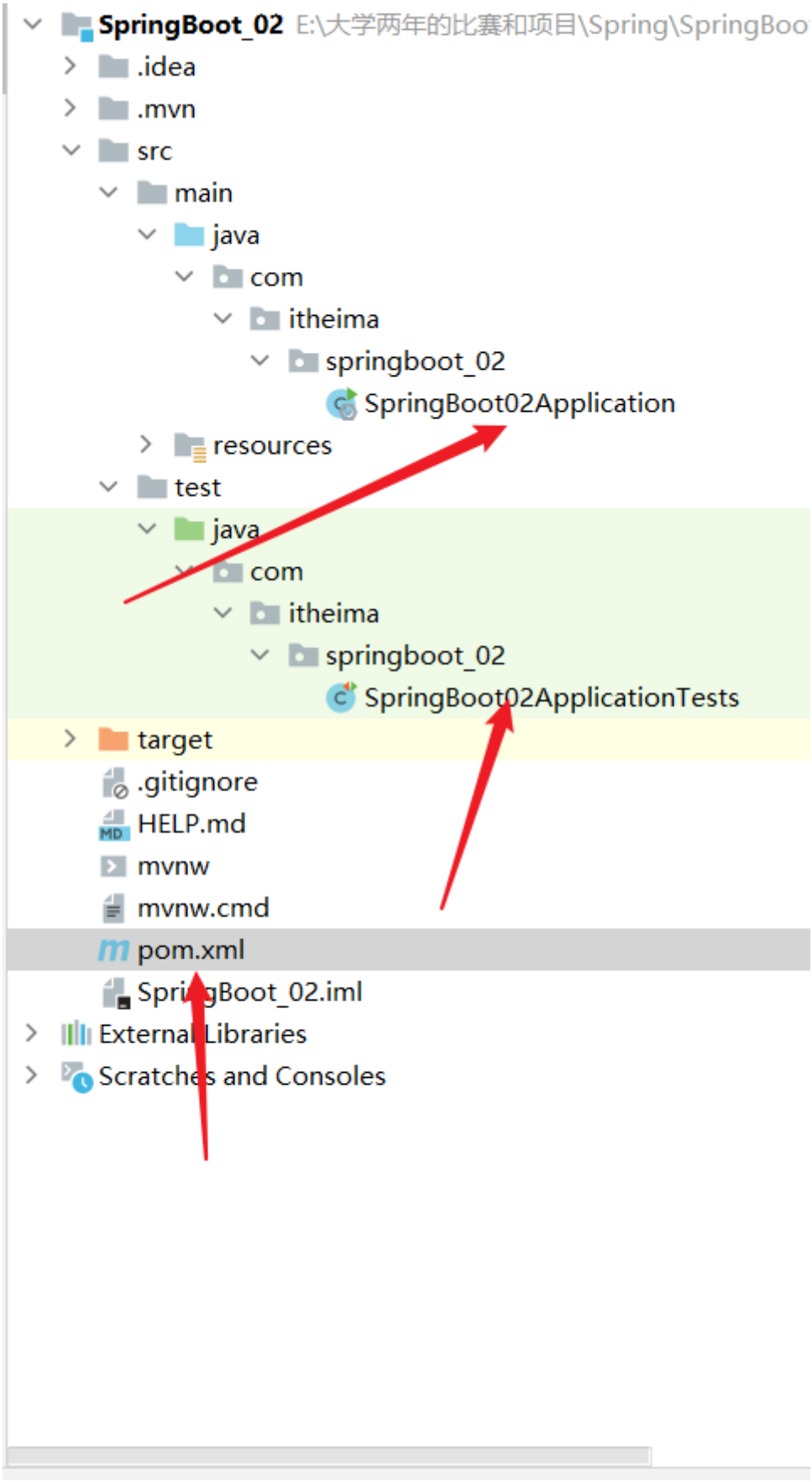
SpringBoot

1 创建文件


File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
New
Open...
Open Recent
Close Project
Settings... Ctrl+Alt+S
Project Structure... Ctrl+Alt+Shift+S
File Properties
Local History
Save All Ctrl+S
Reload All from Disk Ctrl+Alt+Y
Repair IDE...
Invalidate Caches...
Restart IDE...
Manage IDE Settings
New Projects Setup
Save File as Template...
Export
Print...
Power Save Mode
Exit
Project...
Project from Existing Sources...
Project from Version Control...
Module...
Module from Existing Sources...
Scratch File Ctrl+Alt+Shift+Insert
Swing UI Designer
Data Source
DDL Data Source
Data Source from URL
Data Source from Path
Driver and Data Source
Driver
Project
@RestControlle
@RequestMappi


New Module
Java
Maven
Gradle
Android
IntelliJ Platform Plugin
JavaFX
Java Enterprise
Spring Initializr
Quarkus
Micronaut
MicroProfile
Ktor
Groovy
Grails App Forge
Kotlin
JavaScript
Web
Empty Project
Server URL: start.spring.io
Name: demo
2 修改名称
Location: E:\大学两年的比赛和项目\Spring\SpringBoot\SpringBoot_01\demo
Language: Java Kotlin Groovy
存放位置
选择java
Type: Maven Gradle
选择Maven
Group: com.example
Artifact: demo
Package name: com.example.demo
Project SDK: Project SDK 1.8
Java: 17
选择8版本
Packaging: Jar War
选择Jar
1
Previous
Next
Cancel
Help


New Module
Spring Boot: 2.7.3
Download pre-built shared indexes for JDK and Maven libraries
Dependencies:
Search
Developer Tools
Web
Spring Web
Spring Reactive Web
Spring for GraphQL
Rest Repositories
Spring Session
Rest Repositories HAL Explorer
Spring HATEOAS
Spring Web Services
Jersey
Vaadin
Template Engines
Security
SQL
NoSQL
选择SpringWeb
Spring Web
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
Building a RESTful Web Service
Serving Web Content with Spring MVC
Building REST services with Spring
Added dependencies:
Spring Web
Previous
Finish
Cancel
Help


SpringBoot_02 E:\大学两年的比赛和项目\Spring\SpringBoo
.idea
.mvn
src
main
java
com
itheima
springboot_02
SpringBoot02Application
resources
test
java
com
itheima
springboot_02
SpringBoot02ApplicationTests
target
.gitignore
HELP.md
mvnw
mvnw.cmd
pom.xml
SpringBoot_02.iml
External Libraries
Scratches and Consoles

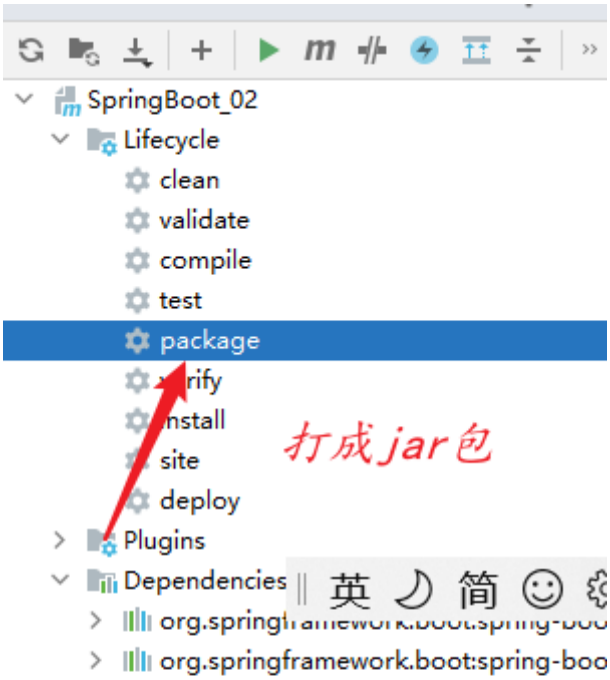
2 执行第一个SpringBoot文件

到PostMan中去执行 http://localhost:8080/Books/2

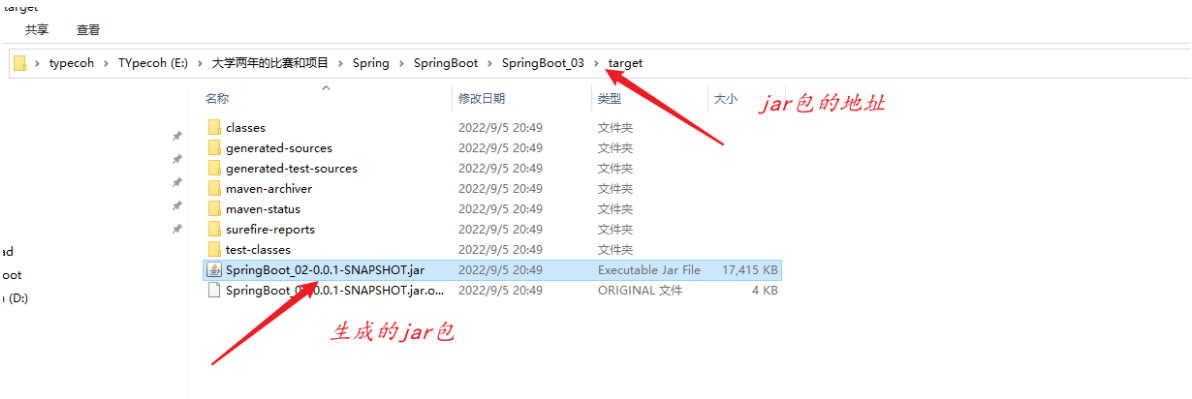
```
package com.itheima.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * @ClassName controller
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/5 12:29
 * @Version 1.0
 */
@RestController
@RequestMapping("/Books")
public class controller {

    @GetMapping("/{id}")
    public String getById(@PathVariable int id){

        System.out.println("Id == >" + id);

        return "hello SpringBoot!";

    }

}
```

3 快速运行开发

1 打成jar包



2 生成jar



3 执行jar

4 测试cmd执行的代码

4 配置文件格式

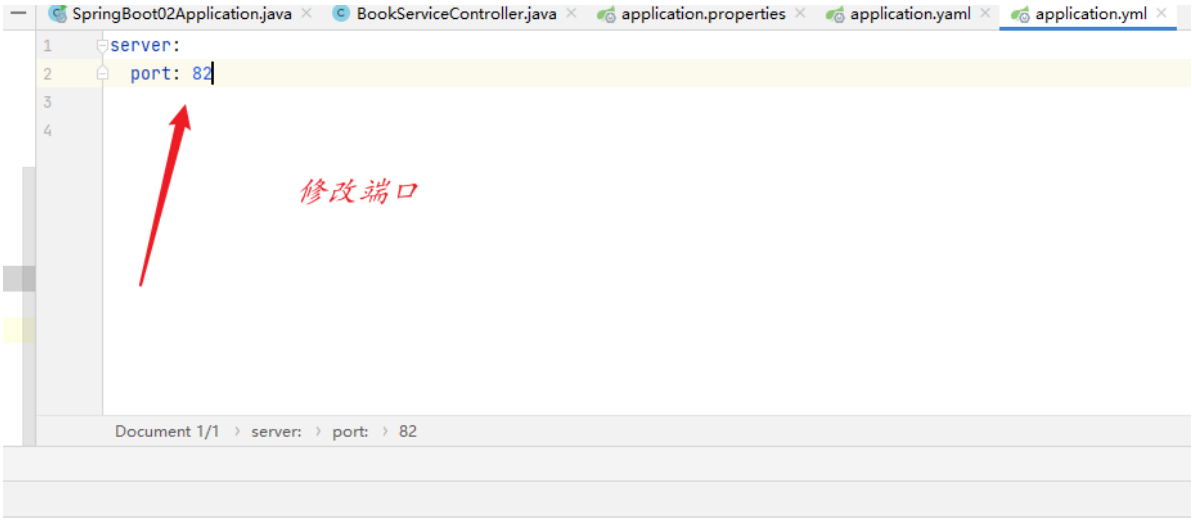
1 properties 文件

2 yaml文件


```
main] com.itheima.SpringBoot02Application      : Starting SpringBoot02Application using Java 1.8.0_151 on DESK
main] com.itheima.SpringBoot02Application      : No active profile set, falling back to 1 default profile: "de
main] org.eclipse.jetty.util.log               : Logging initialized @2186ms to org.eclipse.jetty.util.log.Slf
main] o.s.b.w.e.j.JettyServletWebServerFactory : Server initialized with port: 82
```

修改成功

3 yml文件




```
main] com.itheima.SpringBoot02Application      : Starting SpringBoot02Application using Java 1.8.0_151 on DESKTOP
main] com.itheima.SpringBoot02Application      : No active profile set, falling back to 1 default profile: "defau
main] org.eclipse.jetty.util.log               : Logging initialized @2074ms to org.eclipse.jetty.util.log.Slf4jL
main] o.s.b.w.e.j.JettyServletWebServerFactory : Server initialized with port: 82
main] org.eclipse.jetty.server.Server          : jetty-9.4.48.v20220622; built: 2022-06-21T20:42:25.880Z; git: 6b
main] o.e.j.s.h.ContextHandler.application     : Initializing Spring embedded WebApplicationContext
```

端口修改成功

需要注意的是这三个顺序:

application.properties > application.yml > application.yaml

5 文件数据格式配置

1 使用properties文件配置数据

```
#设置properties文件的内容格式
enterprise.name=itcast
enterprise.age=16
enterprise.tel=4006184000
```

注入变量

```
@Value("${enterprise.name}")
private String name;
@Value("${enterprise.age}")
private Integer age;
@Value("${enterprise.tel}")
private String tel;
```

测试文件

```
package com.itheima.controller;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * @ClassName GetDataController
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/5 21:12
 * @Version 1.0
 */

@RestController
@RequestMapping("/getdata")
public class GetDataController {

    @Value("${enterprise.name}")
    private String name;
    @Value("${enterprise.age}")
    private Integer age;
    @Value("${enterprise.tel}")
    private String tel;

    @GetMapping
    public String GetData(){

        System.out.println("使用properties文件格式设置 数据格式");
        System.out.println("name ==>" + name);
        System.out.println("name ==>" + age);
        System.out.println("name ==>" + tel);
        System.out.println("-------------------------");
        return "hello SpringBoot!!!";
```

```
    }
}
```

借助Environment对象

```
@Autowired
private Environment environment;

@GetMapping
public String GetDataEnvironment(){

    System.out.println("使用properties文件格式设置 数据格式 并且加上Environment 类");
    System.out.println(environment.getProperty("enterprise.name"));
    System.out.println(environment.getProperty("enterprise.age"));
    System.out.println(environment.getProperty("enterprise.tel"));
    System.out.println("--------------------------");
    return "hello SpringBoot!!!";

}
```

那既然存在这个Environment 那不妨就自己在一个类

创建一个Enterprise

```
package com.itheima.domain;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

import java.util.Arrays;

/**
 * @ClassName Enterprise
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/5 21:10
 * @Version 1.0
 */
@Component
@ConfigurationProperties(prefix = "enterprise")
public class Enterprise {

    private String name;
    private Integer age;
    private String tel;
    private String[] likes;

    @Override
    public String toString() {
        return "Enterprise{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", tel='" + tel + '\'' +
                ", likes=" + Arrays.toString(likes) +
                '}';
    }
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public String getTel() {
        return tel;
    }

    public void setTel(String tel) {
        this.tel = tel;
    }

    public String[] getLikes() {
        return likes;
    }

    public void setLikes(String[] likes) {
        this.likes = likes;
    }
}
```

```
@Autowired
private Enterprise enterprise;

@GetMapping
public String getdataEnterprise(){

        System.out.println("使用properties文件格式设置 数据格式 并且加上Enterprise
类");
        System.out.println(enterprise.getName());
        System.out.println(enterprise.getAge());
        System.out.println(enterprise.getTel());
        System.out.println("-------------------------");
        return "hello SpringBoot!!!";

}
```

2 使用yml文件

```yaml
#server:
#  port: 82
#

#使用 yml 文件格式进行配置
enterprise:
  name: itcast
  age: 16
  tel: 4006184000
  subject:
    - Java
    - Web
    - Node
    - Sql
```

```java
package com.itheima.controller;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * @ClassName GetDataFromYmlController
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/5 21:30
 * @Version 1.0
 */
@RestController
@RequestMapping("/getdata/yml")
public class GetDataFromYmlController {

    @Value("${enterprise.name}")
    private String name;
    @Value("${enterprise.age}")
    private Integer age;
    @Value("${enterprise.tel}")
    private String tel;
    @Value("${enterprise.subject[1]}")
    private String subject;

    @GetMapping
    public String getdata(){


        System.out.println("使用yml文件格式进行显示");

        System.out.println("name ==>" + name);
        System.out.println("name ==>" + age);
        System.out.println("name ==>" + tel);
        System.out.println("subject ==>" + subject);
```

```
        return "hello SpringBoot!!!";
    }

}
```

yml 文件使用 自己创造的类进行数据封装

```
package com.itheima.controller;

import com.itheima.domain.Enterprise;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * @ClassName GetDataFromYmlController
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/5 21:30
 * @Version 1.0
 */
@RestController
@RequestMapping("/getdata/yml")
public class GetDataFromYmlController {

    // @Value("${enterprise.name}")
    // private String name;
    // @Value("${enterprise.age}")
    // private Integer age;
    // @Value("${enterprise.tel}")
    // private String tel;
    // @Value("${enterprise.subject[1]}")
    // private String subject;
    //
    // @GetMapping
    // public String getdata(){
    //
    //
    //     System.out.println("使用yml文件格式进行显示");
    //
    //     System.out.println("name ==>" + name);
    //     System.out.println("name ==>" + age);
    //     System.out.println("name ==>" + tel);
    //     System.out.println("subject ==>" + subject);
    //
    //
    //     return "hello SpringBoot!!!";
    // }

    @Autowired
    private Enterprise enterprise;

    @GetMapping
```

```
    public String getdata(){


        System.out.println("使用yml文件格式进行显示");

        System.out.println("name ==>" + enterprise.getName());
        System.out.println("name ==>" + enterprise.getAge());
        System.out.println("name ==>" + enterprise.getTel());
        // enterprise.getSubject() 返回一个数组名 + 【1】 得到对应的内容
        System.out.println("subject ==>" + enterprise.getSubject()[1]);


        return "hello SpringBoot!!!";
    }

}
```

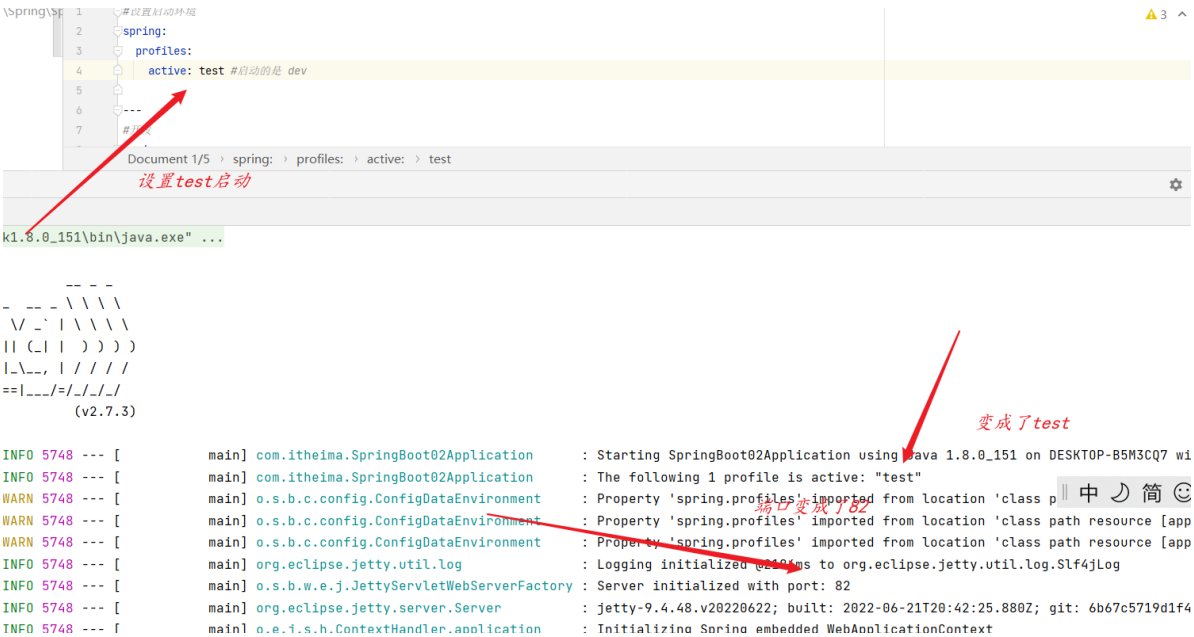
6 多环境配置

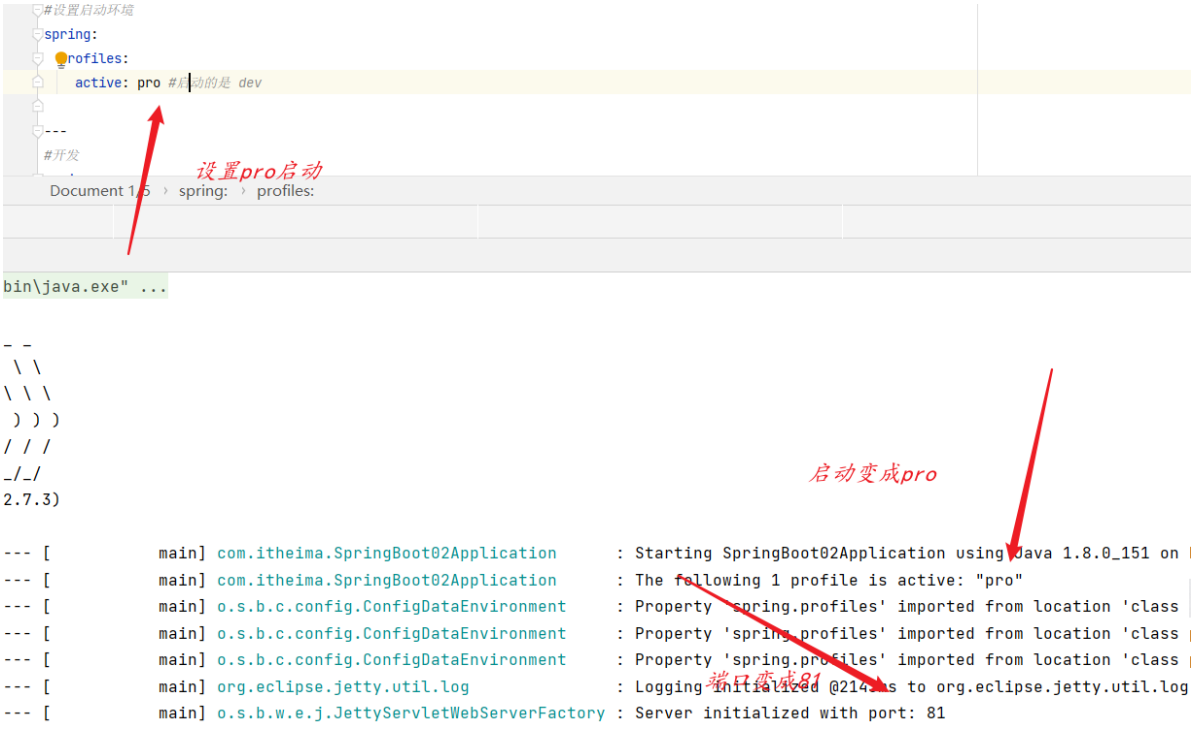
环境大致分为 开发环境 测试环境 生产环境

多环境的好处是什么呢?

为了避免 在这三种环境之间相互转换，SpringBoot 给我们提供了很方便的配置，避免转换

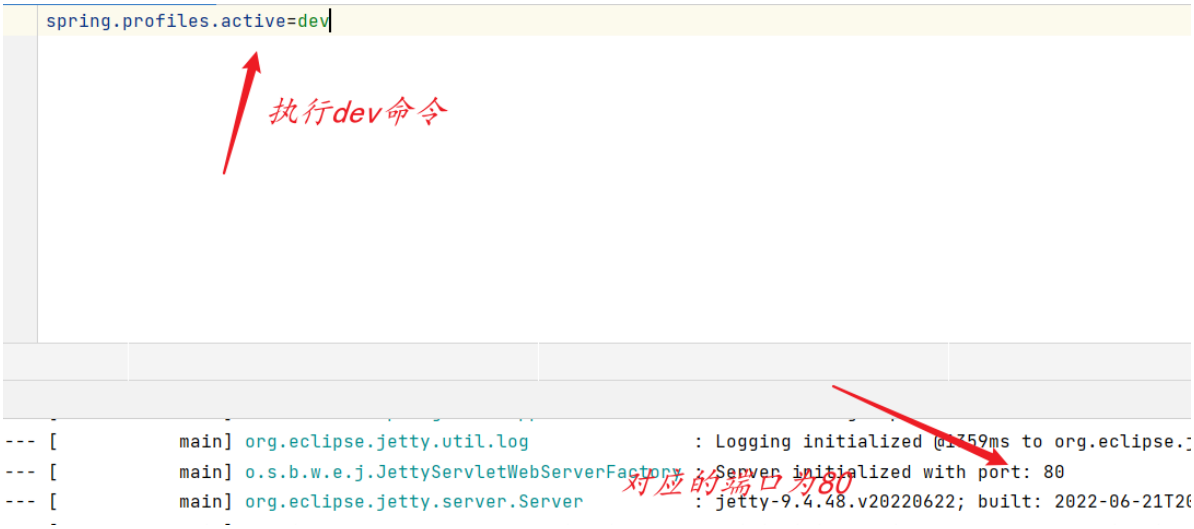
1 在yml 和 yaml文件中

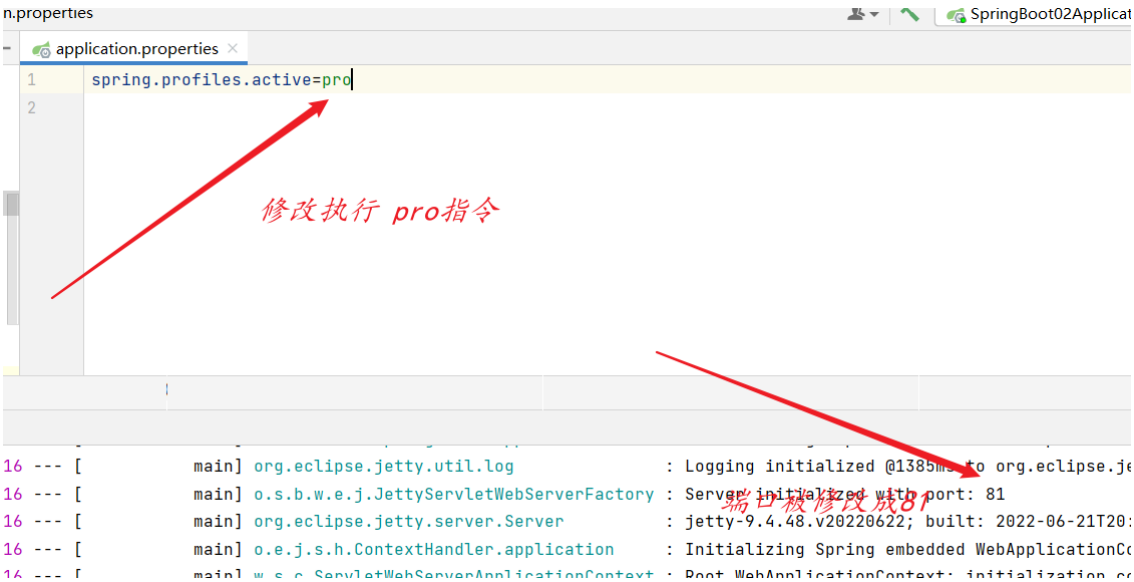


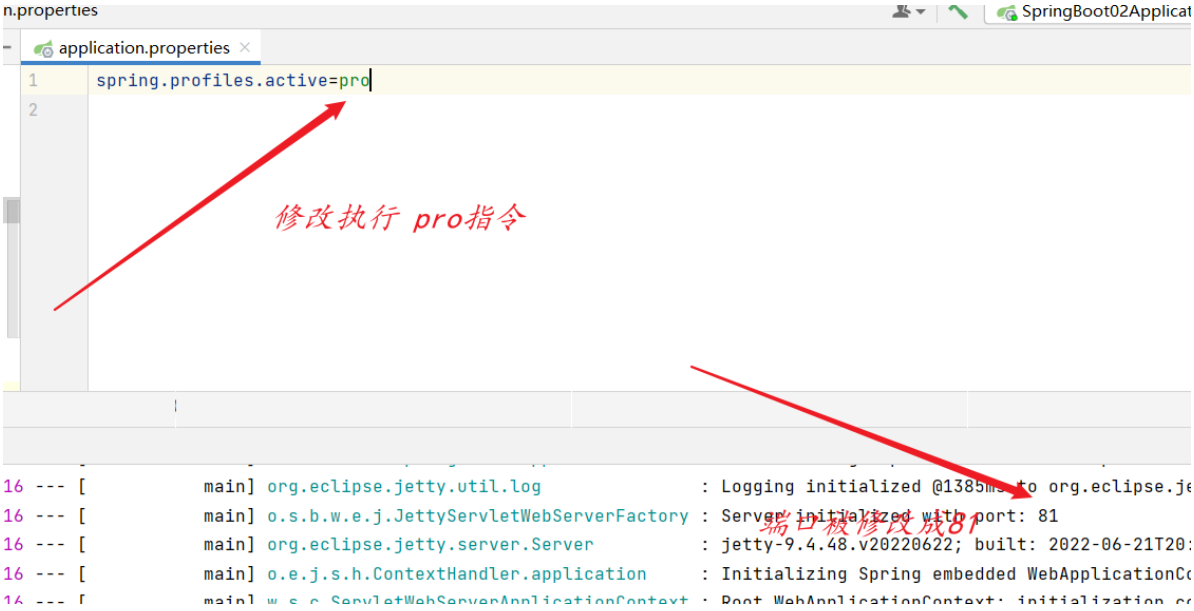


2 在properties文件中

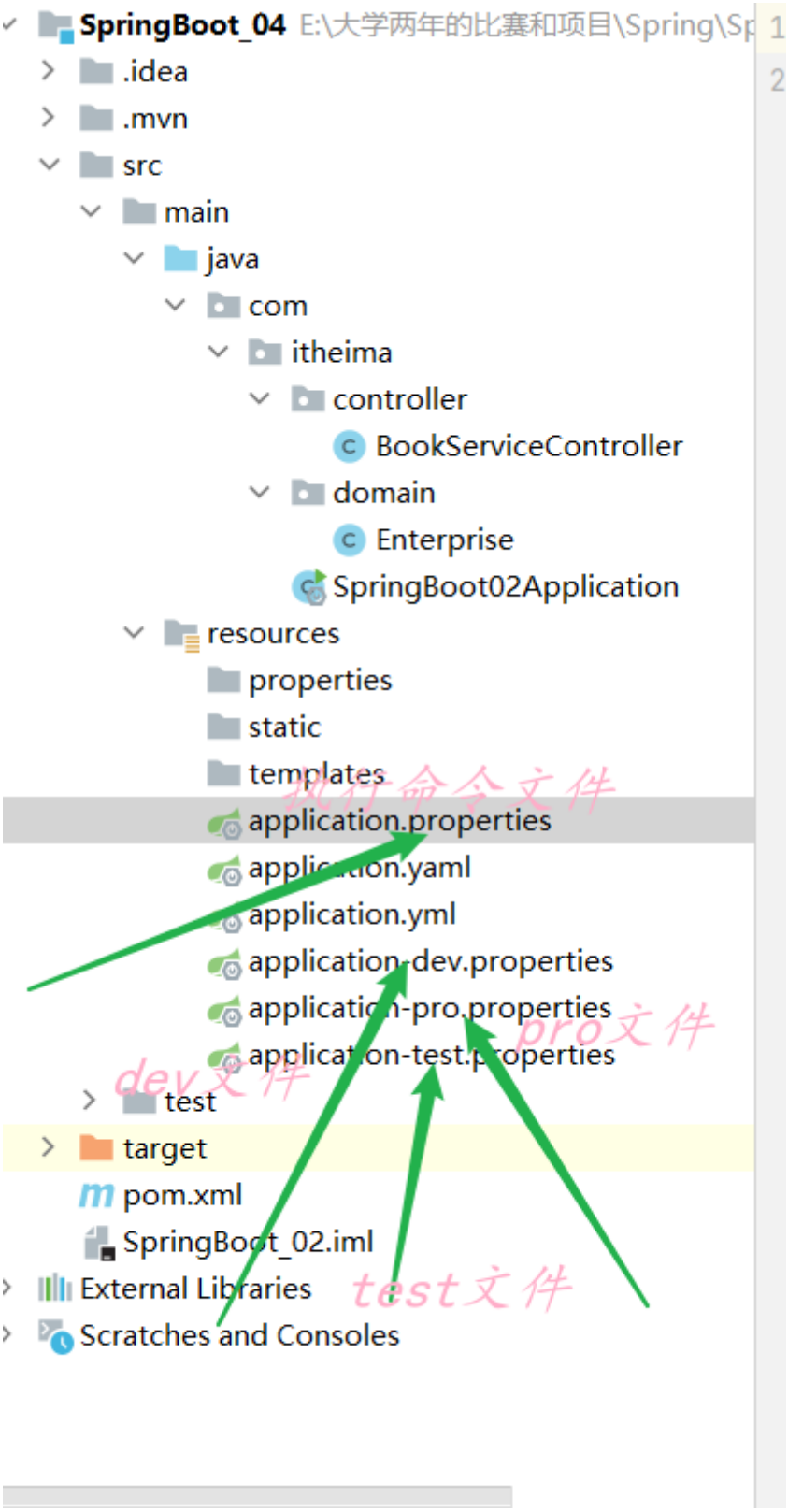
分别执行 dev pro test等执行命令


spring.profiles.active=dev
执行dev命令
--- [main] org.eclipse.jetty.util.log : Logging initialized @1759ms to org.eclipse.
--- [main] o.s.b.w.e.j.JettyServletWebServerFactory : Server initialized with port: 80
对应的端口为80
--- [main] org.eclipse.jetty.server.Server : jetty-9.4.48.v20220622; built: 2022-06-21T20


n.properties
SpringBoot02Applicat
application.properties
1 spring.profiles.active=pro
2
修改执行 pro指令
16 --- [main] org.eclipse.jetty.util.log : Logging initialized @1385ms to org.eclipse.je
16 --- [main] o.s.b.w.e.j.JettyServletWebServerFactory : Server initialized with port: 81
端口被修改成81
16 --- [main] org.eclipse.jetty.server.Server : jetty-9.4.48.v20220622; built: 2022-06-21T20:
16 --- [main] o.e.j.s.h.ContextHandler.application : Initializing Spring embedded WebApplicationC


n.properties
SpringBoot02Applicat
application.properties
1 spring.profiles.active=pro
2
修改执行 pro指令
16 --- [main] org.eclipse.jetty.util.log : Logging initialized @1385ms to org.eclipse.je
16 --- [main] o.s.b.w.e.j.JettyServletWebServerFactory : Server initialized with port: 81
端口被修改成81
16 --- [main] org.eclipse.jetty.server.Server : jetty-9.4.48.v20220622; built: 2022-06-21T20:
16 --- [main] o.e.j.s.h.ContextHandler.application : Initializing Spring embedded WebApplicationC

文件相对位置


SpringBoot_04 E:\大学两年的比赛和项目\Spring\Sp
.idea
.mvn
src
main
java
com
itheima
controller
BookServiceController
domain
Enterprise
SpringBoot02Application
resources
properties
static
templates
application.properties
application.yaml
application.yml
application-dev.properties
application-pro.properties
application-test.properties
test
target
pom.xml
SpringBoot_02.iml
External Libraries
Scratches and Consoles
执行命令文件
dev文件
pro文件
test文件

我们知道 `jar` 包其实就是一个压缩包，可以解压缩，然后修改配置，最后再打成jar包就可以了。这种方式显然有点麻烦，而 `SpringBoot` 提供了在运行 `jar` 时设置开启指定的环境的方式，如下

```
java -jar xxx.jar --spring.profiles.active=test
```

中

那么这种方式能不能临时修改端口号呢？也是可以的，可以通过如下方式

```
java -jar xxx.jar --server.port=88
```

当然也可以同时设置多个配置，比如即指定启用哪个环境配置，又临时指定端口，如下

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

大家进行测试后就会发现命令行设置的端口号优先级高（也就是使用的是命令行设置的端口号），配置的优先级其实 `SpringBoot` 官网已经进行了说明，参见：

```
https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-external-config
```

7 SpringBoot整合

1 SpringBoot整合Junit

1 在service层造一个接口

```java
package com.itheima.service;

/**
 * @ClassName BookService
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/6 14:29
 * @Version 1.0
 */

public interface BookService {

    public void save();

}
```

在 impl 实现这个接口

```java
package com.itheima.service.impl;

import com.itheima.service.BookService;
import org.springframework.stereotype.Service;

/**
 * @ClassName BookServiceImpl
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/6 14:29
 * @Version 1.0
 */
@Service
```

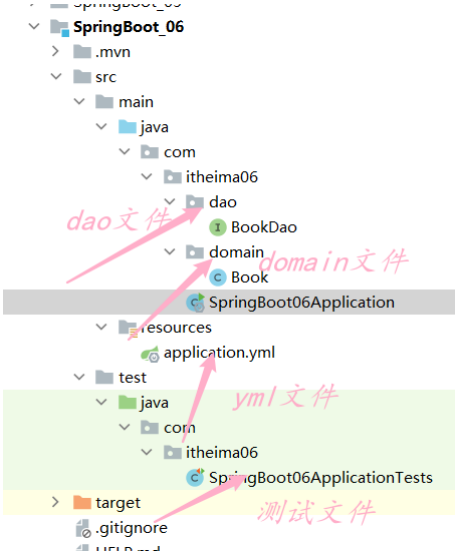
```
public class BookServiceImpl implements BookService {

    @Override
    public void save() {

        System.out.println("save is running...");

    }
}
```

2 创一个Test类

```
package com.itheima;

import com.itheima.service.BookService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

/**
 * @ClassName Test01
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/6 14:41
 * @Version 1.0
 */

@SpringBootTest
public class Test01 {


    @Autowired
    private BookService bookService;

    @Test
    public void Test(){

        System.out.println(1);

        bookService.save();

    }

}
```

2 SpringBoot 整合Mybatis

1 文件目录



2 创建一个domain文件

这个文件放置自己的 Book类

```
package com.itheima06.domain;

/**
 * @ClassName Book
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/6 17:46
 * @Version 1.0
 */

public class Book {

    private Integer id;
    private String type;
    private String description;
    private String name;

    @Override
    public String toString() {
        return "Book{" +
                "id=" + id +
                ", type='" + type + '\'' +
                ", description='" + description + '\'' +
                ", name='" + name + '\'' +
                '}';
    }

    public Integer getId() {
        return id;
    }

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

    public String getType() {
        return type;
    }
```

```
    public void setType(String type) {
        this.type = type;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

2 创建dao文件

这个文件主要是放置 sql 查询语句

```
package com.itheima06.dao;

import com.itheima06.domain.Book;
import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Select;

/**
 * @ClassName BookDao
 * @Description TODO
 * @Author lenovo
 * @Date 2022/9/6 17:47
 * @Version 1.0
 */

@Mapper
public interface BookDao {

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

}
```

3 生成yml文件

```yaml
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    username: root
    password: admin123
    url: jdbc:mysql://localhost/mybatis
```

4 生成一个测试类进行测试

```java
package com.itheima06;

import com.itheima06.dao.BookDao;
import com.itheima06.domain.Book;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class SpringBoot06ApplicationTests {

    @Autowired
    private BookDao bookDao;

    @Test
    void DoGetById() {

        System.out.println(1);

        Book book = bookDao.getById(2);

        System.out.println(book);

    }

}
```

至此 SpringBoot整合 Mybatis 结束

MybatisPlus

1 入门程序

在dao的 UserDao中代码修改

```java
package com.itheima.dao;

import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.itheima.domain.Book;
import org.apache.ibatis.annotations.Mapper;

/**
```

```
 * @ClassName UserDao
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/6 21:21
 * @Version 1.0
 */

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

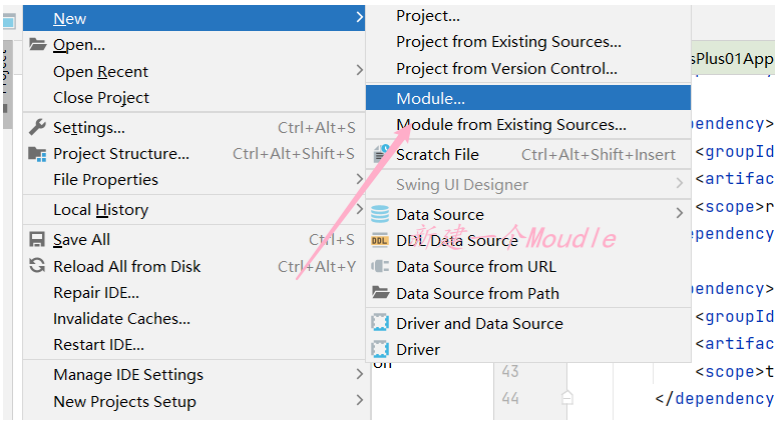
在测试中代码

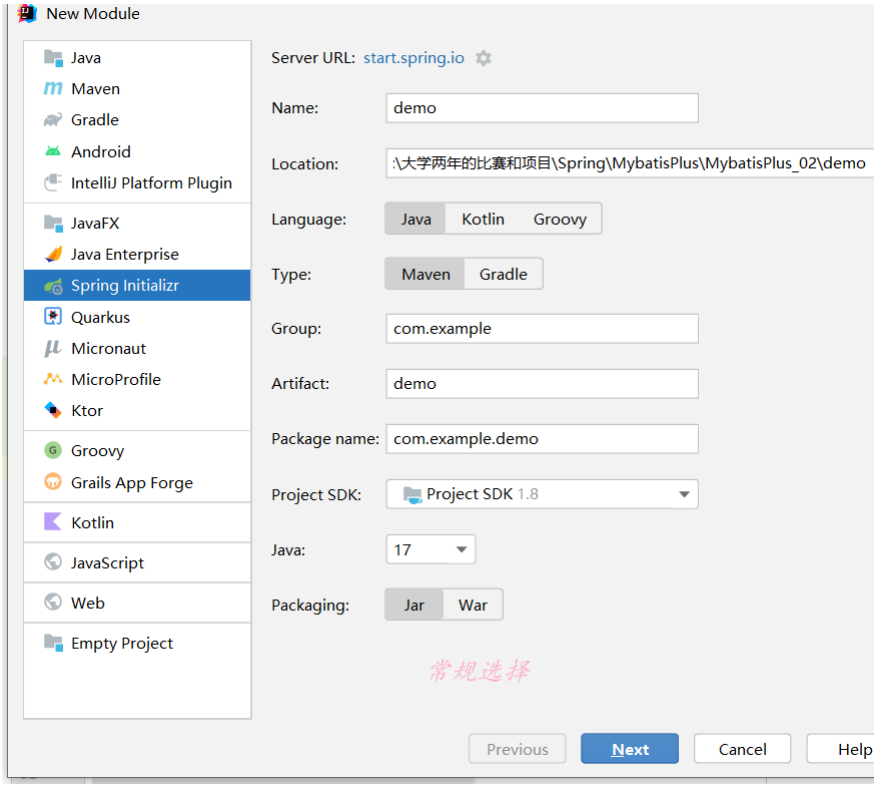
```
@Autowired
private UserDao userDao;
@Test
void TestGetAll(){

    List<Book> list = userDao.selectList(null);

    System.out.println(list);

}
```

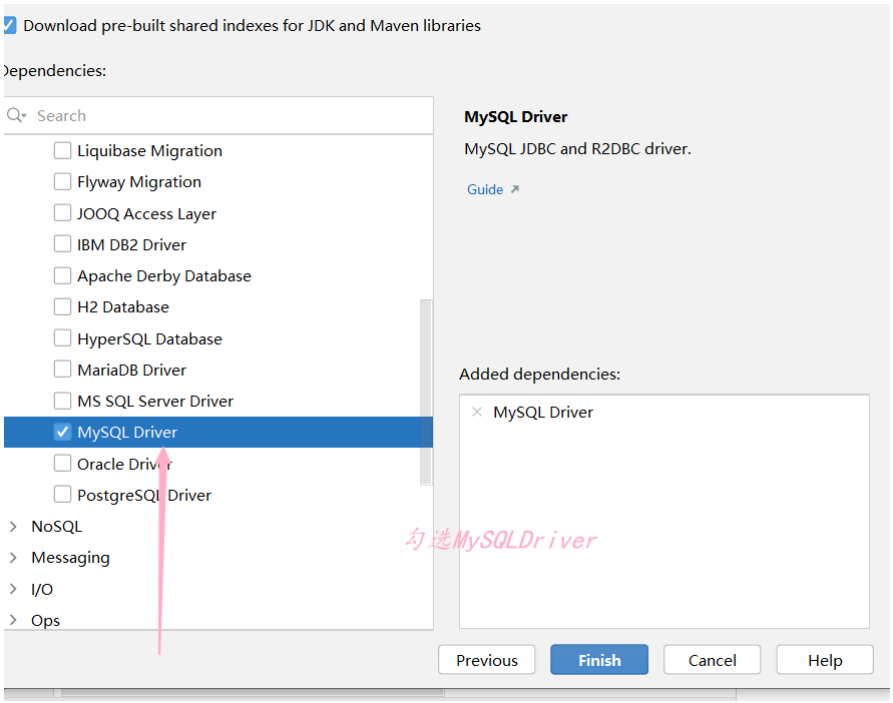
- **刚刚认识MybatisPlus 发现没有写任何查询或者其他的sql语句 但是在调用的时候一个都不少**

这就是惊人之处

2 创建MybatisPlus步骤







配置yml文件

```yaml
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    username: root
    password: admin123
    url: jdbc:mysql://localhost:3306/mybatis
```

使用继承BaseMapper 之后的 SQL函数

```
package com.itheima;

import com.itheima.dao.UserDao;
import com.itheima.domain.Book;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.List;

@SpringBootTest
class MybatisPlus01ApplicationTests {

    @Autowired
    private UserDao userDao;
    @Test
    void TestGetAll(){

        List<Book> list = userDao.selectList(null);

        System.out.println(list);

    }

    @Test
    void TestAdd(){

       Book book = new Book();

       book.setId(14);

       book.setType("Spring实战 第6版");

       book.setName("计算机导论");

       book.setDescription("十年沉淀之作，手写MybatisPlus精华思想");

       userDao.insert(book);

    }

    @Test
    void TestDelete(){

       userDao.deleteById(14);

    }

    @Test
    void TestUpdate(){

        Book book = new Book();

        book.setId(1);
```

```
        book.setName("MybatisPlus");

        book.setType("hello world");

        book.setDescription("深入MyabtisPlus源码剖析MyabtisPlus源码中蕴含的10大设计模式");

        userDao.updateById(book);

    }

    @Test
    void TestGetById(){

        Book book = userDao.selectById(2);

        System.out.println(book);

    }


}
```

对domain中的类进行简化

原始程序

```
public class Book {

    private Integer id;
    private String type;
    private String name;
    private String description;

    public Integer getId() {
        return id;
    }

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

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    @Override
    public String toString() {
        return "Book{" +
                "id=" + id +
                ", type='" + type + '\'' +
                ", name='" + name + '\'' +
                ", description='" + description + '\'' +
                '}';
    }
}
```

加入依赖之后的文件

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

```
package com.itheima.domain;

import lombok.*;

/**
 * @ClassName Book
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/6 21:15
 * @Version 1.0
 */

@Getter // 提供所有的 get方法
@Setter // 提供所有的 set方法
@ToString // 提供所有的 Tostring 方法
@EqualsAndHashCode // 为模型类的属性提供equals和hashcode方法
@Data //  将上述几个注解 合并在一起
@NoArgsConstructor // 无参构造方法
@AllArgsConstructor  // 提供包含所有参数的 构造方法

public class Book {

    private Integer id;
    private String type;
    private String name;
    private String description;
```

```
}
```

3 分页功能

创建一个 IPage类

```
//在这里设定 当前页 和 每页数量
IPage iPage = new Page(1,2);
```

调用查询Page方法

```
userDao.selectPage(iPage,null);
```

设置查询条件

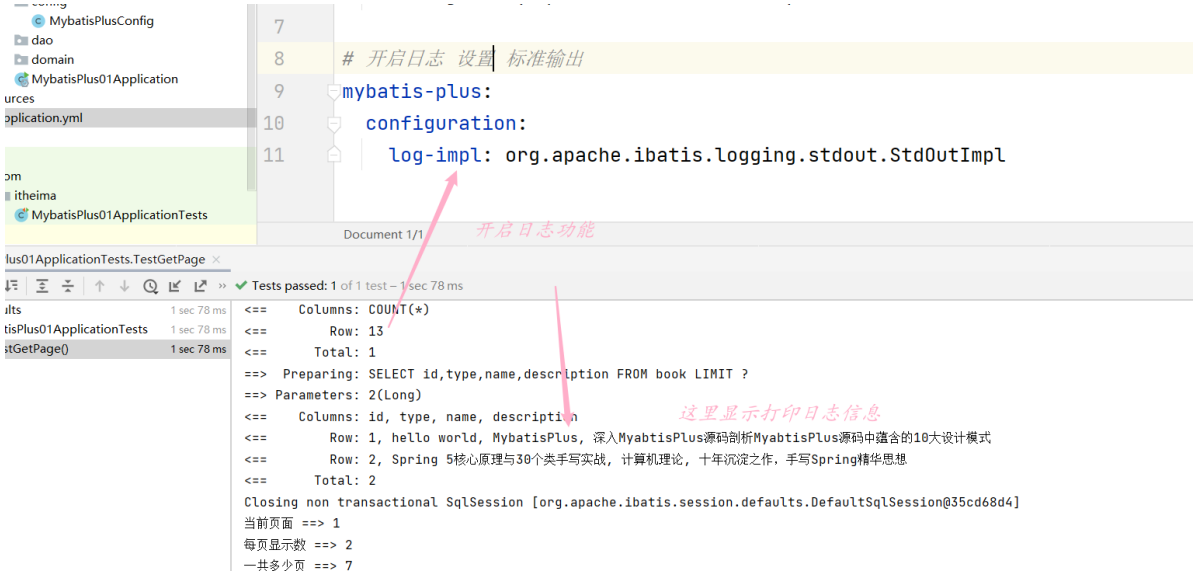
```
//这个是获得自己设定的数据
System.out.println("当前页面 ==> " + iPage.getCurrent());
System.out.println("每页显示数 ==> " + iPage.getSize());
//这里需要设置拦截器 获取数据
System.out.println("一共多少页 ==> " + iPage.getPages());
System.out.println("一共多少数据 ==> " + iPage.getTotal());
//这里是获取所有的数据源
System.out.println("数据 ==> " + iPage.getRecords());
```

```
package com.itheima;

import com.baomidou.mybatisplus.core.metadata.IPage;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.itheima.dao.UserDao;
import com.itheima.domain.Book;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.List;

@SpringBootTest
class MybatisPlus01ApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    void TestGetPage(){

        //在这里设定 当前页 和 每页数量
        IPage iPage = new Page(1,2);

        userDao.selectPage(iPage,null);

        //这个是获得自己设定的数据
        System.out.println("当前页面 ==> " + iPage.getCurrent());
        System.out.println("每页显示数 ==> " + iPage.getSize());
        //这里需要设置拦截器 获取数据
        System.out.println("一共多少页 ==> " + iPage.getPages());
```

```
        System.out.println("一共多少数据 ==> " + iPage.getTotal());
        //这里是获取所有的数据源
        System.out.println("数据 ==> " + iPage.getRecords());


    }

}
```

设置拦截器

```
package com.itheima.config;

import com.baomidou.mybatisplus.extension.plugins.MybatisPlusInterceptor;
import
com.baomidou.mybatisplus.extension.plugins.inner.PaginationInnerInterceptor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * @ClassName MybatisPlusConfig
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/7 10:47
 * @Version 1.0
 */

@Configuration
public class MybatisPlusConfig {

    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor(){

        //创建一个拦截器的 对象
        MybatisPlusInterceptor mybatisPlusInterceptor = new
MybatisPlusInterceptor();

        //增加具体的拦截器 == 》 分页拦截器
        mybatisPlusInterceptor.addInnerInterceptor(new
PaginationInnerInterceptor());

        return mybatisPlusInterceptor;

    }

}
```



4 DQL编程控制

1 条件查询

查询小于

```
@Autowired
private UserDao userDao;

@Test
public void TestGetAll(){

    // 按条件查询操作
    QueryWrapper queryWrapper = new QueryWrapper();
    // 查询大于小于操作

    //小于 &lt
    // 查询 id < 10的
    queryWrapper.lt("id",10);

    List<Book> books = userDao.selectList(queryWrapper);
    System.out.println(books);

}
```

另外一种书写形式

```
@Test
public void TestLambda(){
```

```
    //    查询方式二 按照lambda表达式查询

    QueryWrapper<Book> qw = new QueryWrapper<Book>();

    qw.lambda().lt(Book::getId,10);


    List<Book> books = userDao.selectList(qw);

    System.out.println(books);


}
```

多条件查询

```
@Test
public void TestLambdaQW(){

    //    查询方式三 按照LambdaQueryWrapper表达式查询

    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();

    lqw.lt(Book::getId,12);

    // 多条件查询

    lqw.gt(Book::getId,10);

    List<Book> books = userDao.selectList(lqw);

    System.out.println(books);


}
```

lambda链式编程

```
@Test
public void TestLambdaQW_lanshi(){

    //    查询方式三 按照LambdaQueryWrapper表达式查询

    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();

    // 多条件查询

    lqw.gt(Book::getId,10).lt(Book::getId,12);

    List<Book> books = userDao.selectList(lqw);

    System.out.println(books);


}
```

空值判断

```
@Test
public  void TestNull(){

    //创建一个 UserQuery对象
    UserQuery uq = new UserQuery();

    // 将对象的值进行赋值
    uq.setId2(10);
    uq.setId(5);

    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();

    // 进行判断
    if(null != uq.getId2()){
        lqw.lt(Book::getId,uq.getId2());
    }

    if(null != uq.getId()){
        lqw.gt(Book::getId,uq.getId());
    }

    //输出结果
    List<Book> books = userDao.selectList(lqw);

    System.out.println(books);


}
```

```
package com.itheima;

import com.baomidou.mybatisplus.core.conditions.query.LambdaQueryWrapper;
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.baomidou.mybatisplus.core.metadata.IPage;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.itheima.dao.UserDao;
import com.itheima.domain.Book;
import com.itheima.query.UserQuery;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.List;

@SpringBootTest
class MybatisPlus01ApplicationTests {

    @Autowired
    private UserDao userDao;

    @Test
    public void TestGetAll(){

        // 按条件查询操作
```

```
        QueryWrapper queryWrapper = new QueryWrapper();
        // 查询大于小于操作

        //小于 &lt
        // 查询 id < 10的
        queryWrapper.lt("id",10);

        List<Book> books = userDao.selectList(queryWrapper);
        System.out.println(books);



    }

    @Test
    public void TestLambda(){

        //    查询方式二 按照lambda表达式查询

        QueryWrapper<Book> qw = new QueryWrapper<Book>();

        qw.lambda().lt(Book::getId,10);


        List<Book> books = userDao.selectList(qw);

        System.out.println(books);


    }

    @Test
    public void TestLambdaQW(){

        //    查询方式三 按照LambdaQueryWrapper表达式查询

        LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();

        lqw.lt(Book::getId,12);

        // 多条件查询

        lqw.gt(Book::getId,10);

        List<Book> books = userDao.selectList(lqw);

        System.out.println(books);


    }

    @Test
    public void TestLambdaQW_lanshi(){

        //    查询方式三 按照LambdaQueryWrapper表达式查询

        LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();
```

```
        // 多条件查询

        lqw.gt(Book::getId,10).lt(Book::getId,12);

        List<Book> books = userDao.selectList(lqw);

        System.out.println(books);


    }

    @Test
    public  void TestNull(){

        //创建一个 UserQuery对象
        UserQuery uq = new UserQuery();

        // 将对象的值进行赋值
        uq.setId2(10);
        uq.setId(5);

        LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();

        // 进行判断
        if(null != uq.getId2()){
            lqw.lt(Book::getId,uq.getId2());
        }

        if(null != uq.getId()){
            lqw.gt(Book::getId,uq.getId());
        }

        //输出结果
        List<Book> books = userDao.selectList(lqw);

        System.out.println(books);


    }

    @Test
    public  void TestNullLambda(){

        //创建一个 UserQuery对象
        UserQuery uq = new UserQuery();

        // 将对象的值进行赋值
        uq.setId2(10);
        uq.setId(5);

        LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();

        lqw.lt(null != uq.getId2(),Book::getId,uq.getId2());
        lqw.gt(null != uq.getId(),Book::getId,uq.getId());

        List<Book> books = userDao.selectList(lqw);

        System.out.println(books);
```

```
    }

    @Test
    public  void TestNullLambdaLanshi(){

        //创建一个 UserQuery对象
        UserQuery uq = new UserQuery();

        // 将对象的值进行赋值
        uq.setId2(10);
        uq.setId(5);

        LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();

        lqw.lt(null != uq.getId2(),Book::getId,uq.getId2())
           .gt(null != uq.getId(),Book::getId,uq.getId());

        List<Book> books = userDao.selectList(lqw);

        System.out.println(books);
    }
}
```

2 投影查询

通过投影的方法选着查看的字段

```
lqw.select(Book::getId,Book::getName);
```

```
@Autowired
private UserDao userDao;

@Test
public void TestGet01(){

    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();

    //使用投影 显示结果
    lqw.select(Book::getId,Book::getName);

    List<Book> books = userDao.selectList(lqw);

    System.out.println(books);


}
```

以Map形式查询

```
@Test
public void TestGet03(){
```

```
    QueryWrapper<Book> qw = new QueryWrapper<Book>();
    //使用Map 体现 select

    qw.select("count(*) as number");

    List<Map<String, Object>> maps = userDao.selectMaps(qw);

    System.out.println(maps);


}
```

分组查询

```
@Test
public void TestGet04(){

    QueryWrapper<Book> qw = new QueryWrapper<Book>();
    // 根据 name 进行分组

    qw.select("count(*) as number","name");
    qw.groupBy("name");

    // List<Book> books = userDao.selectList(qw);

    List<Map<String, Object>> maps = userDao.selectMaps(qw);

    System.out.println(maps);

}
```

```
package com.itheima;

import com.baomidou.mybatisplus.core.conditions.query.LambdaQueryWrapper;
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.baomidou.mybatisplus.core.metadata.IPage;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.itheima.dao.UserDao;
import com.itheima.domain.Book;
import com.itheima.query.UserQuery;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.List;
import java.util.Map;

@SpringBootTest
class MybatisPlus01ApplicationTests {

    @Autowired
    private UserDao userDao;
```

```java
@Test
public void TestGet01(){

    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();

    //使用投影 显示结果
    lqw.select(Book::getId,Book::getName);

    List<Book> books = userDao.selectList(lqw);

    System.out.println(books);


}

@Test
public void TestGet02(){

    QueryWrapper<Book> qw = new QueryWrapper<Book>();
    //使用投影 显示结果
    qw.select("id","name");

    List<Book> books = userDao.selectList(qw);

    System.out.println(books);


}

@Test
public void TestGet03(){

    QueryWrapper<Book> qw = new QueryWrapper<Book>();
    //使用Map 体现 select

    qw.select("count(*) as number");

    List<Map<String, Object>> maps = userDao.selectMaps(qw);

    System.out.println(maps);


}


@Test
public void TestGet04(){

    QueryWrapper<Book> qw = new QueryWrapper<Book>();
    // 根据 name 进行分组

    qw.select("count(*) as number","name");
    qw.groupBy("name");

    // List<Book> books = userDao.selectList(qw);

    List<Map<String, Object>> maps = userDao.selectMaps(qw);
```

```
        System.out.println(maps);

    }

}
```

3 查询条件

之前我们演示了 lt() gt() 方法

其他一些方法可以查看网站

MyBatis-Plus (baomidou.com)

简单演示几个函

eq

```
/**
 * 等于操作 查询
 * lqw.eq
 */

@Test
public void TestGet01(){

    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();

    lqw.eq(Book::getId,"10").eq(Book::getName,"市场营销");

    Book book = userDao.selectOne(lqw);

    System.out.println(book);

}
```

between

```
/**
 * 范围查询
 * between
 *
 */

@Test
public void TestGet02(){

   LambdaQueryWrapper<Book> lwq = new LambdaQueryWrapper<Book>();

   lwq.between(Book::getId,2,10);

    List<Book> books = userDao.selectList(lwq);

    System.out.println(books);

}
```

模糊匹配like

```
/**
 * 范围查询
 * 模糊匹配like
 *
 */

@Test
public void TestGet03(){

    LambdaQueryWrapper<Book> lwq = new LambdaQueryWrapper<Book>();

    lwq.like(Book::getName,"论");

    List<Book> books = userDao.selectList(lwq);

    System.out.println(books);

}
```

MybatisPlus还有一般内容没有学习 到这里 先停止学习 等做项目的时候在便学期

JDBC

JDBC创建

1.注册驱动

```
//        1. 注册驱动
        Class.forName("com.mysql.jdbc.Driver");
```

2.获得连接

```
String url = "jdbc:mysql://localhost:3306/databases_01";
String username = "root";
String password = "admin123";

Connection coon = DriverManager.getConnection(url,username,password);
```

3.定义SQL语句

```
 String sql = "UPDATE USER SET NAME = 'jac'  WHERE id = 2";
```

4.获取SQL对象

```
Statement stmt = coon.createStatement();
```

5.执行SQL

```
int count = stmt.executeUpdate(sql);
```

6.处理结果

```
System.out.println(count);
```

7.释放资源

```
stmt.close();
coon.close();
```

JDBC的API讲解

ConnectionAPI讲解

数据库事务 利用抛出异常执行多条sql语句

```
//          开启事务

        try{

            coon.setAutoCommit(false);
//         5.执行sql
            int count1 = stmt.executeUpdate(sql1);
//          6.处理的结果
            System.out.println(count1 + "count1");
//         5.执行sql
            int count2 = stmt.executeUpdate(sql2);
//          6.处理的结果
            System.out.println(count2 + "count2");

            coon.commit();

        } catch (Exception e){

//              回滚事务 代码中任何地方出现异常就回滚到最初地方
            coon.rollback();
            e.printStackTrace();
        }
```

StatementAPI注解

DML 执行增删改查功能

修改功能

```
package com.itheima.JDBC;

import org.testng.annotations.Test;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;

/**
 * JDBC 快速入门
 */

public class JDBCDemo4_Statement {

    @Test
    public void testDML() throws Exception {

        Class.forName("com.mysql.jdbc.Driver");

        String url = "jdbc:mysql://localhost:3306/databases_01";

        String username = "root";
        String password = "admin123";

        Connection coon = DriverManager.getConnection(url,username,password);

        String sql1 = "UPDATE USER SET NAME = 'j'  WHERE id = 1";

        Statement stmt = coon.createStatement();

        int count = stmt.executeUpdate(sql1);

        if(count > 0) {
            System.out.println("修改成功");
        }else {
            System.out.println("修改失败");
        }

        System.out.println(count);

        stmt.close();

        coon.close();
    }
}
```

DDL

创建一个新的数据库 删除数据库

```
package com.itheima.JDBC;

import org.testng.annotations.Test;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;

/**
 * JDBC 快速入门
 */

public class JDBCDemo4_Statement {

    @Test
    public void testDML() throws Exception {

        Class.forName("com.mysql.jdbc.Driver");

        String url = "jdbc:mysql://localhost:3306/databases_01";

        String username = "root";
        String password = "admin123";

        Connection coon = DriverManager.getConnection(url,username,password);

//        执行DML操作
//        String sql1 = "UPDATE USER SET NAME = 'j'  WHERE id = 1";

//        执行DDL操作 创建一个数据库
        String sql1 = "create database ty";

        Statement stmt = coon.createStatement();

        int count = stmt.executeUpdate(sql1);

        if(count > 0) {
            System.out.println("修改成功");
        }else {
            System.out.println("修改失败");
        }

        stmt.close();

        coon.close();
    }
}
```

ResultSet注解

DQL查询

使用executeQuery执行查询sql语句 返回一个ResultSet对象

```
   ResultSet rs = stmt.executeQuery(sql);
```

ResultSet中的next()

进行向下检索 类型于 MySQL中的 游标

```
        while(rs.next()){

            user us = new user();

            int id = rs.getInt("id");
            String name = rs.getString("name");
            int age = rs.getInt("age");
            int status = rs.getInt("status");
            String gender = rs.getString("gender");

            us.setId(id);
            us.setName(name);
            us.setAge(age);
            us.setGender(gender);
            us.setStatus(status);


            list.add(us);

        }
```

ResultSet 中的 get和set方法

```
int id = rs.getInt("id");
String name = rs.getString("name");
int age = rs.getInt("age");
int status = rs.getInt("status");
String gender = rs.getString("gender");

us.setId(id);
us.setName(name);
us.setAge(age);
us.setGender(gender);
us.setStatus(status);
```

创建user类

```
package com.itheima.pojo;

public class user {

    int id;
    String name;
    int age;
```

```
    int status;
    String gender;

    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int getStatus() {
        return status;
    }

    public void setStatus(int status) {
        this.status = status;
    }

    public String getGender() {
        return gender;
    }

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

    @Override
    public String toString() {
        return "user{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                ", status=" + status +
                ", gender='" + gender + '\'' +
                '}';
    }
}
```

结合

创建一个列表存储user对象

```
  List<user> list = new ArrayList<>();
```

创建一个user对象存储rs对象

```
user us = new user();

int id = rs.getInt("id");
String name = rs.getString("name");
int age = rs.getInt("age");
int status = rs.getInt("status");
String gender = rs.getString("gender");

us.setId(id);
us.setName(name);
us.setAge(age);
us.setGender(gender);
us.setStatus(status);
```

将 user对象加到 list对象中

```
list.add(us);
```

PreparedStatement

PrepareStatement对象

```
PreparedStatement pstmt =  coon.prepareStatement(sql);
```

pstmt方法

```
pstmt.setInt(1,id);
pstmt.setString(2,name);
```

pstmt执行SQL

```
//执行SQL

ResultSet rs = pstmt.executeQuery();

if(rs.next()){
    System.out.println("成功");
}else{
    System.out.println("失败");
}
```

释放资源

```
//       进行释放资源
rs.close();
pstmt.close();
coon.close();
```

PrepareStatement好处

1 预编译SQL，性能更高

1.开启预编译代码

```
useServerPrepStmts=true
```

2.配置mysql配置文件（my.ini）

```
log-output=FILE
general-log=1
general_log_file="D:\mysql.log"
slow-query-log=1
slow_query_log_file="D:\mysql_slow.log"
long_query_time=2
```

2 防止SQL注入

数据库连接池

MyBatis

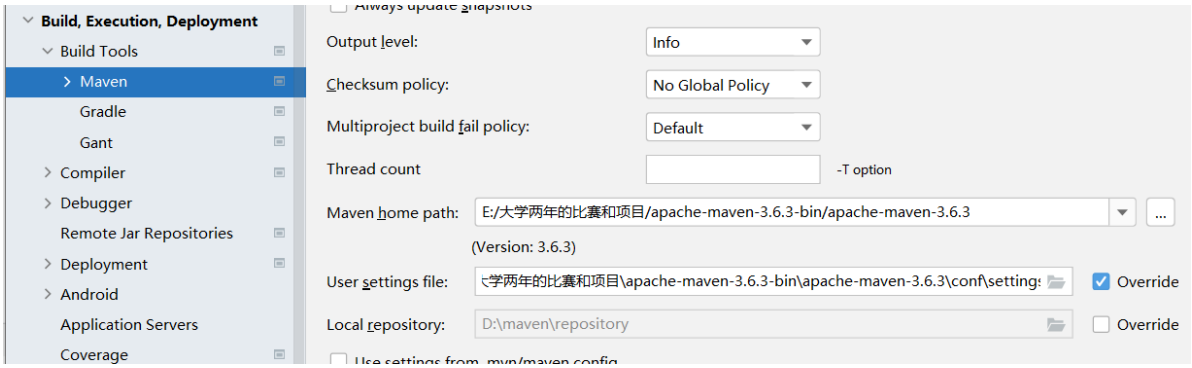
是持久性框架 简化JDBC开发的

导入MyBatis依赖

写入依赖

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

配置对应Maven home path文件



数据库连接过程中字符编码出现问题

错误提示

```
Unknown initial character set index '255' received from server. Initial client
character set can be forced via the 'characterEncoding' property.
```

最初代码

```
jdbc:mysql://localhost:3306/mybatis
```

修改之后的代码

```
jdbc:mysql://localhost:3306/mybatis?useUnicode=true&amp;characterEncoding=utf8
```

导入logback配置文件

在main的resources文件中加入logback.xml文件 内容如下

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <!--
        CONSOLE ： 表示当前的日志信息是可以输出到控制台的。
    -->
    <appender name="Console" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>[%level] %blue(%d{HH:mm:ss.SSS}) %cyan([%thread])
%boldGreen(%logger{15}) - %msg %n</pattern>
        </encoder>
    </appender>

    <logger name="com.itheima" level="DEBUG" additivity="false">
        <appender-ref ref="Console"/>
    </logger>


    <!--

      level:用来设置打印级别，大小写无关：TRACE, DEBUG, INFO, WARN, ERROR, ALL 和 OFF
     ，  默认debug
      <root>可以包含零个或多个<appender-ref>元素，标识这个输出位置将会被本日志级别控制。
      -->
```

```
    <root level="DEBUG">
        <appender-ref ref="Console"/>
    </root>
</configuration>
```

创建mybatis-config.xml文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <!--数据库链接信息-->
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/mybatis?
useUnicode=true&amp;characterEncoding=utf8"/>
                <property name="username" value="root"/>
                <property name="password" value="admin123"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
<!--加载sql映射文件-->
        <mapper resource="com/itheima/Mapper/UserMapper.xml"/>
    </mappers>
</configuration>
```

编写SQL映射文件 统一管理文件 解决硬编码问题

在resources文件下创建UserMapper.xml

```
<?xml version="1.0" encoding="UTF-8" ?><!DOCTYPE mapper PUBLIC
"-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-
mapper.dtd">
<mapper namespace="test">
    <select id="selectAll" resultType="com.itheima.pojo.User">
        select * from tb_user;
    </select>
</mapper>
```

编码

在com.itheima.pojo包下创建User类

```
package com.itheima.pojo;

public class User {

    private Integer id          ;
```

```
    private String username    ;
    private String PASSWORD    ;
    private String gender      ;
    private String addr        ;

    public Integer getId() {
        return id;
    }

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

    public String getUsername() {
        return username;
    }

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

    public String getPASSWORD() {
        return PASSWORD;
    }

    public void setPASSWORD(String PASSWORD) {
        this.PASSWORD = PASSWORD;
    }

    public String getGender() {
        return gender;
    }

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

    public String getAddr() {
        return addr;
    }

    public void setAddr(String addr) {
        this.addr = addr;
    }


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

测试代码

在com.itheima中编写测试代码

```
package com.itheima;

import com.itheima.pojo.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.IOException;
import java.io.InputStream;
import java.util.List;

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

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

//        1. 加载mybatis的核心配置文件，获取SqlSessionFactory

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

//        2 获取SqlSession对象 用对象执行Sql
        SqlSession sqlSession =  sqlSessionFactory.openSession();

//        3 执行sql语句 执行多条select

        // 返回一个泛型 List<User>
        List<User> users = sqlSession.selectList("test.selectAll");

//        4 执行单条sql
//        sqlSession.selectOne()


//        打印输出列表

        System.out.println(users);

//        释放资源
        sqlSession.close();

    }
}
```

Mapper代理开发

1 创建UserMapper接口

在com.itheima.Mapper中创建UserMapper接口

2 UserMapper的接口和xml文件路径要一样

将xml文件放到resources对应的com/itheima/Mapper文件目录下 切记这里使用 / 而不是 .

3 修改Mapper的namespace为接口

接口名：com.itheima.Mapper.UserMapper

4 定义接口中的方法于id名一致

定义一个selectAll的方法

resultType的返回类型就是com.itheima.pojo.User

5 代理对象完成之后 测试代码

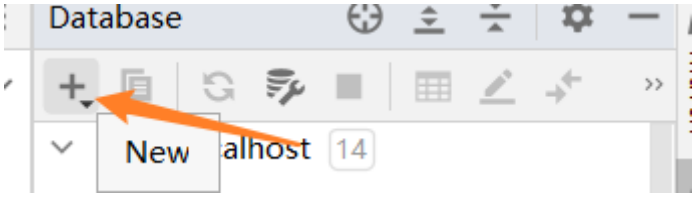
```
package com.itheima;
import com.itheima.Mapper.UserMapper;
import com.itheima.pojo.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;
import java.util.List;

/**
 * MyBatis 快速入门代码
 *
 */
public class MyBatisDemo2 {

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

//        1. 加载mybatis的核心配置文件，获取SqlSessionFactory

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

//        2 获取SqlSession对象 用对象执行Sql
        SqlSession sqlSession =  sqlSessionFactory.openSession();

//        获取接口代理对象

       UserMapper userMapper =  sqlSession.getMapper(UserMapper.class);

//        3 执行sql语句 执行多条select

        // 返回一个泛型 List<User>
```

```
        List<User> users = userMapper.selectAll();

//          打印输出列表

        System.out.println(users);

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

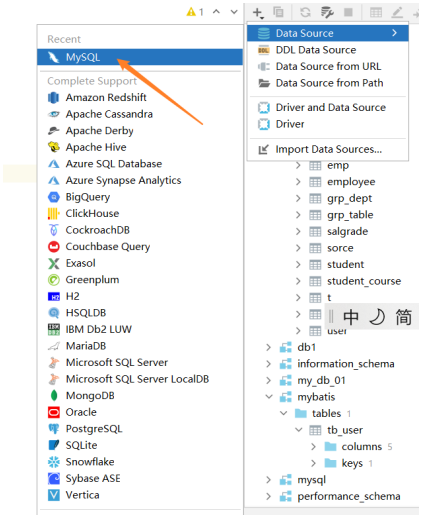
Mysql在IDEA中显示

1 点击Database

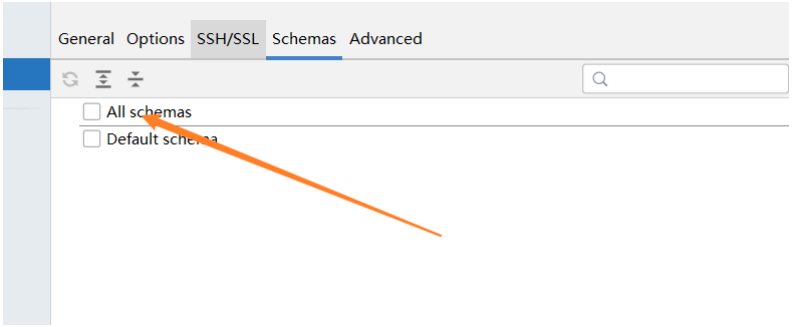
2 点击 +



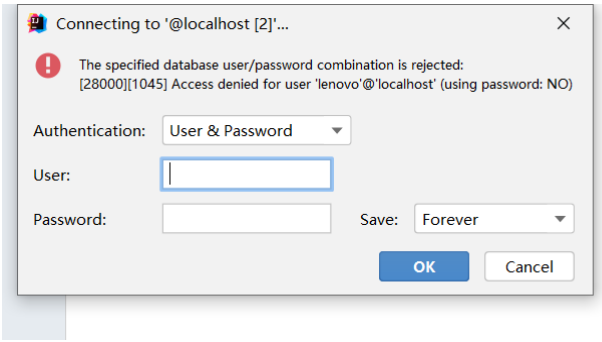
3 选择Mysql数据库



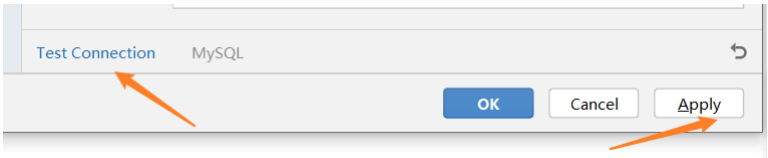
4 勾选所有是数据库



5 输入 用户名和密码



6 点击Test Connection 和 Apply



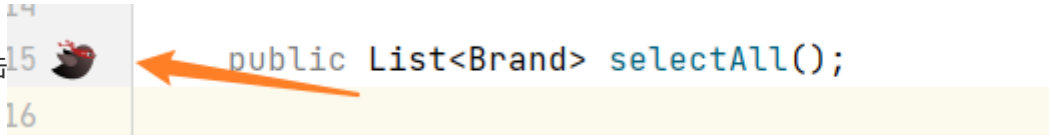
Mybatis的增删改查功能

查询所有数据selectAll

1 在接口中定义方法selectAll

```
public List<Brand> selectAll()
```

点击



生成对应的select语句

```
<select id="selectAll" resultType="Brand">
#         select * from tb_brand;
select
```

2 在test文件中写入对应的测试方法

```
    @Test

    public  void testSelectAll() throws Exception {
```

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

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

//        3 获取Mapper代理对象

        BrandMapper brandMapper = sqlSession.getMapper(BrandMapper.class);

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

//        5 输出结果

        System.out.println(brands);

//        6 关闭资源

        sqlSession.close();


    }
```

3 简化sql语句

每次生成一个查询方法对需要写对应的select过于繁琐

1 使用sql语句块

```
<sql id = "brand_colunm">
    id, brand_name brandName, company_name companyName, ordered, description,
status
</sql>
```

```
<select id="selectAll" resultType="Brand">
    select
      <include refid="brand_colunm"></include>  // 代替 *
    from tb_brand;
</select>
```

但是sql语句块不灵活每次更换select里面的内容需要更换一个sql文件

2 使用resultMap 解决

```
<resultMap id="brandResultMap" type="brand">
    <!--         有两种属性 -->
    <!--
        1 id 主属性
        2 result  非主属性
    -->
    <result column="brand_name" property="brandName"></result>
    <result column="company_name" property="companyName"></result>

</resultMap>
```

1.id 属性： 是唯一标示

2 type属性： 是和接口类名一致

result属性中 有两种： 1 id 主属性 2 result 非主属性

使用的标签是： column 表示列名 property 表示替换名

3 参数占位符

```
1 #{}: 会将其退换成?,防止SQL注入
2 ${}: 拼SQL,会存在SQL注入问题

参数类型 parameterType 可以省略

特数字符处理
    1 转义处理 比如 < ==> &lt
    2 CDARA区：   处理  <![CDATA[
                            <
                        ]]>
```

参数传递问题

使用@Param

1 使用 @Param("参数名称") 标记每一个参数，在映射配置文件中就需要使用 #{参数名称} 进行占位

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

2 在xml内编写SQL语句

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

3 在测试代码中实现

```
@Test

    public  void selectByCondition() throws Exception {

//        参数设定

        int status = 1;
        String companyName = "华为";
        String brandName = "华为";

//         参数处理 将 company_name和 brand_name 进行模糊处理

        companyName = "%" + companyName + "%";
        brandName = "%" + brandName + "%";


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

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

//        3 获取Mapper代理对象

        BrandMapper brandMapper = sqlSession.getMapper(BrandMapper.class);

//        4. 执行方法
//         1 使用传参方法进行
        List<Brand> brands =
brandMapper.selectByCondition(status,companyName,brandName);

//         5 输出结果

        System.out.println(brands);

//        6 关闭资源

        sqlSession.close();


    }
```

2 多个参数封装成实体对象

1 创建一个对象接口

```
List<Brand> selectByCondition(Brand brand);
```

2 编写sql语句

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

3 编写测试方法

```
 @Test
    /**
     *  将多个参数封装成实体对象
     */
    public  void selectByCondition_Obj() throws Exception {

//        参数设定

        int status = 1;
        String companyName = "华为";
        String brandName = "华为";

//         参数处理 将 company_name和 brand_name 进行模糊处理

        companyName = "%" + companyName + "%";
        brandName = "%" + brandName + "%";


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

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

//        3 获取Mapper代理对象

        BrandMapper brandMapper = sqlSession.getMapper(BrandMapper.class);

        Brand brand = new Brand();

        /**
         * 封装成对象形式
         */
        brand.setStatus(status);
        brand.setBrandName(brandName);
        brand.setCompanyName(companyName);

//        4. 执行方法
//         List<Brand> brands =
brandMapper.selectByCondition(status,companyName,brandName);
        List<Brand> brands = brandMapper.selectByCondition(brand);
 //         5 输出结果

        System.out.println(brands);
```

```
//        6 关闭资源

        sqlSession.close();


    }
```

3 使用Map

1 创建map为参数的方法

```
List<Brand> selectByCondition(Map map);
```

2 编写sql语句

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

3 编写对象的test方法

```
@Test
    /**
     *  使用Map作为接口封装对象
     */
    public  void selectByCondition_Map() throws Exception {

//        参数设定

        int status = 1;
        String companyName = "华为";
        String brandName = "华为";

//         参数处理 将 company_name和 brand_name 进行模糊处理

        companyName = "%" + companyName + "%";
        brandName = "%" + brandName + "%";


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

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

//        3 获取Mapper代理对象

        BrandMapper brandMapper = sqlSession.getMapper(BrandMapper.class);

        /**
```

```
         * 封装成Map对象形式
         */

        Map map = new HashMap();

        map.put("status",status);
        map.put("companyName",companyName);
        map.put("brandName",brandName);

//        4. 执行方法
//         List<Brand> brands =
brandMapper.selectByCondition(status,companyName,brandName);
        List<Brand> brands = brandMapper.selectByCondition(map);
        //        5 输出结果

        System.out.println(brands);

//        6 关闭资源

        sqlSession.close();


    }
```

动态SQL

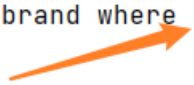
使用if标签进行判断

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

存在的问题

当将status这个条件进行去除 根据xml中SQL中的语法可以知道 where后面跟一个and 出现异常

```
==>  Preparing: select * from tb_brand where and company_name like ?
==> Parameters: %华为%(String)
```



解决方法

1 改写SQL语句

```
<select id="selectByCondition" resultMap="brandResultMap">
    select *  from tb_brand  where 1 = 1
        <if test="status != null">
            and status = #{status}
        </if>
        <if test="companyName != null and companyName != '' ">
            and company_name like #{companyName}
        </if>
        <if test="brandName != null and brandName != '' ">
            and brand_name like #{brandName}
        </if>
</select>
```

2 使用where标签替代原来的where

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

添加数据

1 在接口中创建添加函数

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

2 在xml文件中加入对应的SQl语句

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

3 在test中加入测试函数

```
    @Test
    /**
     *  add方法
     *
     */
```

```
    public  void add() throws Exception {

//        参数设定

        int status = 1;
        String companyName = "波导手机";
        String brandName = "波导手机";
        String description = "手机中的战斗机";
        int ordered = 100;

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

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

//        3 获取Mapper代理对象

        BrandMapper brandMapper = sqlSession.getMapper(BrandMapper.class);

       Brand brand = new Brand();
       brand.setStatus(status);
       brand.setCompanyName(companyName);
       brand.setBrandName(brandName);
       brand.setDescription(description);
       brand.setOrdered(ordered);

//        4. 执行方法
       brandMapper.add(brand);

//        5. 手动提交事务
//         sqlSession.commit();

//        6 关闭资源

        sqlSession.close();
    }
```

4 添加主键

useGeneratedKeys：是够获取自动增长的主键值。true表示获取

keyProperty ： 指定将获取到的主键值封装到哪儿个属性里

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

5 开始事务

1 openSession开始事务

```
SqlSession sqlSession = sqlSessionFactory.openSession(true);
```

2 手动提交

```
 sqlSession.commit();
```

删除数据

1 单个删除

1. 编写接口方法

```
public void delete(int id);
```

2.编写SQL语句

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

3. 编写测试方法

注意不提交事务数据库查询结果查询不到

提交事务的说明和上次一样

```
    @Test
    /**
     *  add方法
     *
     */

    public  void delete() throws Exception {

//        参数设定
        int id = 6;

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

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

//        3 获取Mapper代理对象

        BrandMapper brandMapper = sqlSession.getMapper(BrandMapper.class);
```

```
//        4. 执行方法

        brandMapper.delete(id);

//        5. 手动提交事务
        sqlSession.commit();

//        6 关闭资源

        sqlSession.close();
    }
```

2 多数删除

1 创建一个接口函数

```
public void deleteByids(@Param("ids") int[] ids);
```

2 编写对应的SQL语句

```
<delete id="deleteByids">
    delete from tb_brand where id
         in

    <foreach collection="ids" item = "id" separator="," open="(" close=")">
         #{id}
    </foreach>
</delete>
```

3 对应的test的方法

```
@Test
    /**
     *  deleteByids方法
     *
     */

    public  void deleteByids() throws Exception {

//        参数设定
        int []ids = {3,5};

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

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

//        3 获取Mapper代理对象

        BrandMapper brandMapper = sqlSession.getMapper(BrandMapper.class);
```

```
//        4. 执行方法

        brandMapper.deleteByids(ids);

//        5. 手动提交事务
        sqlSession.commit();

//        6 关闭资源

        sqlSession.close();
    }
```

参数传递问题

单个参数传递问题

1 POJO类型

直接使用。要求 属性名 和 参数占位符名称 一致

2 Map类型

直接使用。要求 map集合的键名 和 参数占位符名称 一致

3 collection集合类型

1 创建一个Collection的对象的函数

```
User select (Collection collection);
```

2 创建出Hashset() 对象

```
User user = userMapper.select(new HashSet());
```

3 进入ParamNameResolver

对象是collection类型的时候 map.put 会调用两次

第一个 是 map.put("collection",collection)

map.put("name",collection) ==> name = arg0

```
public static Object wrapToMapIfCollection(Object object, String actualParamName) {
    ParamMap map;
    if (object instanceof Collection) {
        map = new ParamMap();
        map.put("collection", object);
        if (object instanceof List) {
            map.put("list", object);
        }

        Optional.ofNullable(actualParamName).ifPresent( consumer: (name) -> {
            map.put(name, object);
        });
        return map;
```

4 List集合类型

1 创建一个List的对象的函数

```
User select (List list);
```

2 创建出Arraylist() 对象

```
User user = userMapper.select(new Arraylist());
```

3 进入ParamNameResolver

对象是List类型的时候 map.put 会调用两次

第一个 是 map.put("collection",collection)

map.put("list",collection)

map.put("name",collection) ==> name = arg0

```
public static Object wrapToMapIfCollection(Object object, String actualParamName) {
    ParamMap map;
    if (object instanceof Collection) {
        map = new ParamMap();
        map.put("collection", object);
        if (object instanceof List) {
            map.put("list", object);
        }

        Optional.ofNullable(actualParamName).ifPresent( consumer: (name) -> {
            map.put(name, object);
        });
        return map;
    } else if (object != null && object.getClass().isArray()) {
        map = new ParamMap();
        map.put("array", object);
        Optional.ofNullable(actualParamName).ifPresent( consumer: (name) -> {
            map.put(name, object);
        });
        return map;
    } else {
        return object;
    }
}
}
```


5 Array类型

1 创建一个Array的对象的函数

```
User select (Array array);
```

2 创建出Array() 对象

```
User user = userMapper.select(new array());
```

3 进入ParamNameResolver

对象是Array类型的时候 map.put 会调用两次

第一个 是 map.put("arg0"，数组)

map.put("array"，数组);

6 其他类型

多个参数传递问题

1 生成对应接口函数

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

当mybatis发现传递参数是多个的时候就会会将这些参数自动封装成Map集合，构造成对应的键值对

内部会自动使用

map.put("arg0",参数值1)

map.put("param1",参数值1)

map.put("arg1",参数值2)

map.put("param2",参数值2)

2 查询的时候编写SQL语句

```
    <select id="select" resultMap="UserresultMapper">
        select * from tb_user where
            username = #{username}
        and password = #{password}
    </select>
```

3 当没有使用@Param注注解

```
    <select id="select" resultMap="UserresultMapper">
        select * from tb_user where
            username = #{arg0}
        and password = #{arg01}
    </select>
```

```
    <select id="select" resultMap="UserresultMapper">
        select * from tb_user where
            username = #{param1}
        and password = #{parma2}
    </select>
```

4 推荐使用@param注解 去替代Map中的键名

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

  <select id="select" resultMap="UserresultMapper">
        select * from tb_user where
            username = #{username}
        and password = #{password}
    </select>
```

使用注解的方法

当sql语句比较简单的时候使用注解

```
    @Select("select * from tb_brand where addr =  #{addr}")
    User selectByAddr(String addr);
```

当sql语句比较复杂的时候使用xml文件配置

注解类型

1查询 @Select

2 插入 @Insert

3 更新 @Update

4 删除 @Delete

Mybatis总结

创建文件summary_Mybatis

创建main中的java里面的文件

1 创建一个类User

2 创建对应的接口UserMapper

3 配置pom文件各种依赖

配置依赖的时候去Maven官网搜索相对应的依赖

4 修改对应的文件

5 在resource文件中配置好mybatis_config.xml 和 logoback.xml

6 连接数据库

7 在Maven中进行Compile 和 Test

8 在resource文件中创建和UserMapper对应的xml文件

至此基本文件框架写完了

代码书写

不使用Mapper

1 在 UserMapper 中创建一个方法

```
   public User selectById(int id);
```

2 使用resultType类型书SQL

```
    <select id="selectAll" resultType="user">
        select * from tb_user;
    </select>
```

3 编写测试代码

```
    //  1 加载文件获取SqlsessionFactory

        String resource = "mybatis_config.xml";

        InputStream inputStream = Resources.getResourceAsStream(resource);

        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

    //    2 获取SqlSession

        SqlSession sqlsession = sqlSessionFactory.openSession();

    //    3 执行SQL语句
        List<User> user = sqlsession.selectList("selectAll");

        System.out.println(user);
```

```
        sqlsession.close();
```

使用Mapper映射文件

首先看个错误提示：没有找到映射文件

```
Type interface com.itheima.Mapper.UserMapper_1 is not known to the
MapperRegistry.
```

配置映射文件步骤

1 在MyBatis_config.xml中进行配配置

```
  <mapper resource="com/itheima/Mapper/BrandMapper.xml"/>
```

2 在对应接口的Mapper中配置

namespace要和路径保持一致

```
<mapper namespace="com.itheima.Mapper.UserMapper_1">

</mapper>
```

在使用Mapper进行测试

重新创建一个接口

```
User selectById(int id);
```

对应的Mapper文件中书写SQL

```
 <select id="selectById" resultType="User">
        select * from tb_user where id = #{id};
 </select>
```

编写测试代码

获取Mapper映射对象 之后就不需要写入id名

统一使用UserMapper_1.class

```
         // 纪录文件
        String resource = "mybatis_config.xml";

        // 加载文件 生成SqlSessionFactory对象
        InputStream inputStream = Resources.getResourceAsStream(resource);

        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

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

        // 获取Mapper映射对象
```

```
        UserMapper_1 userMapper_1 = sqlSession.getMapper(UserMapper_1.class);

        //  执行sql
        User user = userMapper_1.selectById(2);

        //  打印输出
        System.out.println(user);

        //  释放资源
        sqlSession.close();
```

通过在mybatis_config.xml设置别名

```
<typeAliases>
    <package name="com.itheima.pojo"/>
</typeAliases>
```

可以将 com.itheima.pojo.User 简化

```
resultType="User"
```

至此Mapper基本功能就讲述完成

之后的增删改查功能就后续项目实战复习

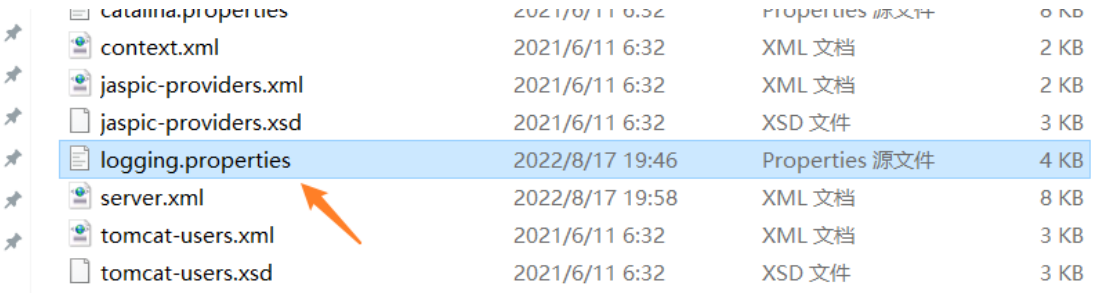
TomCat

1 解压压缩包

2 修改conf文件中的server.xml中默认端口

```
<Connector port="8081" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />
<!-- A "Connector" using the shared thread pool-->
<!--
<Connector executor="tomcatThreadPool"
           port="8080" protocol="HTTP/1.1"
```

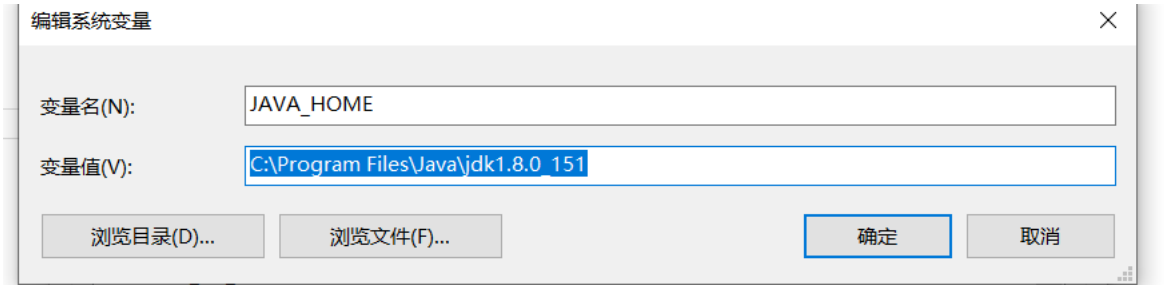
3 解决乱码问题



```
44
45  java.util.logging.ConsoleHandler.level = FINE
46  java.util.logging.ConsoleHandler.formatter = org.apache.juli.OneLineFormatter
47  java.util.logging.ConsoleHandler.encoding = GBK
48
```

4 配置环境

如果出现闪退的情况可能是出现了 java配配置环境出现了错误



C:\Program Files\Java\jdk1.8.0_151

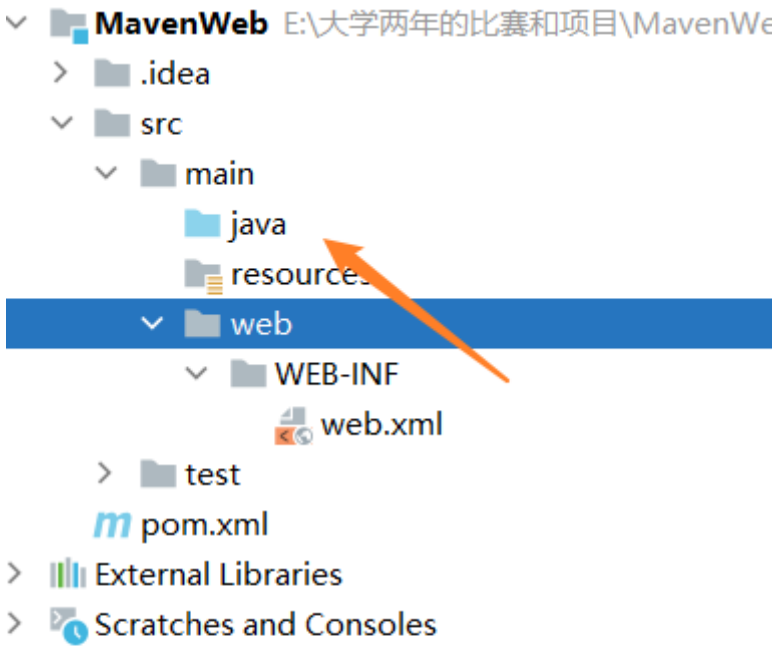
5 部署

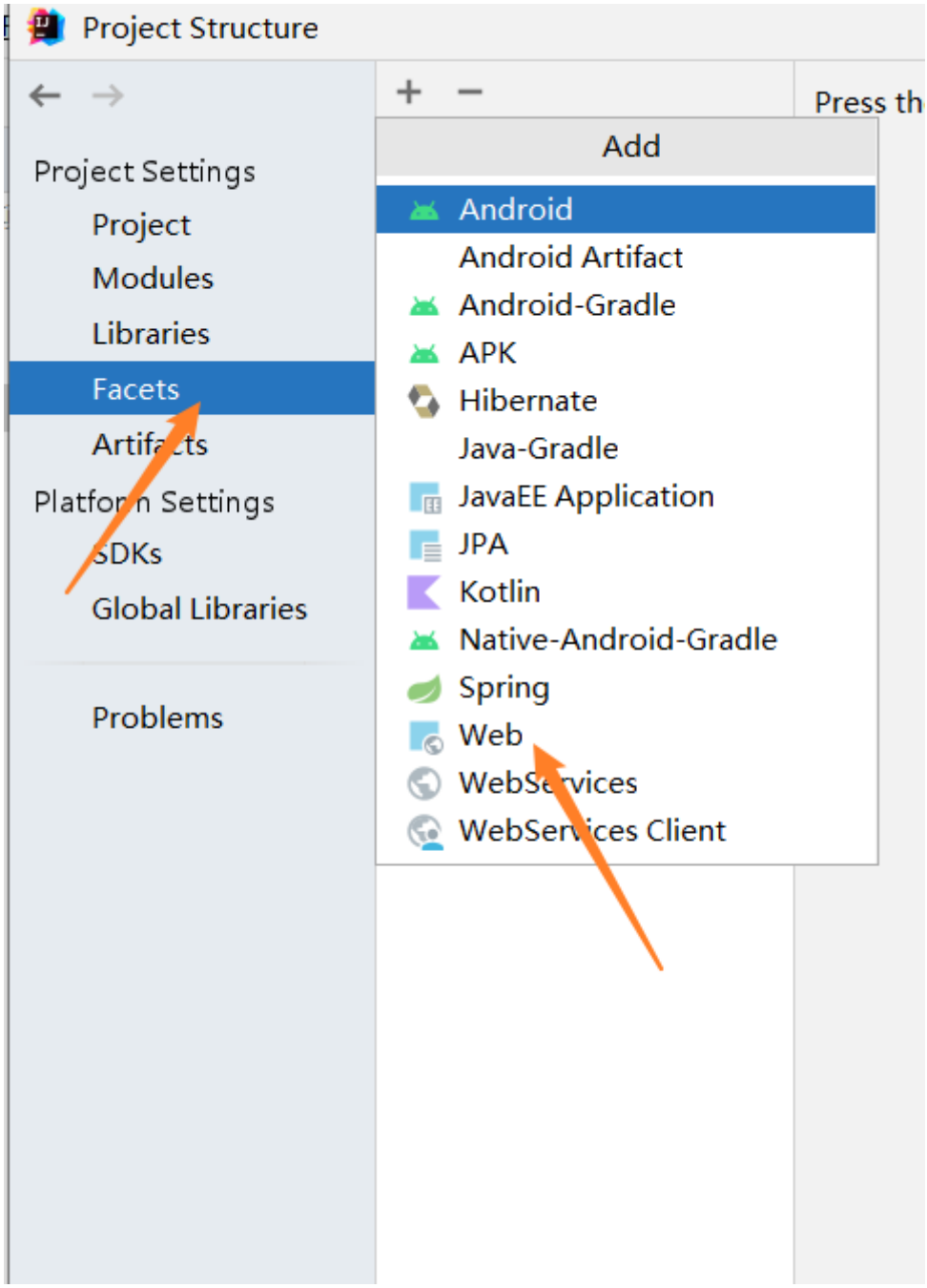
将html项目放在webapps中

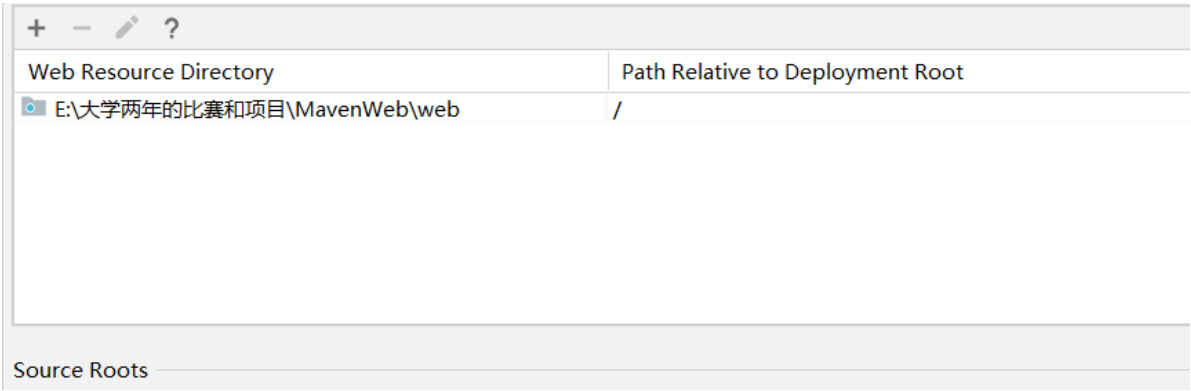
将项目打成一个war包会自动解压

6 在IDEA中创建Web项目

双击java




Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
Add
Android
Android Artifact
Android-Gradle
APK
Hibernate
Java-Gradle
JavaEE Application
JPA
Kotlin
Native-Android-Gradle
Spring
Web
WebServices
WebServices Client
Press th


Web Resource Directory
Path Relative to Deployment Root
E:\大学两年的比赛和项目\MavenWeb\web
/
Source Roots

创建成功

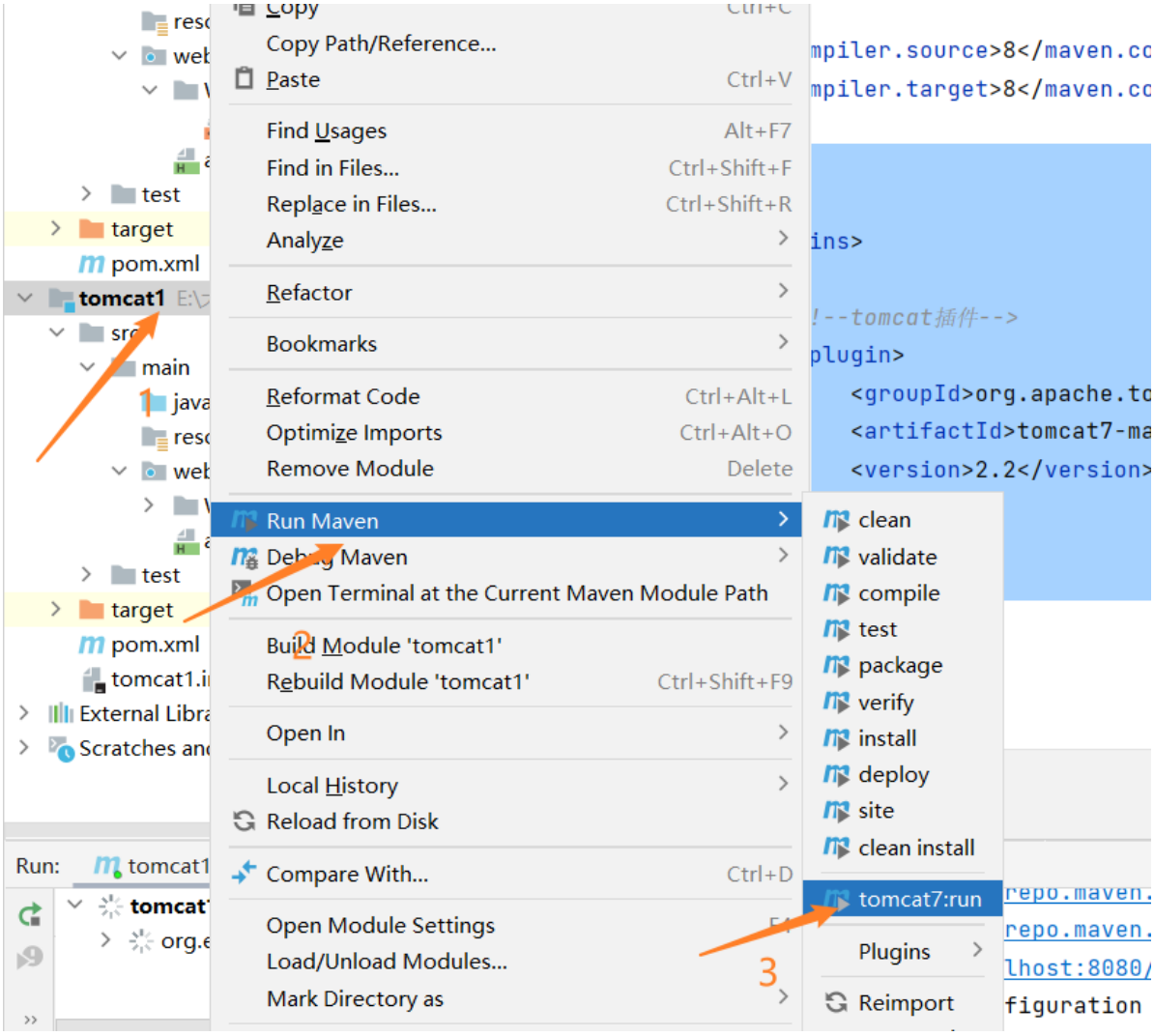
IDEA中使用TomCat

1 使用插件的方法

1 在pom中导入坐标

```
<build>
    <plugins>

        <!--tomcat插件-->
        <plugin>
            <groupId>org.apache.tomcat.maven</groupId>
            <artifactId>tomcat7-maven-plugin</artifactId>
            <version>2.2</version>
        </plugin>
    </plugins>
</build>
```

2 运行步骤



Servlet

1 Servlet快速入门

1 导入依赖

```
    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.1.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
```

2 创建ServletDemo1类

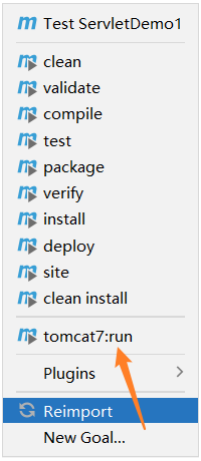
将ServletDemo1继承Servlet类

指定访问路径

实现具体方法

```
public class ServletDemo1 implements Servlet {
    @Override
    public void init(ServletConfig servletConfig) throws ServletException {

    }

    @Override
    public ServletConfig getServletConfig() {
        return null;
    }

    @Override
    public void service(ServletRequest servletRequest, ServletResponse
servletResponse) throws ServletException, IOException {
        System.out.println("Servlet hello world...");
    }

    @Override
    public String getServletInfo() {
        return null;
    }

    @Override
    public void destroy() {

    }
}
```

3 运行展示结果



网页空白

输出窗口显示

```
Servlet hello world...
```

2 Servlet方法

1 init初始化方法

```
    **
     * 初始化方法
     * 1.调用时机：默认方法：Servlet被第一次访问时，调用
     * 2.调用次数：1次
     * @param servletConfig
     * @throws ServletException
     */
    @Override
    public void init(ServletConfig servletConfig) throws ServletException {
        System.out.println("init...");
    }
```

这个init只会执行一次

loadOnStarup 控制 Servlet创建对象

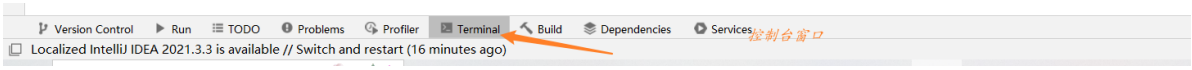
```
@WebServlet(value = "/demo2",loadOnStartup = -1) // loadOnStartup = -1 是默认的
```

```
@WebServlet(value = "/demo2",loadOnStartup = 1) // loadOnStartup 改成正数的时候，没有启动Servlet服务的时候 init方法也会被调用
```

2 service方法

```
/**
     * 提供服务
     * 1 调用时间：每一次Servlet被访问时被调用
     * 2 调用次数：多次
     * @param servletRequest
     * @param servletResponse
```

```
     * @throws ServletException
     * @throws IOException
     */

    @Override
    public void service(ServletRequest servletRequest, ServletResponse
servletResponse) throws ServletException, IOException {
        System.out.println("Servlet hello world...");
    }
```



手动运行tomcat命令

```
mvn tomcat7:run
```

3 destroy方法

```
    /**
     * 销毁方法
     * 1 调用时机： 内存释放或者服务器关闭的时候，Servlet对象会被销毁，调用
     * 2 调用次数： 只调用一次
     */
    @Override
    public void destroy() {
        System.out.println("destroy...");
    }
```

手动关闭时可以触发这个函数执行

```
ctrl + c
```

4 ServletConfig 方法

```
    private ServletConfig config;

    /**
     * 初始化方法
     * 1.调用时机：默认方法：Servlet被第一次访问时，调用
     * 2.调用次数：1次
     * @param servletConfig
     * @throws ServletException
     */
    @Override
    public void init(ServletConfig servletConfig) throws ServletException {
        this.config = servletConfig;
        System.out.println("init...");
    }

    /**
     * 返回Config对象
     * 可以将init的形参的作用域提升
```

```
     * 创建一个对象进行传递
     * @return
     */
```

3 Servlet体系结构

1 继承HttpServlet

内部函数

```
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        System.out.println("get...");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        System.out.println("post...");
    }
```

doGet() doPost() 是自己封装出来的方法

2 urlPattern配置

1 一个Servlet可以配置多个路径

```
/**
 *
 * 配置多个路径
 *
 * @ClassName ServletDemo5
 * @Description TODO
 * @Author lenovo
 * @Date 2022/8/18 22:19
 * @Version 1.0
 */
@WebServlet(urlPatterns =  {"/demo5","/demo6"})
public class ServletDemo5 extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        System.out.println("get...");
        System.out.println("/demo7");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        System.out.println("post...");
```

```
    }
}
```

2 精确匹配路径

```
/**
 *
 * 精确匹配
 *
 * @ClassName ServletDemo5
 * @Description TODO
 * @Author lenovo
 * @Date 2022/8/18 22:19
 * @Version 1.0
 */
@WebServlet("user/demo6")
```

3 目录匹配

```
/**
 *
 * 精确匹配
 *
 * @ClassName ServletDemo5
 * @Description TODO
 * @Author lenovo
 * @Date 2022/8/18 22:19
 * @Version 1.0
 */
@WebServlet("user/*")
```

4 扩展名匹配

```

/**
 *
 * 扩展名匹配
 *
 * @ClassName ServletDemo5
 * @Description TODO
 * @Author lenovo
 * @Date 2022/8/18 22:19
 * @Version 1.0
 */
@WebServlet("*.demo6")
```

5 任意匹配

```
/**
 *
 * 任意匹配
 *
 * @ClassName ServletDemo5
 * @Description TODO
 * @Author lenovo
 * @Date 2022/8/18 22:19
 * @Version 1.0
 */
@WebServlet("/")
```

Request and Respondse

Reques Get函数

```
// String getMethod()：获取请求方式： GET
String method = req.getMethod();
System.out.println(method);
```

```
// String getContextPath()：获取虚拟目录(项目访问路径)： /MavenWeb

String contextPath = req.getContextPath();
System.out.println(contextPath);
```

```
// String getRequestURL(): 获取URL http://localhost:8080/MavenWeb/req
StringBuffer requestURL = req.getRequestURL();
System.out.println(requestURL);
```

```
// String getRequestURI()：获取URI(统一资源标识符)： /MavenWeb/req
String requestURI = req.getRequestURI();
System.out.println(requestURI);
```

```
// String getQueryString()：获取请求参数（GET方式）：?username=zhangsan&password=123
String queryString = req.getQueryString();
System.out.println(queryString);
```

```
 @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        // String getMethod()：获取请求方式： GET
        String method = req.getMethod();
        System.out.println(method);

        // String getContextPath()：获取虚拟目录(项目访问路径)： /MavenWeb

        String contextPath = req.getContextPath();
        System.out.println(contextPath);

        // String getRequestURL(): 获取URL http://localhost:8080/MavenWeb/req
        StringBuffer requestURL = req.getRequestURL();
        System.out.println(requestURL);
```

```
        // String getRequestURI(): 获取URI(统一资源标识符):  /MavenWeb/req
        String requestURI = req.getRequestURI();
        System.out.println(requestURI);

        // String getQueryString(): 获取请求参数（GET方式）: ?
username=zhangsan&password=123
        String queryString = req.getQueryString();
        System.out.println(queryString);
    }
```

Request Post函数

```
  //  获取post 请求体：请求参数
    //    1. 获取字符输入流

        BufferedReader reader = req.getReader();

        String line = reader.readLine();

        System.out.println(line);
```

Request通过请求方式请求参数

1 req.getParameterMap()

```
 Map<String, String[]> map = req.getParameterMap();

        for(String key : map.keySet()){
            System.out.print(key + ":");

            String[] values = map.get(key);

            for(String value : values){
                System.out.print(value + " ");
            }

            System.out.println();
        }
```

2 req.getParameterValues

```
String[] hobbies = req.getParameterValues("hobby");

    for (String hobby : hobbies) {
        System.out.println(hobby);
    }
```

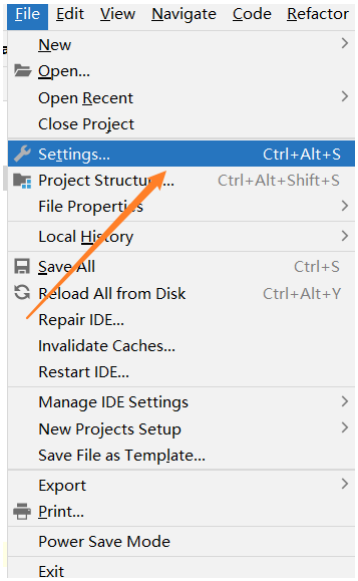
3 req.getParameter

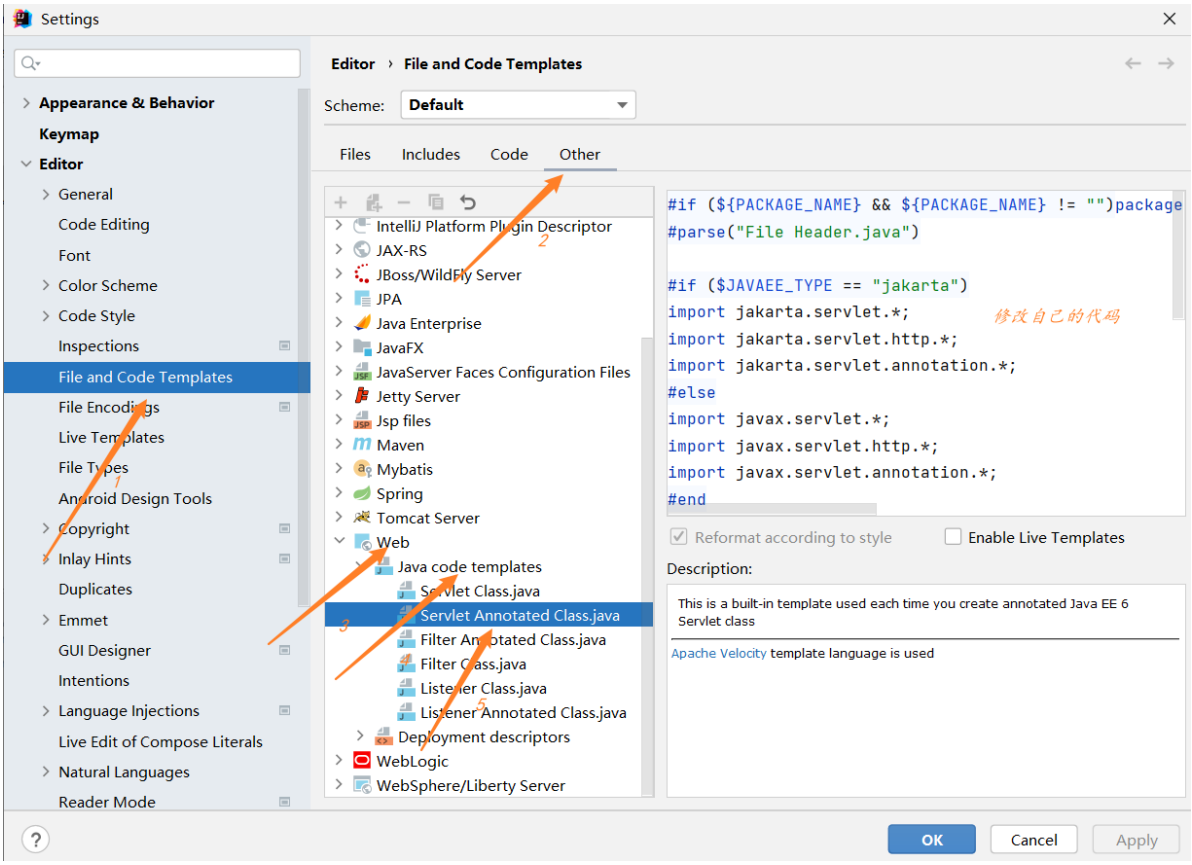
```
String username = req.getParameter("username");
String password = req.getParameter("password");
System.out.println(username);
System.out.println(password);
```

当请求时是post的形式

```
// 调用 doPost请求的时候

this.doGet(req,resp);
```

IDEA模板创建Servlet





请求参数乱码问题

1 post请求

```
// post 请求解决乱码问题
// 设置字符输入流的编码
request.setCharacterEncoding("UTF-8");
```

2 get请求

出错问题

```
package com.itheima.web;

import java.io.UnsupportedEncodingException;
import java.net.URLDecoder;
import java.net.URLEncoder;

/**
 * @ClassName URLDemo
 * @Description TODO
 * @Author lenovo
 * @Date 2022/8/20 14:45
 * @Version 1.0
 */

public class URLDemo {

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

        String username = "张三";

        //    url编码

        String encode = URLEncoder.encode(username, "utf-8");

        // ISO-8859-1 解码

        String decode = URLDecoder.decode(encode, "ISO-8859-1");

        // 获得字节数组

        byte[] bytes = decode.getBytes("ISO-8859-1");

        for (byte aByte : bytes) {
            System.out.print(aByte + " ");
        }

        System.out.println();

        System.out.println(encode);

        System.out.println(decode);

    //    将字节数组转换成字符串

        String s = new String(bytes, "utf-8");

        System.out.println(s);

    }

}
```

```
//    解决get乱码问题 先编码后解码

        String newUsername = new
String(username.getBytes(StandardCharsets.ISO_8859_1), StandardCharsets.UTF_8);

        System.out.println(newUsername);
```

request请求转发

```
System.out.println("demo5...");

//    进行转发操作

request.getRequestDispatcher("/RequsetDemo5").forward(request,response);
```

1 设置转发对象

```
//    设置 转发对象

request.setAttribute("msg","hello");
```

2 接受转发的请求对象

```
Object msg = request.getAttribute("msg");

System.out.println(msg);
```

Response

重定向

方法一

```
    //    设置响应号码

        response.setStatus(302);

    //    设置响应头 Location

        response.setHeader("Location","/MavenWeb/ResponseDemo2");
```

方法二

```
    // //    设置重定向的建议步骤

        response.sendRedirect("/MavenWeb/ResponseDemo2");
```

加不加虚拟目录

在服务器内部之间转换不需要加上虚拟目录

在网页之间转变需要加上虚拟目录

Response响应字符&字节数据

```
//    1 读取文件

        FileInputStream fileInputStream = new FileInputStream("E:\\大学两年的比赛和项目\\MavenWeb\\src\\main\\java\\com\\itheima\\web\\response\\2.jpg");

        //    2 获得字节输出流

        ServletOutputStream os = response.getOutputStream();

        //    3 完成copy

        byte[] bytes = new byte[1024];

        int len = 0;

        while((len = fileInputStream.read(bytes))!= -1){
            os.write(bytes,0,len);

            System.out.println(bytes);

            System.out.println(len);
        }
```

```
while((len = fileInputStream.read(bytes))!= -1){
            os.write(bytes,0,len);

            System.out.println(bytes);

            System.out.println(len);
        }
```

Spring二刷

耦合度

在书写代码的时候我们因该是追求低耦合的开发

- 代码书写现状
  - 耦合度偏高

业务层实现

```
public class BookServiceImpl implements BookService {

    private BookDao bookDao = new BookDaoImpl2();

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

导致这个地方也需要相应的变化

当BookDaoImpl发生改变的时候

数据层实现

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

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

如此一来有需要重新编译 打包 发布 这样的话 就会浪费资源
原因是因为耦合度比较高

解决方法是：**不要自己的主动去 new 一个对象 而是 让外部提供一个对象 这种思想是 IoC**

- **IoC（Inversion of Control）控制反转**
  - 使用对象时，由主动new产生对象转换为由**外部**提供对象，此过程中对象创建控制权由程序转移到**外部**，此思想称为控制反转

- **IoC（Inversion of Control）控制反转**
  - 使用对象时，由主动new产生对象转换为由**外部**提供对象，此过程中对象创建控制权由程序转移到**外部**，此思想称为控制反转
- Spring技术对IoC思想进行了实现
  - Spring提供了一个容器，称为IoC容器，用来充当IoC思想中的“外部”

业务层实现

```
public class BookServiceImpl implements BookService {

    private BookDao bookDao;

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

数据层实现

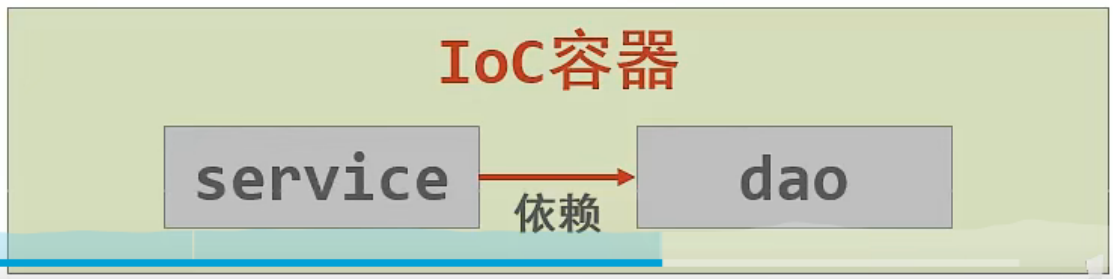
```
public class BookDaoImpl implements BookDao {
    public void save() {
        System.out.println("book dao save ...");
    }
}
```

- IoC容器负责对象的创建、初始化等一系列工作，被创建或被管理的对象在IoC容器中统称为Bean

- **DI（Dependency Injection）依赖注入**

如果两个bean之间存在依赖关系，那么IoC容器会将这两个bean之间进行绑定

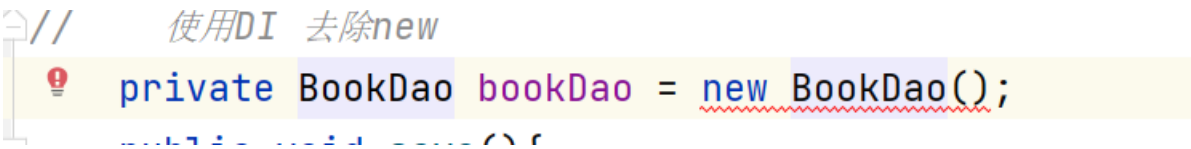
  - 在容器中建立bean与bean之间的依赖关系的整个过程，称为依赖注入



接口

1、**接口的出现是因为 java类之间的继承是单一的，为了打破这个局面，后来引出了接口这个概念，类是可以继承多个接口的**

2、**接口是不能创造对象的** 使用接口创建对象直接报错了



DI实现 和 为什么需要这个

创建对象的时候使用他的实现类

```
private Service service = new ServiceImpl();
```

这个实现类的内部

```
public class ServiceImpl implements Service {

    // private BookDao bookDao = new BookDaoImpl();

    private BookDao bookDao = new BookDaoImpl();

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

在这个是实现类的内部 使用了另外一个接口 使用 对应的实现类 创建一个 对象

```
   private BookDao bookDao = new BookDaoImpl();
```

这么一来 这个耦合度就比较大

删除new 的方法之后

加入bean的管理

这个setter方法是 IoC容器去调用的

这个也就是DI 实现

Bean造对象

1 直接使用Bean造对象

直接通过获取bean的形式 获取一个接口 在 xml 文件中将这个接口 配置成bean的形式

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

//方式一

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

bookDao.save();
```

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

</bean>
```

2 使用工厂造bean

```
public class OrderDaoFactory {

    //  自己 创建一个dao 返回回去

    public static OrderDao getOrderDao(){

        return new OrderDaoImpl();

    }

}
```


```
OrderDao orderDao = OrderDaoFactory.getOrderDao();

orderDao.save();
```

修改成bean的形式

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

// 方式二

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

orderDao.save();
```

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

在配置的时候 加以说明 工厂造对象的方法是 哪一个

补充:

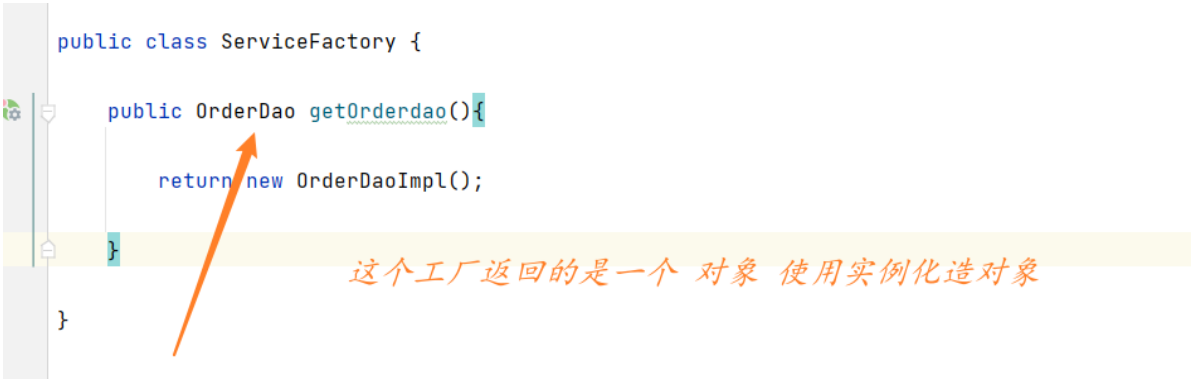
区别:

- 1、静态方法是使用static关键字修饰的方法，属于类的，不属于对象；非静态方法是不使用static关键字修饰的普通方法，属于对象，不属于类。
- **2、静态方法可以直接调用，类名调用和对象调用；非静态方法只能通过对象调用。**
- 3、生命周期不同。

3使用 实例化对象造对象

```
ServiceFactory serviceFactory = new ServiceFactory();

OrderDao orderDao = serviceFactory.getOrderdao();
```

```
public class ServiceFactory {

    public OrderDao getOrderdao(){

        return new OrderDaoImpl();

    }

}
```



改成bean的形式

4 使用FactoryBean简化开发

```
package com.itheima.Factory;

import com.itheima.dao.OrderDao;
import com.itheima.dao.impl.OrderDaoImpl;
import org.springframework.beans.factory.FactoryBean;
import org.springframework.core.annotation.Order;

/**
 * @ClassName UserDaoFactoryBean
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/8 21:36
 * @Version 1.0
 */

public class UserDaoFactoryBean implements FactoryBean<OrderDao> {

    public OrderDao getObject() throws Exception {
        return new OrderDaoImpl();
    }

    public Class<?> getObjectType() {
```

```
        return OrderDao.class;

    }
}
```

```
<bean id="orderdao" class="com.itheima.Factory.UserDaoFactoryBean"></bean>
```

使用FactoryBean 对象 简化很多

Setter注入

第一次看你这个东西的时候 不是很清楚这里面的 setter的方法是怎么被调用的

第二次看的时候就是到原来执行的都是 容器进行执行

代码 分为以下三部分

- 测试类的书写

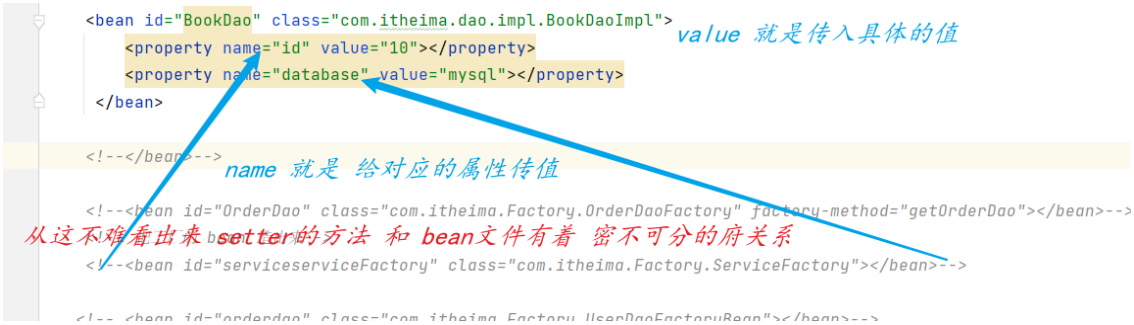
```
package com.itheima;

import com.itheima.dao.BookDao;
import javafx.scene.transform.Scale;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * @ClassName testbean
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/8 21:51
 * @Version 1.0
 */

public class testbean {

    @Test
    public void test01(){

        ApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationcontext.xml");

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

        bookDao.save();


    }

}
```

- 接口类的实现

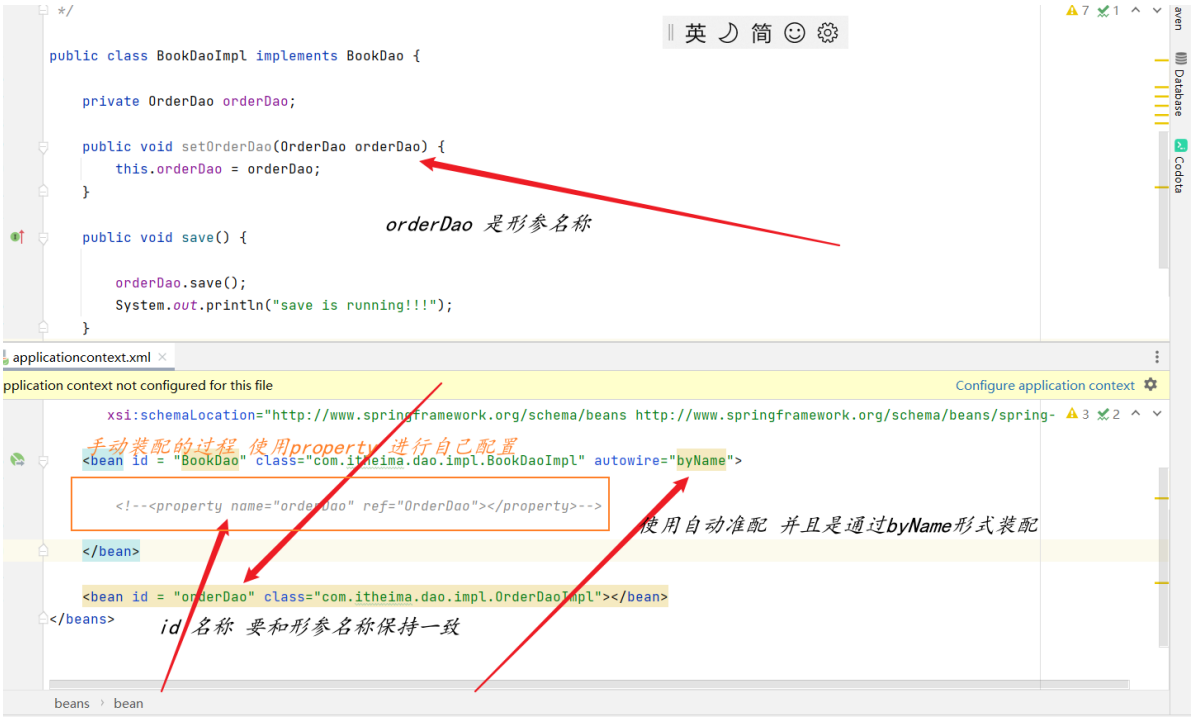
```java
package com.itheima.dao.impl;

import com.itheima.dao.BookDao;
import com.itheima.dao.OrderDao;
import org.springframework.core.annotation.Order;

import java.security.PrivateKey;

/**
 * @ClassName BookDaoImpl
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/8 10:01
 * @Version 1.0
 */

public class BookDaoImpl implements BookDao {

    private Integer id;

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

    public void setDatabase(String database) {
        this.database = database;
    }

    private String database;

    public void save() {

        System.out.println("save is running!!!");
        System.out.println("id == >" + id + " database ==> " + database );

    }

}
```

- bean配置 文件的书写 xml

```xml
<bean id="BookDao" class="com.itheima.dao.impl.BookDaoImpl">
    <property name="id" value="10"></property>
    <property name="database" value="mysql"></property>
 </bean>
```

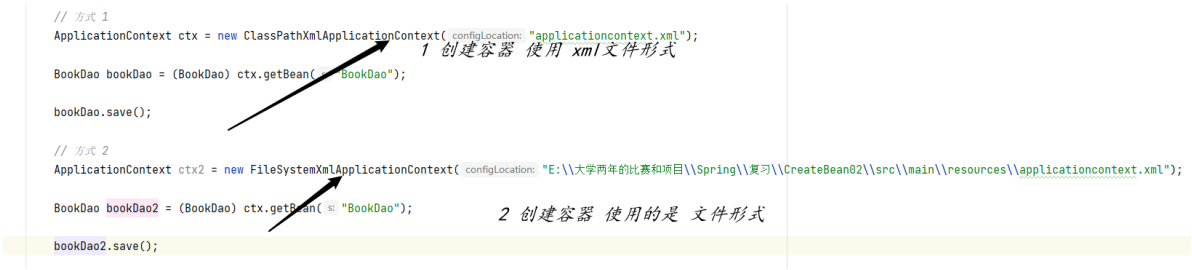


自动装配



容器

1 创建容器



2 获取bean形式

3 使用 FactoryBean

```
Resource resources = new ClassPathResource( path: "applicationContext.xml");
BeanFactory bf = new XmlBeanFactory( resource: resources);
BookDao bookDao = (BookDao) bf.getBean( name: "BookDao");
bookDao.save();
```

2 通过 xml 获取 resources

1 使用Xml 获取 resuorces

3 获取 bean

注解开发

一开始使用注解进行开发文件

一步一步进行替换

```
import com.itheima.dao.BookDao;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;

/**
 * @ClassName BookDaoImpl
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/8 10:01
 * @Version 1.0
 */
// @Repository
// @Component
public class BookDaoImpl implements BookDao {

    public void save() {

        System.out.println("save is running!!!");
    }

}
```

现在 使用 Component组件的方式 去替代这个 <bean> 标签

```
<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.
    <bean id = "BookDao" class="com.itheima.dao.impl.BookDaoImpl" autowire="byName">
    </bean>

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

</beans>
```

最初的写法是 在 xml 文件中 使用 <bean> 标签进行配置 bean 最终生成的bean就是class指定的

此时又有一个问题 你在这个地方加了一个Component注解之后 怎么加载到 xml文件中来呢 于是 使用 component-scan扫描 识别那个

后来啊 java开发最喜欢的就是使用配置类来代替一些东西

于是产生了配置类 代替 xml 文件

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

    <!--<bean id = "BookDao" class="com.itheima.dao.impl.BookDaoImpl" autowire="byNam
    <!--</bean>-->

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

</beans>
```

```
package com.itheima.config;

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

/**
 * @ClassName SpringConfig
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/9 14:52
 * @Version 1.0
 */
@Configuration
@ComponentScan("com.itheima")
public class SpringConfig {
}
```

使用Configuration 注解可以代替剩下的所有文件

使用这个注解代替这个 标签

```
    //使用文件开发形式
    // ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationcontext.xml.bak");
    //
    // BookDao bookDao = ctx.getBean(BookDao.class);
    //
    // bookDao.save();

//    使用注解开发形式

    ApplicationContext ctx= new AnnotationConfigApplicationContext( ...componentClasses: SpringConfig.class);

    BookDao bookDao = ctx.getBean( aClass: BookDao.class);

    bookDao.save();
```

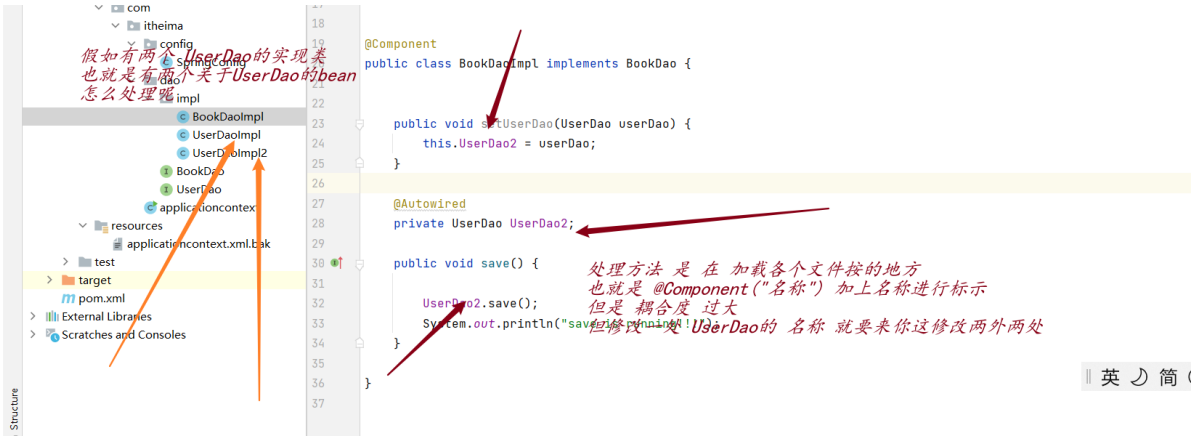
原来产生容器的时候使用的就是 ClassPathXml这个类

现在使用注解的使用使用的类 是 Annotation这个类

正式使用Autowired

```
@Component
public class BookDaoImpl implements BookDao {


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

    @Autowired
    private UserDao userDao;

    public void save() {

        userDao.save();
        System.out.println("save is running!!!");
    }

}
```

Autowired 自动装配这个东西 解释一下
之前在使用 bean这个标签的时候
我们时使用的 是 properties这个文件 id name 进行 传输值
进行处理 但是在这里我们使用自动装配之后 不需要传值处理
这里是用过 表示名进行匹配



假如有两个UserDao的实现类
也就是有两个关于UserDao的bean
怎么处理呢

处理方法 是 在 加载各个文件按的地方
也就是 @Component("名称") 加上名称进行标示
但是 耦合度 过大
但修改 是 UserDao的 名称 就要来你这修改两外两处

```
 * @Date 2022/9/8 10:01
 * @Version 1.0
 */


@Component
public class BookDaoImpl implements BookDao {


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

    @Autowired
    @Qualifier("UserDao2")
    private UserDao userDao;

    public void save() {

        userDao.save();
        System.out.println("save is running!!!");
    }
```

使用@Qualirier这个注解
只需要将 里面的参数进行更改即可

使用value注解 或者就比较简单就不在这里说明

第三方注解@Bean

```
public class applicationcontext {

    public static void main(String[] args) {

        ApplicationContext ctx= new AnnotationConfigApplicationC

        //DataSource 其实 也是个Bean 知识个 第三方的bean 所以配置起来不
        DataSource dataSource = ctx.getBean( aClass: DataSource.cla

        System.out.println(dataSource);

    }
}
```

在这个地方拿到的就是一个bean的 对象

```
 * @Version 1.0
 */

public class JdbcConfig {

    // 最简单一步就是 加上@Bean注解
    @Bean
    public DataSource dataSource(){

        DruidDataSource ds = new DruidDataSource();

        ds.setDriverClassName("com.mysql.jdbc.Driver");
        ds.setUsername("root");
        ds.setPassword("admin123");
        ds.setUrl("jdbc:mysql://localhost:3306/mybitis");

        return ds;
    }
}
```

创建一个类

在这个地方 加上@Bean注解 就是 说明 返回值 是一个 Bean的形式 这个 就是 创建第三方Bean的形式

这个返回值 是 一个 bean

使自己创建的这个Bean生效的话 也就是 需要导入 到 SpringCongfig类中去

```
 * @Description TODO
 * @Author Typecoh
 * @Date 2022/9/9 14:52
 * @Version 1.0
 */
@Configuration
@ComponentScan("com.itheima")
@Import({JdbcConfig.class})
public class SpringConfig {
}
```

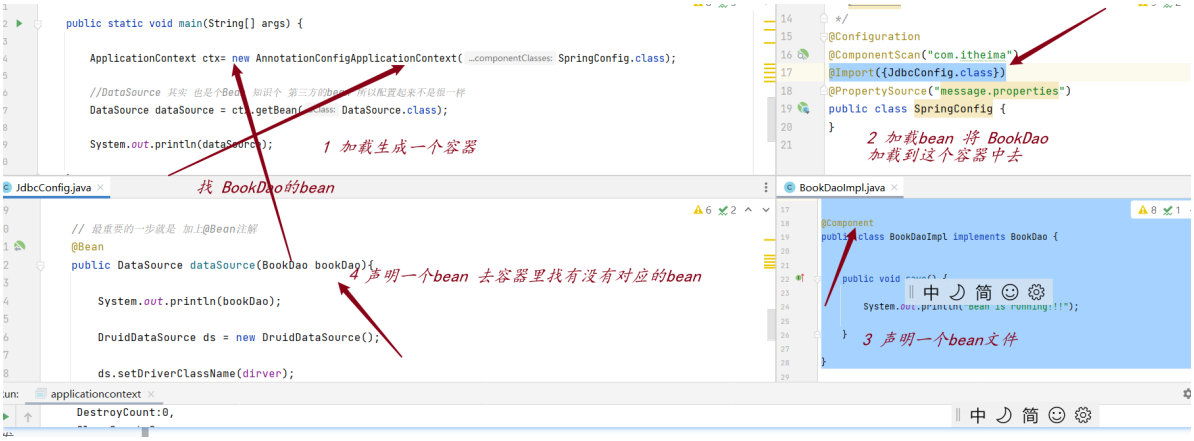
使用@Import方式

当bean需要 文件或者 类型时 直接使用形参传入格式

```
    // 最重要的一步就是 加上@Bean注解
    @Bean
    public DataSource dataSource(BookDao bookDao){

        System.out.println(bookDao);

        DruidDataSource ds = new DruidDataSource();

        ds.setDriverClassName(dirver);
        ds.setUsername(username);
        ds.setPassword(password);
        ds.setUrl(url);

        return ds;
    }
}
```

使用 形参 传入 自动装配

流程是



Jdbc 和 Mybatis

Jdbc的创建 是 比较简单的 在 Spring文件当中有详细的介绍 在这里就不过多 阐述了

但是 为什么又有Mybatis 这个玩意出现呢

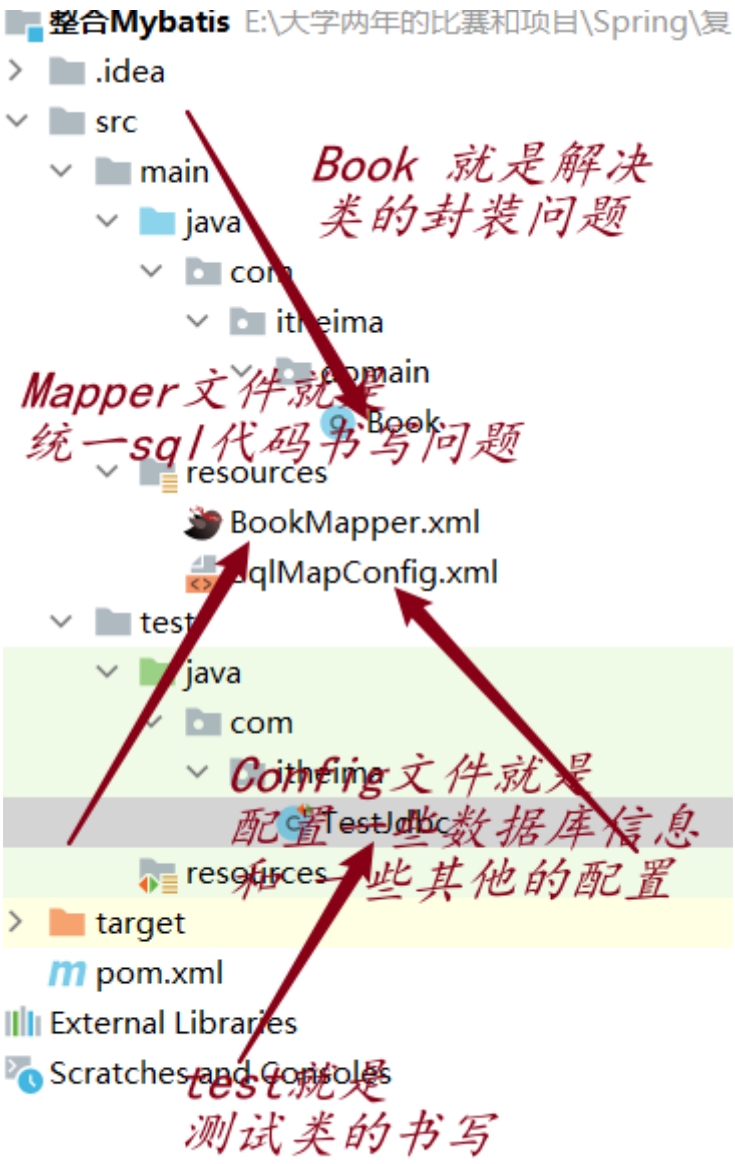
```
public class TestJdbc {

    @Test
    public void test01() throws Exception {

        Class.forName("com.mysql.jdbc.Driver");

        String url = "jdbc:mysql://localhost:3306/mybatis";

        String username = "root";

        String password = "admin123";

        Connection coon = DriverManager.getConnection( url: url, user: username, password: password);

        String sql = "select  * from book where id = 2";

        Statement sttm = coon.createStatement();

        ResultSet resultSet = sttm.executeQuery( sql: sql);

        while(resultSet.next()){

            Book book = new Book();

            int id = resultSet.getInt( columnLabel: "id");
            String name = resultSet.getString( columnLabel: "name");
            String type = resultSet.getString( columnLabel: "type");
            String description = resultSet.getString( columnLabel: "description");

            book.setId(id);
            book.setType(type);
            book.setName(name);
            book.setDescription(description);

            System.out.println(book);
        }
        sttm.close();
    }
```

数据库的固定信息
可以简化 过于繁琐

sql语句的编写 可以统一到文件中

这个类的封装过于固定
所以可以简化

为了简化上面的情况 产生了 mybatis技术



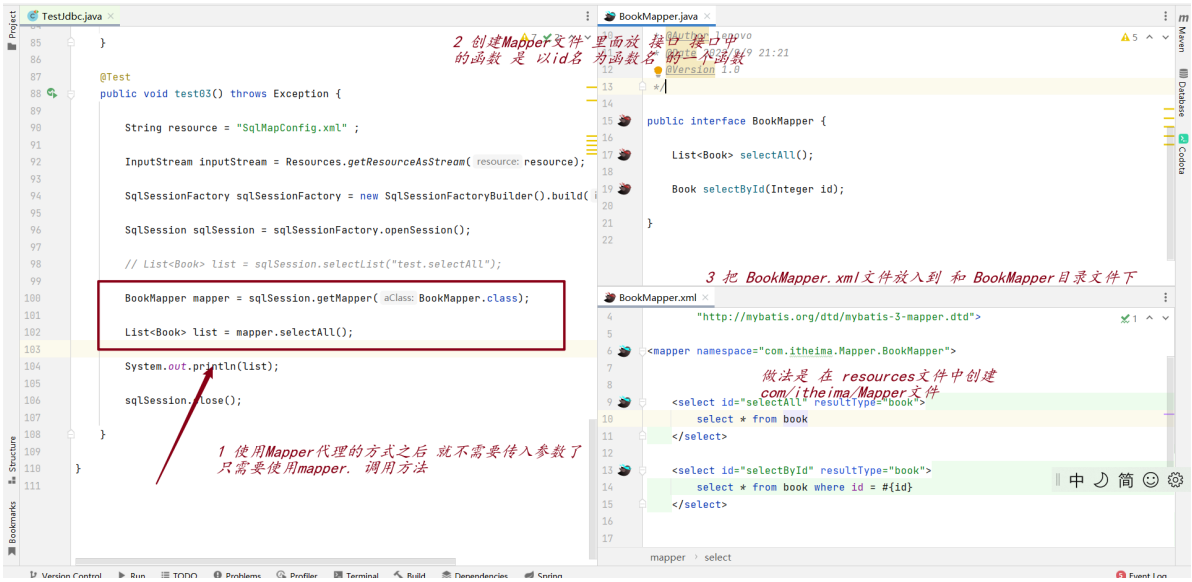

```xml
<mapper namespace="test">

    <select id="selectAll" resultType="com.itheima.domain.Book">
        select * from book
    </select>

</mapper>
```

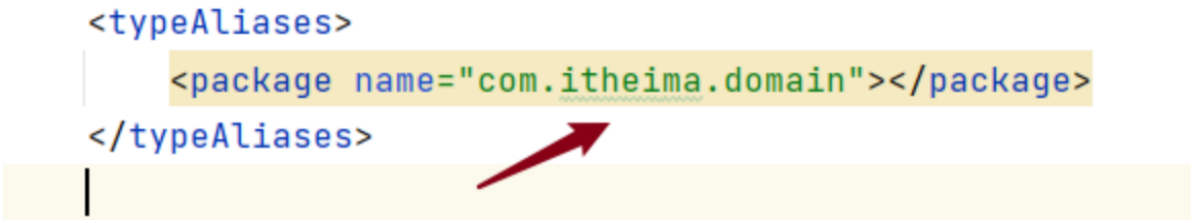
```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
```

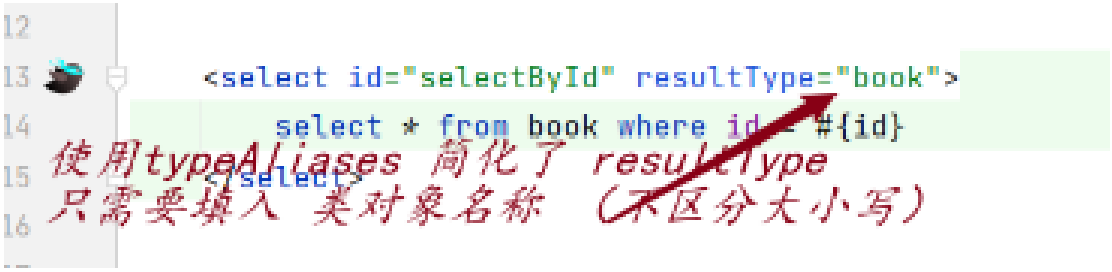
```
                <property name="url"
value="jdbc:mysql://localhost:3306/mybatis"/>
                <property name="username" value="root"/>
                <property name="password" value="admin123"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="BookMapper.xml"/>
    </mappers>
</configuration>
```

```
@Test
public void test02() throws Exception {

    String resource = "SqlMapConfig.xml" ;

    InputStream inputStream = Resources.getResourceAsStream(resource);

    SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

    SqlSession sqlSession = sqlSessionFactory.openSession();

    List<Book> list = sqlSession.selectList("test.selectAll");

    System.out.println(list);

    sqlSession.close();

}
```

为了解决这个办法于是采用Mapper代理的方式去处理







当自己创建一个实体类，如果没有继承父类就会自动继承基类 当继承了父类 就不会继承基类

MyBatis 源码分析（二）：反射模块 - 简书 (get方法)

在 Mybatis 中调用 数据库 中执行 Mapper中的 SQL 语句是 在 Mybatis 中使用 将数据库中的 字段名 去匹配

setXxx 通过反射将类中的变量命名的set方法和数据库 的字段进行对应，如果说 类的变量名和数据库 的字段

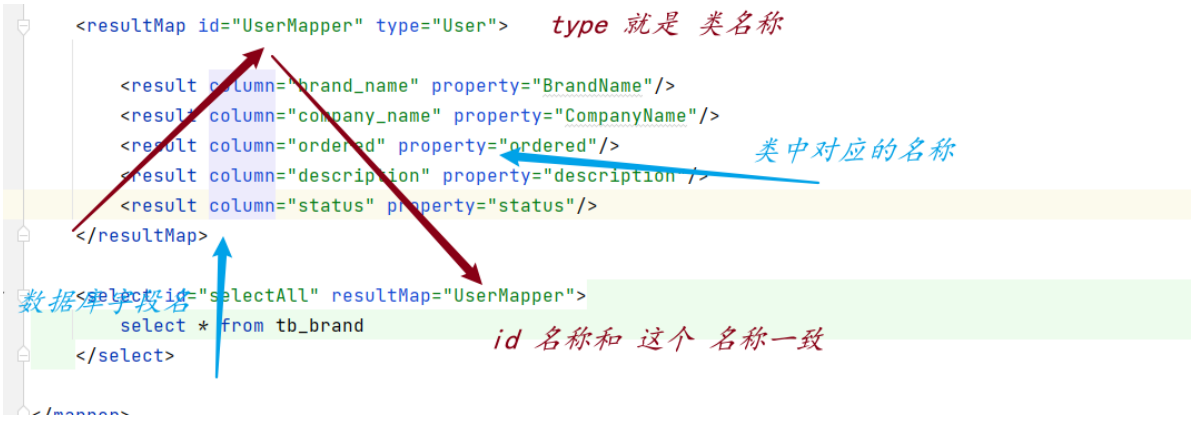
如果一致就去调取这个set的方法，否则就不去调用

反射是可以去拿到 类的函数 类名 等等东西，发生在比如反射就是发生在 java 运行时完成的。 在类加载阶段，虚拟机会在 java 堆中生成一个代表这个类的 java.lang.Class 对象，通过这个 Class 对象就可以访问到 jvm 中的这个类。 Class 类与 class 是不同，Class 是实实在在存在于 java.lang.Class 包中的一个类。

所以为了解决这个问题

我们引入 这个标签解决

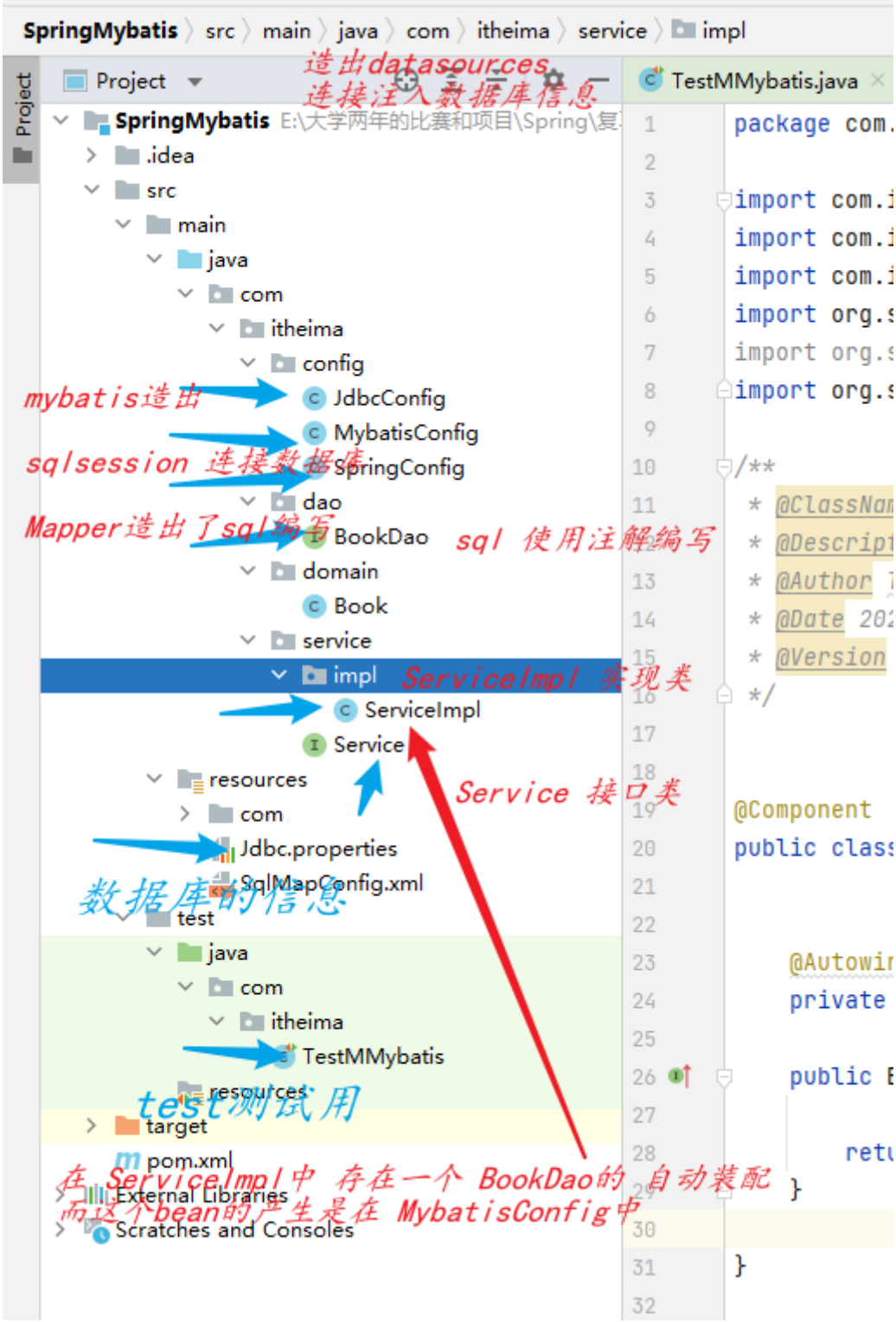
```
<resultMap id="UserMapper" type="User">

    <result column="brand_name" property="BrandName"/>
    <result column="company_name" property="CompanyName"/>
    <result column="ordered" property="ordered"/>
    <result column="description" property="description"/>
    <result column="status" property="status"/>
</resultMap>

<select id="selectAll" resultMap="UserMapper">
    select * from tb_brand
</select>
```



之后一个点就是说 使用注解 简化这个开发

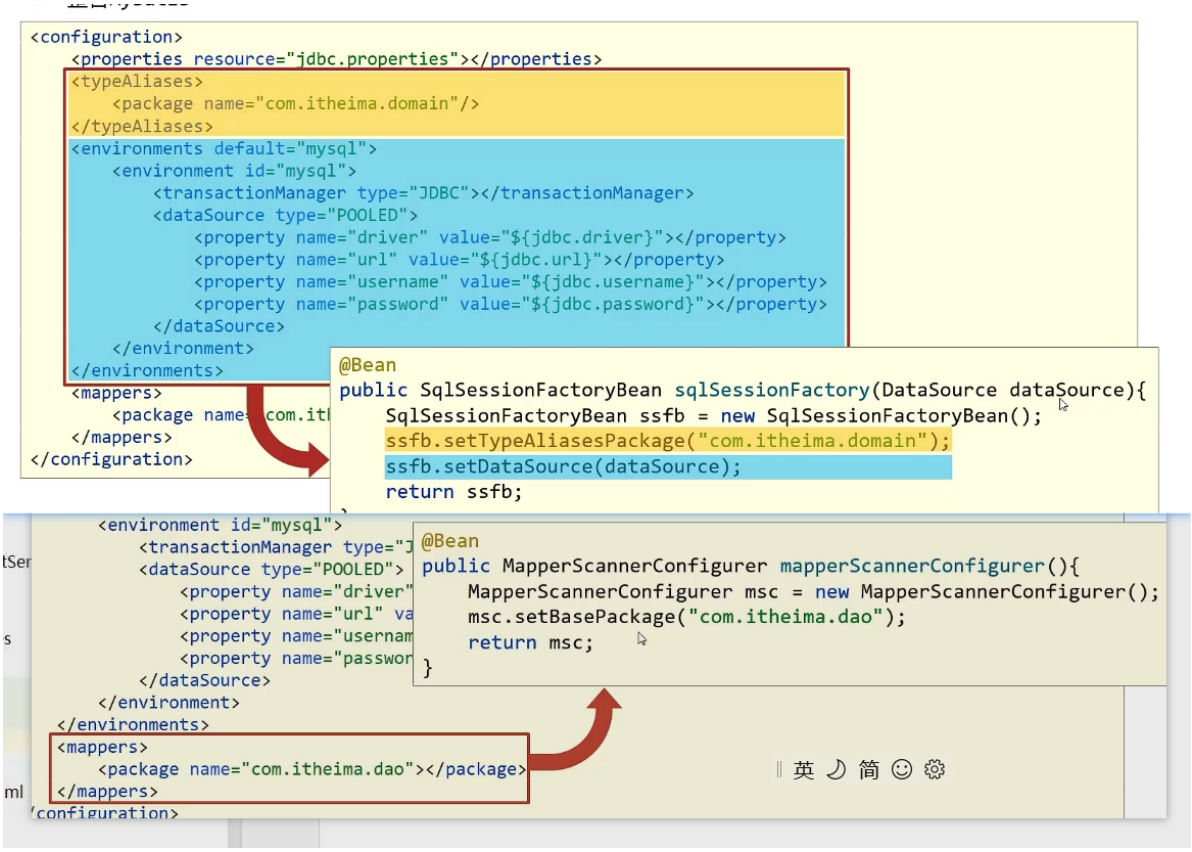
```
@Select("select * from tb_brand where status = #{statis}")
User selectById(Integer status);
```

Spring整合Mybatis



具体代码

整合 前后差异



JUnit整合

首先说说这个JUnit这个玩意的用处是什么，为啥总有一些稀奇古怪的东西

在测试类中

正常去写一个测试数据层代码的时候

```
ApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);

BookDao bean = ctx.getBean(BookDao.class);

List<Book> list = bean.SelectAll();

System.out.println(list);
```

你总要去创建一个容器吧，要是连容器都没有那还玩个啥

但是，只是去测一个类的接口，你就让我写一个容器，完了之后还要拿bean，那这个就有点过分了

于是 JUnit说我来帮你解决这个问题

于是

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = SpringConfig.class)
```

```
@Test

public void test03(){

    Book book = service.SelectById(1);

    System.out.println(book);

}
```

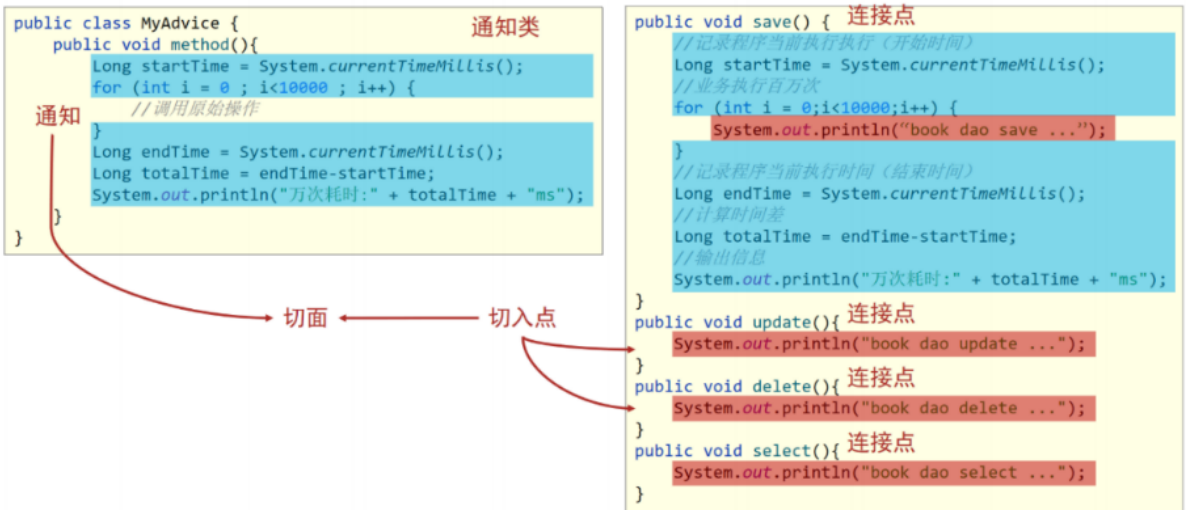
至此就简单方便的执行了

AOP

先来聊聊这个玩意主要是来干什么的，AOP这个好家伙是用来做增强的作用，它可以在原来的基础上不做处理但是能加入一些别的东西进去做增强

那具体是怎么实现的呢？ 先来说说AOP中的几个概念

- **连接点😁：连接点说白了就是一个函数的名称，这个函数就是想来做AOP的那个函数**
- **切入点：但是也不是所有的方法都需要做增强，需要做增强的那些方法被称为切入点**
- **通知：也就是说这个通知是个函数，这个函数是需要增强的内容**
- **通知类：封装这个通知的类被称为通知类**
- **切面：通知和切入点之间的关系描述，我们给起了个名字叫切面**



制作AOP入门案例图示结构

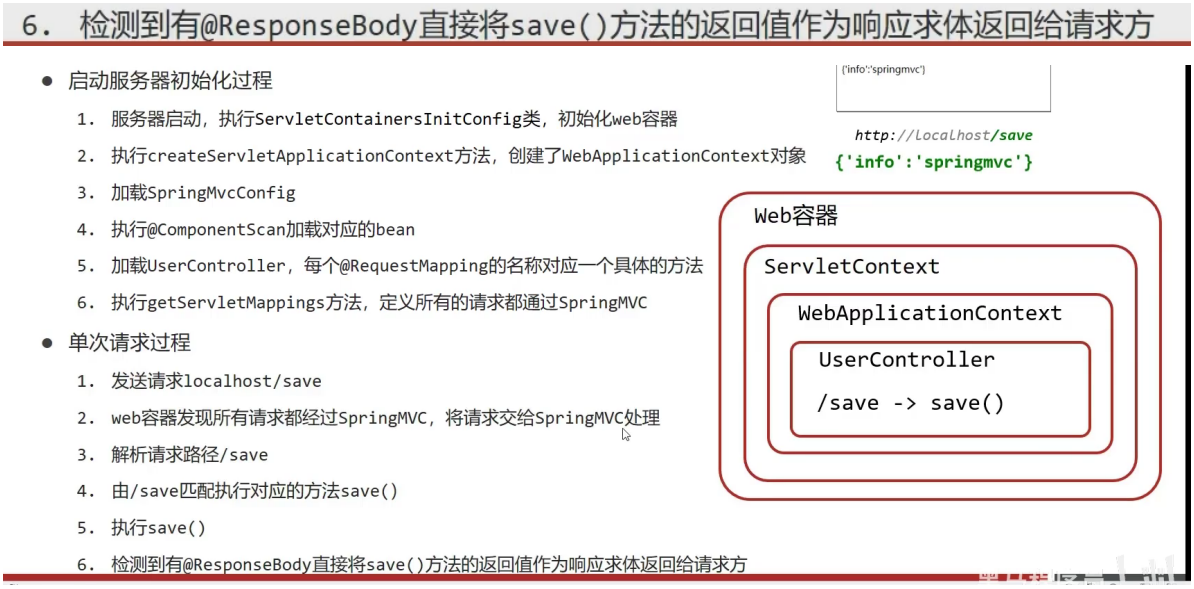
AOP还有其他很多东西，在这里就不一一说明了 在Spring中是有说明的

SpringMvc

SpringMvc这个东西就是在 Servlet技术上进行做了增强，简便操作

先介绍两个东西

- **@Responsebody 注解一般是跟在 @RequestMapping之后，使用了@RequestMapping注解往往返回值是跳转页面；比如jsp形式，但是加上了@Responsebody 之后往往是将正文给显示出来**
- **@Requestbody 这个注解一般是跟在形参的前面，修饰形参，说明形参的那个格式 是以什么形式表示；比如 json，xml**



到此SpringMvc到此时就先到一段了，在SpringMvc中其实有很多相关的知识点，这些知识点是可以结合代码+Spring开发文档+pdf进行学习

SSM整合

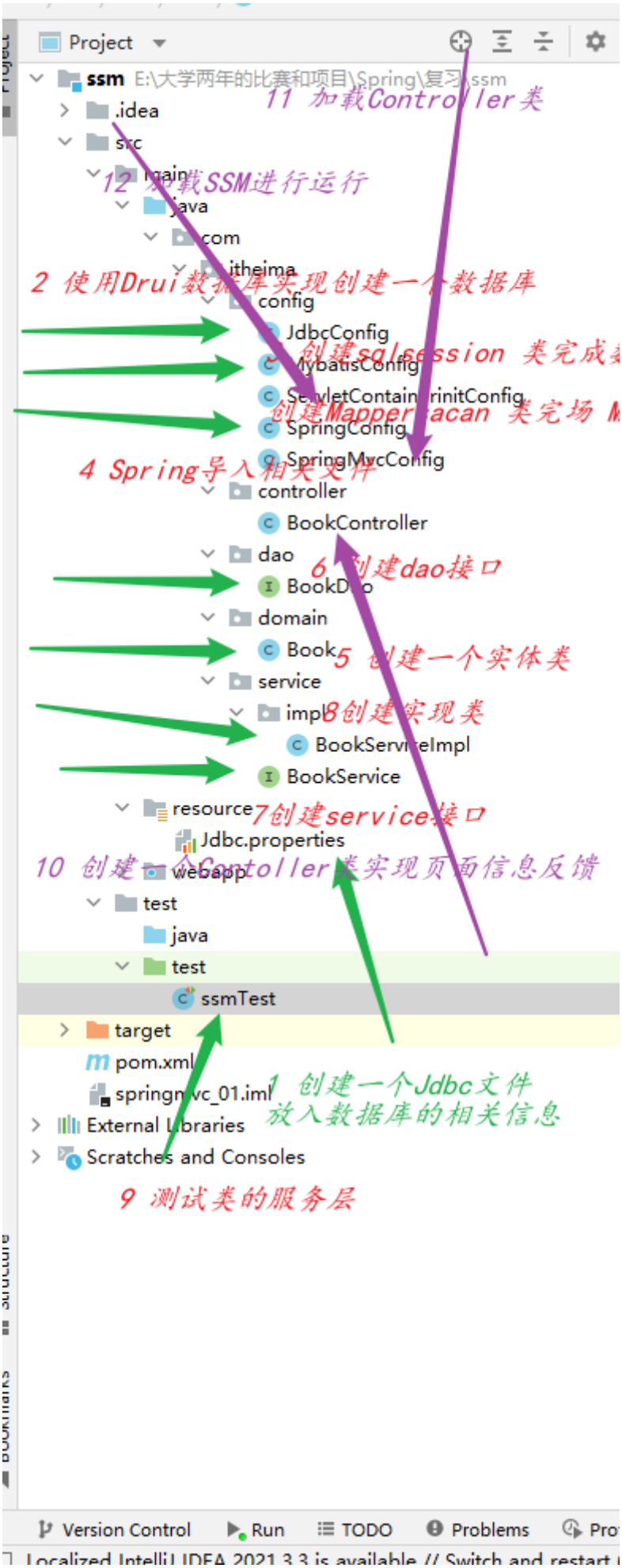
ssm整合技术：是将Sping Mybatis SpringMvc三者整合到一起去

先说一个熟悉的知识点

在使用SSM技术时 统一在这个地方加上前缀 classpath 否则找不到文件

```
@PropertySource("classpath:Jdbc.properties")
```

简单回顾以下整合SSM的步骤


Project
ssm E:\大学两年的比赛和项目\Spring\复习\ssm
.idea
src
main
java
com
itheima
config
JdbcConfig
MybatisConfig
ServletContainerInitConfig
SpringConfig
SpringMvcConfig
controller
BookController
dao
BookDao
domain
Book
service
impl
BookServiceImpl
BookService
resource
Jdbc.properties
webapp
test
java
test
ssmTest
target
pom.xml
springmvc_01.iml
External Libraries
Scratches and Consoles
11 加载Controller类
12 加载SSM进行运行
2 使用Druid数据库实现创建一个数据库
3 创建sqlsession 类完成
创建Mapper.class 类完场
4 Spring导入相关文件
6 创建dao接口
5 创建一个实体类
8创建实现类
7创建service接口
10 创建一个Contoller类实现页面信息反馈
1 创建一个Jdbc文件
放入数据库的相关信息
9 测试类的服务层
Version Control
Run
TODO
Problems
Localized IntelliJ IDEA 2021.3.3 is available // Switch and restart

具体文件代码

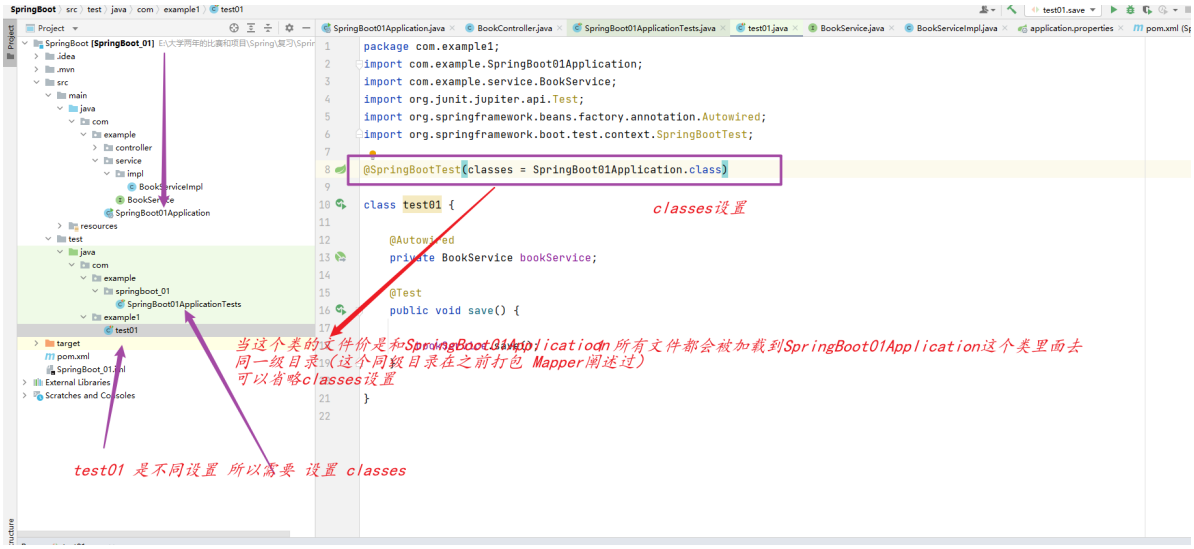
[代码位置]

到此 ssm整合就结束了，同时呢一些细节东西就不过多阐述，只是将流程过了一遍

对于简易开发比如说 rest风格等等这一方面就结合Spring文档查看就行

SpringBoot

在整合Junit中主要涉及 test包和SpringBoot自动生成的加载类的位置是否一致



在整合Mybatis中看看SpringBoot 和 原来之间的区别