目录

[Hibernate 1](#_Toc532736686)

[Hibernate架构 1](#_Toc532736687)

[ORM映射 1](#_Toc532736688)

[普通ORM映射 1](#_Toc532736689)

[关系映射（多表间的映射） 3](#_Toc532736690)

[hibernate持久化 4](#_Toc532736691)

[HQL:hibernate的sql语句 5](#_Toc532736692)

[1.HQL查询 5](#_Toc532736693)

[2.HQL更新和删除 5](#_Toc532736694)

[3执行本地SQL语句 6](#_Toc532736695)

[4 Hibernate Criteria 条件查询 6](#_Toc532736696)

[事务transaction 6](#_Toc532736697)

[Mybatis 7](#_Toc532736698)

[使用方式一：手动添加entry、mapper文件 7](#_Toc532736699)

[使用方式二：使用mybatis generator 生成entity，mapper文件 9](#_Toc532736700)

[mybatis动态sql 9](#_Toc532736701)

[使用别名 11](#_Toc532736702)

[实现关系映射 11](#_Toc532736703)

[1 一对多 11](#_Toc532736704)

[2 多对一 12](#_Toc532736705)

[Spring框架 13](#_Toc532736706)

[Spring框架的架构 13](#_Toc532736707)

[Spring和hibernate整合 14](#_Toc532736708)

[Mybatis和spring整合 14](#_Toc532736709)

[AOP面向方面的编程 16](#_Toc532736710)

[SpringMVC 18](#_Toc532736711)

[基本概念 18](#_Toc532736712)

[Controller中方法的写法 22](#_Toc532736713)

[A @RequestMapping 类中指定一级URL，方法指定二级URL 22](#_Toc532736714)

[B 方法的参数 接收请求中数据的方式 界面--->controller 22](#_Toc532736715)

[C 返回数据的方式 controller-->界面 22](#_Toc532736716)

[D 方法的返回值类型 controller-->界面 22](#_Toc532736717)

[过滤器 23](#_Toc532736718)

[字符集过滤器 23](#_Toc532736719)

[Session过滤器 23](#_Toc532736720)

[Springboot 23](#_Toc532736721)

[controller的使用 23](#_Toc532736722)

[数据库操作 24](#_Toc532736723)

[事务管理 25](#_Toc532736724)

[Web进阶 26](#_Toc532736725)

[对比 27](#_Toc532736726)

[Struts2和springmvc对比 27](#_Toc532736727)

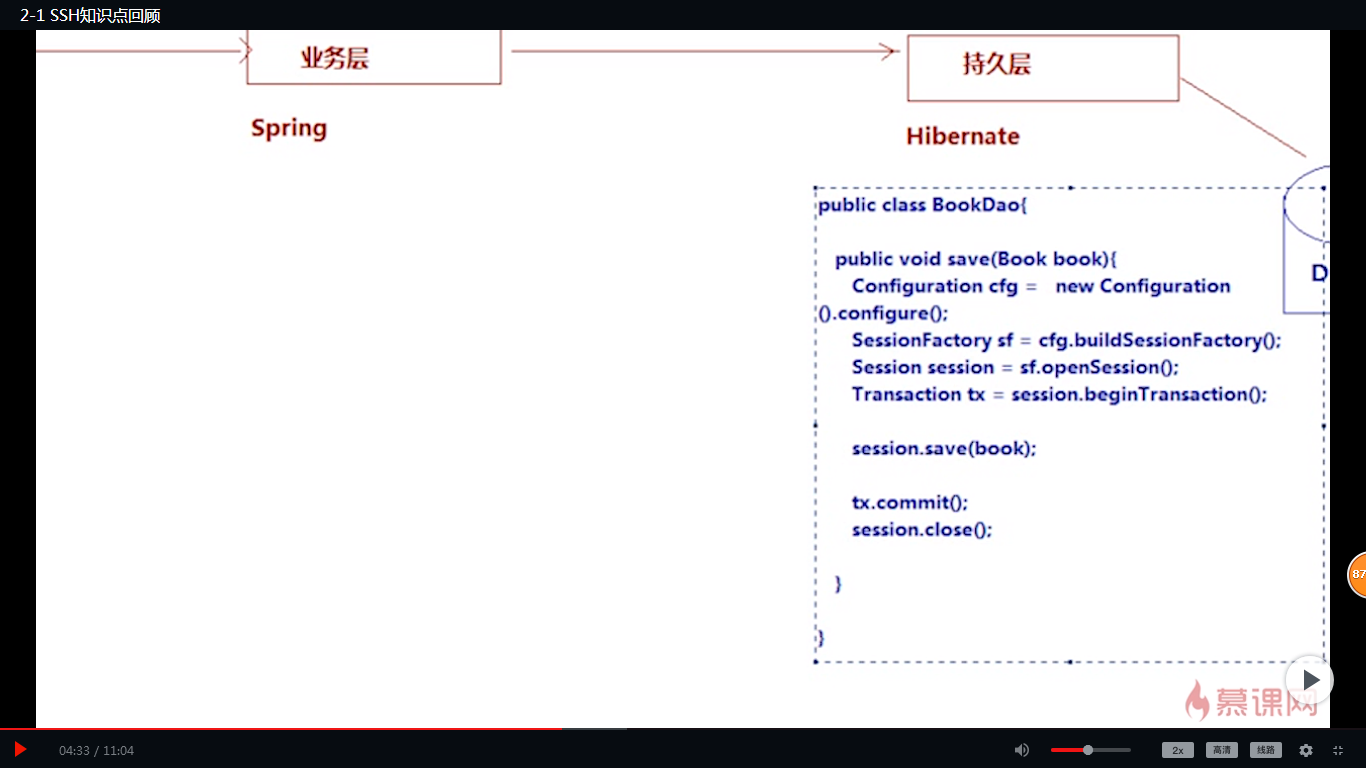
[Spring springmvc springboot对比 28](#_Toc532736728)

# Hibernate

Hibernate 官网：<http://hibernate.org/om/> 优点：实现类和数据库的表的映射(绑定)

## Hibernate架构

单独使用hibernate操作数据时



1 Configuration 读取核心配置文件:hibernate.cfg.xml文件 程序实例化时只会被创建一次

2 Sessionfactory 会话工厂，用于创建会话 做成单例或静态的，须知Configuration，HibernateSessionFactory类：读取hibernate配置和创建sessionfactory等

configuration.configure();

serviceRegistry=new ServiceRegistryBuilder().applySettings(configuration.getProperties()).buildServiceRegistry();

sessionFactory = configuration.buildSessionFactory(serviceRegistry);

3 Session 会话 连接对象 去执行对数据库的增删改查等操作

4 Transaction 事务

5 Query 一个sql语句 实现有sql语句的查询

6 Criteria 实现没有sql语句的查询

## ORM映射

普通ORM映射

（object Java类relation表mapping映射）两种方式二选一：XML文件和注解@(Java类添加)

A手动

1编写xxx.hbm.xml文件 2在hibernate.cfg.xml文件中配置xxx.hbm.xml文件

B利用工具：Hibernate 逆向工程即根据数据库表自动生成Java类和mapping文件

DB browser 找到数据库连接对象下要映射的table，右击-hibernate-Reverse-Engineering

package common;import java.util.Date;

package common; @Entity @Table(name = "room", catalog = "food")

public class Book {//省略getter和setter方法 object Java类

private int bid; private String bname; private String bauthor;

private String bversion; private String btype; private String bpublicshers;

private Date bpublictime; private double bprice; private int bcount;

@GenericGenerator(name = "generator", strategy = "increment")

@Id//主键注解

@GeneratedValue(generator = "generator")

@Column(name = "id", unique = true, nullable = false)

@Column(name = "name")//各属性列注解}

<?xml version="1.0" encoding="UTF-8"?>//xxx.hbm.xml文件

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >

<hibernate-mapping>//属性name为类名全路径 catalog为数据库名 table为表名

<class name="common.Book" catalog="food" table="book">

<id name="bid" column="id" type="integer">

<generator class="increment"></generator> </id>

<property name="bname" column="name" type="string"></property>

<property name="bauthor" column="author" type="string"></property>

<property name="bversion" column="version" type="string"></property>

<property name="bprice" column="price" type="double"></property>

<property name="bcount" column="count" type="integer"></property>

<!-- name(类中的名字 ) column(表里的名字) type(Hibernate的类型 ) 一般一个mapping配置一个类 -->

</class></hibernate-mapping>

<?xml version="1.0" encoding="UTF-8"?>//hibernate.cfg.xml文件，手动配置时要run as 中选2input for JET Transformation

<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd" >

<hibernate-configuration>

<session-factory>

<property name="connection.url">jdbc:mysql://localhost:3306/food </property>

<property name="myeclipse.connection.profile">com.mysql.jdbc.Driver</property>

<property name="connection.username">root</property>

<property name="connection.password"></property>

<property name="connection.driver\_class"> com.mysql.jdbc.Driver</property>

<property name="dialect">org.hibernate.dialect.MySQLDialect</property>

<property name="show\_sql"></property>//在控制台显示sql语句 /指定数据库的方言

<!-- 这里配置映射文件 两种形式-->

<mapping resource="common/Room.hbm.xml"/>

<mapping resource="common/Book.hbm.xml"/>

<mapping class=”common.Book”>//注解

</session-factory> </hibernate-configuration>

public class HibernateSessionFactory {

public static Session getSession() {

Configuration config=new Configuration().configure();

SessionFactory factory=config.buildSessionFactory();

return factory.openSession(); }}

测试类

package common;import java.util.List;import org.hibernate.Query;import org.hibernate.Session;

public class Test {

public static void main(String[] args) {

Session session=HibernateSessionFactory.getSession();//创建session

//创建query hql语句 session.createQuery("from 类名");

Query query = session.createQuery("from common.Room");

List<Room> list = query.list();// 返回数据

for (Room r : list) {// 遍历数据

System.out.print("rid:" + r.getRid() + " ");

System.out.print("rname:" + r.getRname() + " ");

System.out.println(); }}}

关系映射（多表间的映射）

Author类和Book类

public class Author implements java.io.Serializable {//省略getter和setter方法

private Integer id; private String name; private Set<Book> books;}

<hibernate-mapping>//author.hbm.xml

<class name="common.Author" table="author" catalog="food">

<id name="id" type="integer"><column name="id" /> <generator class="assigned" /></id>

<property name="name" type="string" column="name" /> </property>

<set name="books">

<key column="authorid"></key><one-to-many class="common.Book" /></set> </class>

</hibernate-mapping>

public class Book implements java.io.Serializable {//省略getter和setter方法

private Integer id; private String name;private Author author;private Integer authorid;}

<hibernate-mapping>//book.hbm.xml

<class name="common.Book" table="book" catalog="food">

<id name="id" type="integer"><column name="id" /> <generator class="assigned" /></id>

<property name="name" type="string" column="name" /></property>

//<property name="authorid" type="integer" column="authorid" /></property>

<many-to-one name="author" column="authorid" class="common.Author"></many-to-one>

</class></hibernate-mapping>

一对多映射

假设有两张表 book和author，实现需求：查看某一个作者的全部书籍信息，即一个作者VS多本书，hibernate实现一对多，当查询作者信息时，自动查询书籍信息

1 实现普通的ORM

2 在Author类中添加book类的集合

3 在author.hbm.xml中配置books集合

Query qq = ss.createQuery("from common.Author");

List<Author> list = qq.list(); System.out.println("一对多000000000000000");

for (Author r : list) {

System.out.println("作者名:" + r.getName());

Set<Book> books = r.getBooks();

for (Book b : books) {System.out.println(" 书名:" + b.getName() + " ");}}

多对一映射

需求：查询某一本书信息时，同时查询出其作者信息

1 实现普通的ORM

2 在Book类中添加Author类型的对象

3 在book.hbm.xml中配置author对象

一对一 映射

多对一的特殊情况，表a 存放了前一部分数据 表存放了后部分数据，只有一一对应才显示信息

假设一个作者只写一本书 在Author加字段Book b,生成getter和setter，配置文件修改：

<many-to-one name="author" column="authorid" class="common.Author" unique=”true”>

</many-to-one>

多对多映射

假设一个作者可写多本书，一本书也可有多个作者

一个作者VS 多个书籍 Author类中加Set<Book> books同时修改Author类其配置文件

一个书籍VS 多个作者 Book类中加Set<Author> authors 同时修改Book类其配置文件

<set name="books"><key column="authorid" ></key><many-to-many class="common.Book" /></set>

<set name="authors"><key column="authorid" ></key><many-to-many class="common.Author" /></set>

映射集合

<set></set>无序，不可重复

<list></list>有序，可重复

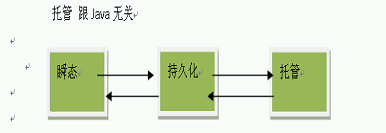
<list name="books">

<key column="authorid"></key> <list-index column="id"></list-index> 需索引 从零开始

<one-to-many class="common.Book" /> </list>

Bag 数据能重复且没有顺序，对应Java的Cellection接口，是set和list接口的共同父接口

## hibernate持久化

瞬态：新建的一瞬间

持久：增删改查等操作，跟数据库有关

托管：跟Java无关，数据库管理

在session中有以下方法实现持久化

Session ss = HibernateSessionFactory.getSession();//获得操作对象

Book bb = (Book) ss.load(Book.class, 111);// 查询

bb.setName("红楼梦");

Transaction ts = (Transaction) ss.beginTransaction();

ss.update(bb); ts.commit();

Book b1 = (Book) ss.load(Book.class, 112);

Transaction ts1 = (Transaction) ss.beginTransaction();

ss.delete(b1); ts1.commit();

Book b2 = new Book(1001, "西游记", 1003);// 添加 new对象后添加到数据库

Transaction ts2 = (Transaction) ss.beginTransaction();

ss.saveOrUpdate(b2); ts2.commit();

## HQL:hibernate的sql语句

HQL 没有insert into ... values 但是可以写 insert into ... select 相当于复制粘贴

1.HQL查询

查询全部列

Query query = session.createQuery("from com.niit.model.Room where id >1010");

List<Room> list = query.list(); from Java类名 相当于 select \* from ...

查询一列

Query query = session.createQuery("select name from com.niit.model.Room");

List<String> nameList = query.list();

查询部分列

Query query1 = session.createQuery("select id,name from com.niit.model.Room");

List<Object> list = query1.list(); // 虽然是Object类型 但是 id,name还是存在里面

for(Object o : list){// o相当于 Room

// Object[] 该类中属性的数组 也就是 id和name组成的数据

Object[] properties = (Object[])o;// 将对象转换为属性数组

System.out.println("id:"+properties[0]+"--name:"+properties[1]);

}

带参数的SQL语句where

Query query4 = session.createQuery( // :rid 和 :rnumber 两个变量

"from com.niit.model.Room where id > :rid and bedsnumber = :rnumber");

query4.setInteger("rid", 1010); query4.setInteger("rnumber", 6);

或 Query query4 = session.createQuery(

"from com.niit.model.Room where id > ? and bedsnumber = ?");

query4.setParameter(0, 1010); query4.setParameter(1, 6);

2.HQL更新和删除

①执行更新系的HQL语句需要开启事务并提交

Transaction ts = ss.beginTransaction();

Query query5 = ss.createQuery(

"update com.niit.model.Room set name=? where id=?");

query5.setParameter(0, "男生宿舍"); query5.setParameter(1, 1008);

int count = query5.executeUpdate();// 更新

Query query6 = ss.createQuery("delete from com.niit.model.Room where id =?");

query6.setParameter(0, 1014);

int delCount = query6.executeUpdate();//删除

ts.commit();

3执行本地SQL语句

Hibernate执行原生的SQL语句即MySQL语句SQLQuery sqlQuery = session.createSQLQuery(MySQL语句);

①查询全部列

SQLQuery sqlQuery = session.createSQLQuery("select \* from room");//返回结果集

sqlQuery.addEntity(Room.class);// 转换为Java类，相当于 com.niit.model.Room

List<Room> list = sqlQuery.list();

②查询部分列

sqlQuery = session.createSQLQuery("select id,name from room");

sqlQuery.setResultTransformer(Transformers.aliasToBean(RoomBean.class));

List<RoomBean> list2 = sqlQuery.list();

for(RoomBean r : list2){ System.out.println(r.getName()); }

class RoomBean{

private int id; private String name;//查询字段及其getter和setter方法

public RoomBean(){}//无参和全参的构造方法

public RoomBean(int id, String name) { super(); this.id = id; this.name = name; } }

4 Hibernate Criteria 条件查询

实现无sql增删改查，Restrictions 提供方法实现不同条件的查询

1 比较运算 模糊查询 范围查询等

Eq即等于= gt即大于> ge即大于等于>= lt即小于<

le即小于等于<= ne即不等于!= Like between在某范围之内的数据

isNull/isNotNull 某列是否为空 isEmpty/isNotEmpty 某列是否为空的值

Eg Criteria c2 = ss.createCriteria(Room.class); c2.add(Restrictions.eq("id", 1005));//id=1005

Room r = (Room) c2.uniqueResult();// 返回单行数据

2 实现or连接

Criteria c5 = ss.createCriteria("common.Room"); //多条件时默认是AND连接

Criterion id = Restrictions.gt("id", 1010);//条件1

Criterion address = Restrictions.eq("address", "上海市");//条件2

c5.add(Restrictions.or(id, address));//or连接

3 排序、聚合函数max/min/avg/sum/count

c5.addOrder(Order.desc("id")); c6.setProjection(Projections.max("bedsnumber"));

## 事务transaction

一、默认的JDBC的事务管理,原理：关闭了数据库的自动提交，然后改为手动提交

Transaction ts = ss.beginTransaction();...........ts.commit();

二、JTA hibernate.cfg.xml文件中配置

<property name="hibernate.transaction.factory\_class">org.hibernate.transaction.JTAtransaction</property>

多个事务件会产生的问题

1 脏读 一个事务可更新另一事务没有提交的数据

2 丢失更新

3 不可重复读 一个事务在不停执行同一个sql时有其他事务对查询的数据进行了update操作

4 幻读 一个事务在不停执行同一个sql语句，但有其他事务进行了insert或delete操作

解决：使用事务的隔离级别，默认级别为0，它们的级别为，1,2,4,8

hibernate.cfg.xml文件中设置级别

<property name="hibernate.connection.isolation">1</property>

脏读 丢失更新 不可重复读 幻读（是否会出现）

未提交读级别 是 是 是 是

提交读取级别 否 是 是 是

重复读级别 否 否 否 是

序列化级别 否 否 否 否

# Mybatis

1.开源框架，官网GitHub下载jar包 mybatis-3.4.2.jar mybatis-3.4.4.pdf 官方的使用指南，在线手册：http://www.mybatis.org/mybatis-3/zh/

2.通过XML或注解方式映射sql语句和Java类，需手动写sql语句

使用方式一：手动添加entry、mapper文件

a. 加入jar包

b. mybatis配置文件即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><!-- 根节点以下配置的文件是关于整个框架 -->

<environments default="development"><!-- 配置使用环境,有开发、测试和商用环境 -->

<environment id="development">

<!-- 指定mybatis事务的管理模式jdbc模式和 managed模式（没什么用） -->

<transactionManager type="JDBC"></transactionManager>

<!-- pooled使用连接池，UNPOOLED不使用连接池时每次连接都需打开新的连接，使用完毕还需关闭连接 -->

<dataSource type="POOLED">

<property name="driver" value="com.mysql.jdbc.Driver"></property>

<property name="url" value="jdbc:mysql://localhost:3306/food"></property>

<property name="username" value="root"></property>

<property name="password" value=""></property>

</dataSource>

</environment>

</environments>

<mappers><!-- 同一个表的映射 最好就使用一种映射方式 -->

<!--<mapper resource="common/RoomMapper.xml" /> -->//XML方式

<mapper class="common.RoomMapper" />//接口添加注解方式

</mappers>

</configuration>

c. 实现映射

①创建 model 层的 实体类Xxx

②实现SQL语句映射文件 XxxMapper.xml或XxxMapper接口注解

<?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="common.model.RoomMapper">

<select id="getRoomById" parameterType="int" resultType="common.model.Room">

select \* from room where id=#{id}

</select>//update、insert、delete等

</mapper>

使用接口的方式实现映射

1 创建抽象方法 2在方法签名上写注释 使用sql语句

3在mybatis的config.xml 中添加配置

最好方案：xml映射和java 接口映射相结合

① RoomMapper.xml 和 RoomMapper.java 并且必须在同一个包

② RoomMapper.xml中的Sql语句Id 必须 跟 RoomMapper.java 中的方法名一样，RoomMapper.xml中的名称空间namespace须和 RoomMapper.java 中的接口的全路径一致

③ 在mybatis-config.xml中 只需要配置RoomMapper.xml即可

④ 在执行SQL语句时候 直接调用RoomMapper.java中的方法

d. 在mybatis-config.xml文件中 添加 映射文件

e. 使用Resources 读入mybatis-config.xml配置文件、创建SqlSessionFactory（封装成一个且是单例模式mybatisUtil类）

package common;//记得导包

public class mybatisUtil {//单例模式打开链接，因在一个项目中SqlSessionFactory只要一个，不需要多次读取配置文件

public static String path = "config/mybatis-config.xml";

private static SqlSessionFactory factory = null;

private mybatisUtil() {}// 构造方法私有化

public static SqlSessionFactory getSessionFactory() {

if (factory == null) {

try { InputStream ins = Resources.getResourceAsStream(path);

SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();

factory = builder.build(ins);

} catch (IOException e) { e.printStackTrace(); }}

return factory; }

public static SqlSession getSqlSession() {return getSessionFactory().openSession();}}

f . 创建SqlSession， 执行SQL语句 返回结果

public class Test {//测试类

public static void main(String[] args) {

SqlSession ss = mybatisUtil.getSqlSession();// 创建Session

RoomMapper mm = ss.getMapper(RoomMapper.class);

Room room1 = mm.getById(1008);

System.out.println("宿舍名：" + room1.getName() + "---宿舍地址："

+ room1.getAddress());

}

使用方式二：使用mybatis generator 生成entity，mapper文件

① 使用命令行

② 使用Mybatisgenerator eclipse 插件

A. 下载插件

B. 解压之后找到generator-master\eclipse\UpdateSite中找到features和plugin

C. 找到MyEclipse的安装路径中 找到features和plugin,将上一步找到的文件夹中的所有文件拷贝到对应文件夹

D 重启MyEclipse

E 创建mybatis generator 配置文件 new--other--mybatis generator config file

F 在项目目录中的generatorConfig.xml文件上右键 mybatis generator生成菜单

generatorConfig.xml 配置文件

1. <classPathEntry location="数据库驱动jar包的全路径" 注意路径中不能有中文

2. context中的配置 指定生成的路径以及生成的目标数据库和表等信息

<jdbcConnection>说明 数据库的连接信息

<javaModelGenerator> 实体类的生成策略 Entity getter和setter类

targetPackage 生成的java类的全路径

targetProject 目标项目名称

<sqlMapGenerator> 生成的映射的XML文件 写着SQL的xml文件

<javaClientGenerator> 生成的映射的Java文件 写着方法的接口

<table> 生成目标表的配置 schema 数据库 tableName 表名

## mybatis动态sql

<if> 当满足某个条件时就会拼接到地址栏

<where>若where中所有都不满足则不生成where子句，否则会生成where子句，并且会自动判断and 或者or决定是否要拼接and或or语句

<set> update语句中的set子句，判断一下哪个字段需要更新即拼接，也会取消掉无用逗号

<foreach>

<?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="common.RoomMapper"><!-- 假设name是固定条件，address是可选条件 -->

<select id="getRoomByNameAndAddress" resultType="common.Room">

select \* from room where name=#{name}

<if test="address!=null"> and address=#{address}</if>

</select>

<select id="getRoomInfo" parameterType="int" resultType="common.Room">

select \* from room

<where><!-- 若where中所有都不满足则不生成where子句，否则会生成where -->

<if test="id!=null">id=#{id}</if>

</where>

</select>

<select id="getRoomByExample" parameterType="common.Room" resultType="common.Room">

select \* from room

<where><!-- 多分支选择 -->

<choose>

<when test="name!=null and address==null">and name=#{name}</when>

<when test="name==null and address!=null">and address=#{address}</when>

<otherwise>and name=#{name} and address=#{address}</otherwise>

</choose>

</where>

</select>

<update id="updateRoomById" parameterType="common.Room">

update room

<set><if test="name!=null">name=#{name},</if>

<if test="address!=null">address=#{address},</if>

<if test="gender!=null">gender=#{gender}</if>

</set>

where id=#{id}

</update>

<select id="getRoomByAddress" resultType="Room">

select \* from room where address in

<foreach collection="list" index="index" item="item" open="("

separator="," close=")">#{item}</foreach>

</select>

</mapper>

package common;//导包

public interface RoomMapper {

// 接口方式做映射即将每一个数据库的操作都写成一个抽象方法

public List<Room> getRoomByNameAndAddress(@Param("name") String roomName,

@Param("address") String roomAddress);

public List<Room> getRoomInfo(@Param("id") int roomid);

public List<Room> getRoomByExample(@Param("name") String roomName,

@Param("address") String roomAddress);

public int updateRoomById(Room room);

public List<Room> getRoomByAddress(List<String> addressList);}

package common;//导包

public class Test {

public static void main(String[] args) {

SqlSession ss=mybatisUtil.getSqlSession();

RoomMapper mm = ss.getMapper(RoomMapper.class);

List<Room> list=mm.getRoomByExample("沁园",null);

List<Room> list=mm.getRoomByNameAndAddress("沁园",null);

System.out.println("宿舍名为沁园的数据条数为："+list.size());

List<Room> list2=mm.getRoomByNameAndAddress("沁园","上海市");

System.out.println("宿舍名为沁园且地址为上海市的数据条数为："+list2.size());

Room rr=new Room(); rr.setId(1011); rr.setName("火星村");

int updateCount=mm.updateRoomById(rr);

if(updateCount>0){System.out.println("id为1011的数据更新成功"); }

List<String> list4=new ArrayList<String>();

list4.add("上海市"); list4.add("贵阳市"); list4.add("海口市");

List<Room> list5=mm.getRoomByAddress(list4);

System.out.println("地址为上海、贵阳和海口的数据有："+list5.size());

ss.commit();// 需手动提交事务，因mybatis 默认不会自动提交事务

}

}

使用别名

mybatis允许定义别名来代替类的全路径 common.Room-->Room(别名）

在整个项目中所要使用common.Room字符串时都可用Room来代替

配置位置：mybatis.config.xml 文件中，在<environment>的上面

<typeAliases><!-- Room是配置文件中命名 的别名 -->

<typeAlias type="common.Room" alias="Room"/>

<typeAlias type="common.User" alias="User"/> .......

</typeAliases>

mybatis-config.xml

SqlSessionFactory

Builder

Resources

Mapper.xml

environment

Mapper.java

1 . SqlSessionFactory 不需要反复创建，一个项目仅需要一个实例(单例模式)

2. SqlSession 相当于 JDBC Connection 对象

执行任何SQL语句之前 必定需要创建一个SqlSession

## 实现关系映射

1 一对多

eg一个作者VS多本书 查询作者信息 就可以关联出来所有的书籍

a. Author类中添加 List<Book> books;

b.查询结果使用resultMap <collection> 配置 Book类

<mapper namespace="common.model.AuthorMapper">

<!-- 自定义结果集，存放select语句执行的结果 -->

<resultMap type="common.model.Author" id="authorBean">

<result property="id" column="id" />

<result property="name" column="name" />

<!-- cellection集合来装book -->

<collection property="books" ofType="common.model.Book"

column="authorid">

<result property="id" column="bid" />

<result property="name" column="bname" />

</collection>

</resultMap>

<select id="getAuthorById" parameterType="int" resultMap="authorBean">

select a.\*,b.id as bid,b.name as bname from author as a,book as b

where b.authorid=a.id and a.id=#{id}

</select>

</mapper>

System.out.println("------一对多--------");

AuthorMapper am=session.getMapper(AuthorMapper.class);

List<Author> alist=am.getAuthorById(1002);

Author a=alist.get(0);

List<Book> books=a.getBooks();

System.out.println("作者："+a.getName()+" 书籍有：");

for(Book b:books){ System.out.println(b.getId()+":"+b.getName()); }

2 多对一

多本书VS一个作者, 查询书籍信息 就可以对应返回 作者信息

a. Book类中添加 Author author ,生成setter和getter方法（注释掉authorid）

b.使用resultMap <association>

<mapper namespace="common.model.BookMapper">

<resultMap type="common.model.Book" id="BookBean">

<result property="id" column="bid" />

<result property="name" column="bname" />

<association property="author" javaType="common.model.Author">

<result property="id" column="id" />

<result property="name" column="name" />

</association>

</resultMap>

<select id="getBookById" parameterType="int" resultMap="BookBean">

select a.\*,b.id as bid,b.name as bname from author as a,book as b

where b.authorid=a.id and b.id=#{id}

</select>

</mapper>

System.out.println("\n------多对一--------");//测试

BookMapper bm = session.getMapper(BookMapper.class);

List<Book> book = bm.getBookById(113);

for (Book b : book) {

System.out.println("书名：" + bb.getName() + " 作者：" + b.getAuthor().getName());}

# Spring框架

概念：微服务、Restful(通过URL实现共享服务)

Spring核心framework a控制反转 b依赖注入 cAOP

## Spring框架的架构

核心模块 容器bean工厂

Dao模块 实现hibernate、mybatis等的标准化，将数据库连接交给spring

ORM模块 将映射关系中的类也交给spring管理，例如spring管理事务

AOP模块

应用程序上下文context 读取完的配置文件 就存在context里

Web上下文 集成其他框架

MVC模块（spring MVC）

<?xml version="1.0" encoding="UTF-8"?>//applicationContext.xml

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.1.xsd">

<!-- id即spring需要用的类的ID，整个文件中应是唯一的 class即bean对应的全路径

property配置类中字段 -->

<bean id="stuBean" class="common.Stu">

<property name="name" value="盛淮南"></property>

</bean>

</beans>

public class Stu {//实体类

private String name;//生成setter方法

public void sayHello() { System.out.println("你好，" + name);}}//普通方法

package common;

public class Test {

public static void main(String[] args) {//测试类

// 读取spring配置文件

// ApplicationContext 应用程序上下文存放了读取到的配置文件（所有的bean定义）

// ClassPath 编译完成的.class文件所在路径

ApplicationContext context = new ClassPathXmlApplicationContext(

"applicationContext.xml");

// Spring 通过应用程序上下文创建Student对象

// applicationContext.xml第一的得bean的id

Stu s1 = (Stu) context.getBean("stuBean");

s1.sayHello(); }}

a控制反转 类将控制权转移即交给中介，类就不自己实例化，spring类似中介的作用，它关注类的控制权，它在合适的时候帮忙实例化！

b依赖注入DI 控制反转思想在代码中的实现

A 手动方式实现依赖注入

1使用setter方法注入

实体类中将被注入的变量或对象生成setter方法

在配置文件中添加配置，使用property属性配置被注入的变量或对象

2使用构造方法注入

将被注入变量会对象作为构造方法的参数

在配置文件中添加配置,使用constructor属性配置被注入的变量或对象

B 自动注入 autowire="default" 类似在Java代码字段上写@autowired

byName根据属性名称自动实现装配，若一个bean的id跟主程序的属性名称一致则进行注入

byType根据属性类型自动实现装配，若一个bean的class跟主程序的属性类型一致则进行注入

Constructor根据主程序构造方法的参数实现的自动注入

注：@Autowired 相当于 在配置文件中写的 autowired=""，但是要是有 @Autowired注解实现自动装配的话 必须在spring配置文件中添加以下代码<context:annotation-config/>

注：若在applicationContext.xml中添加bean，若要使用bean配置则须使用spring方式来创建类的实例

注入类时Spring认为相关类都必须当前文件中配置为一个bean，使用ref属性引用被注入的类对象

Spring和hibernate整合

创建项目--->添加spring的jar包，注意勾选spring persistence包

-->添加hibernate的jar包

Spring 提供的hibernate模板类，该类集成一些关于数据库的操作

HibernateTemplate template=new HibernateTemplate();

Mybatis和spring整合

1配置环境 准备jar包

Spring-4.1-xxx.jar mybatis-3.4.2.jar

mysql-connector mybatis-spring-1.3.0.jar

2 创建mybatis-config.xml文件

3 创建 Room.java(model包)

RoomMapper.java、RoomMapper.xml (dao包)

4 在RoomMapper.java和RoomMapper.xml配置

public interface RoomMapper { public List<Room> getAll();}

<mapper namespace="common.niit.dao.RoomMapper">//名称空间为mapper接口的全路径

<select id="getAll" resultType="common.model.Room">

select \* from room

</select>

</mapper>

5 在mybatis-config.xml文件添加配置

<configuration><!-- environments交给spring管理 typeAlias可根据需要写 -->

<mappers>

<mapper resource="com/niit/dao/RoomMapper.xml"/>

</mappers>

</configuration>

6 在applicationContext.xml文件配置

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

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.1.xsd">

<bean id="dataSource"class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver"></property>

<property name="url" value="jdbc:mysql://localhost:3306/food"></property>

<property name="username" value="root"></property>

<property name="password" value=""></property>

</bean>

<!-- 读取mybatis-config.xml文件 创建SqlSessionFactory -->

<bean id="sessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<property name="dataSource" ref="dataSource"></property><!-- 数据源 -->

<!-- mybatis配置文件的路径 -->

<property name="configLocation" value="classpath:mybatis-config.xml"></property> </bean>

<!-- Mapper工厂用于创建mapper代理类 -->

<bean id="roomMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">

<!-- mapper接口的路径 和sqlsessionfactory -->

<property name="mapperInterface" value="com.niit.dao.RoomMapper"></property> <property name="sqlSessionFactory" ref="sessionFactory"></property>

</bean>或

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer"> <!--自动扫描Mapper文件 -->

<property name="basePackage" value="com.niit.dao"></property>

<property name="sqlSessionFactory" ref="sessionFactory"></property>

</bean>

</beans>

## AOP面向方面的编程

开启、提交事务 输出日志信息

AOP 面向方面的编程

通知： 次要任务，例如开始事务，输出日志

如果设置了通知的条件，只会通知/影响满足条件的方法。

如果没有设置通知的条件，将会影响所有的方法。

通知的种类： 1 在切入方法执行之前执行 2 在切入点方法执行之后 3 拦截器

连接点：主程序中可能被切到的地方（一些方法）

比如 select 方法 insert方法 update方法 delete方法

切入点：最终切入的条件，比如只有当方法名以 update开头的时候才会调用 次要任务

目标 : 主程序所在的类

代理 代理临时类 中间变量

-----------------------------------------------------------------------------------------------

主任务 ： Student类 有一个方法 print

次要任务: 打印一句话

通知: Advice类 连接主任务 和 次要任务

因为没有指定通知的条件，其实通知所有的方法

package com.niit.advice;import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class BeforeAdvice implements MethodBeforeAdvice{// 前置通知 在切入点执行之前 执行

/\*\* 在切入点执行之前 执行该方法

\* @param Method method 当前被通知影响到的方法

\* @param Object[] args 该方法的参数列表

\* @param Object target 主程序所在类的实例 \*/

public void before(Method method, Object[] args, Object target)throws Throwable {

// 模拟的次要任务 被动触发的

System.out.println("前置通知中代码被执行！拦截到主程序方法的参数是:"+(String)args[0]);

}}

package com.niit.advice;import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;

public class AfterAdvice implements AfterReturningAdvice {// 后置通知 切入点方法执行之后 再执行

public void afterReturning(Object arg0, Method arg1, Object[] arg2,

Object arg3) throws Throwable {

// 模拟的次要任务 在切入点执行之后 执行

System.out.println(arg1.getName()+"已经执行完毕了");

}}

package com.niit.java;// 主程序对应的接口

public interface IStudent {

public void print(String name); public void execute(String name);}

package com.niit.java;

public class Student implements IStudent{// 模拟的主程序

public void print(String name){// 连接点 有可能被 次要任务影响

System.out.println("print方法中打印的名字是:"+name);}

public void execute(String name) {// 连接点 有可能被 次要任务影响

System.out.println("execute方法中打印的名字是:"+name);}

}

package com.niit.java;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("applicationContext.xml");

// 因为使用了代理类 所以返回值 是 接口类型

// student 是 配置文件中 代理工厂 bean的 id

IStudent student = (IStudent)context.getBean("student");

student.print("niit");

student.execute("hanliang");

}}

applicationContext.xml

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

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.1.xsd">

<bean id="target" class="com.niit.java.Student"></bean> <!-- 配置 目标类 -->

<bean id="beforeAdvice" class="com.niit.advice.BeforeAdvice"></bean> <!-- 配置通知 -->

<bean id="afterAdvice" class="com.niit.advice.AfterAdvice"></bean>

<!-- 配置代理工厂 使用工厂模式 创建Student类的代理类 -->

<bean id="student" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target" ref="target"></property> <!-- 1.注入 目标类 -->

<property name="interceptorNames"> <!-- 2.注入通知列表 -->

<list> <value>beforeAdvice</value> <value>afterAdvice</value> </list>

</property>

<!-- 3.注入接口 根据该接口 来创建主程序的代理类 -->

<property name="proxyInterfaces" value="com.niit.java.IStudent"></property>

</bean>

</beans>

订票业务

主任务： 订票 一个接口ITicket 一个类 Ticket

次要任务： 一个类 CheckTicket

1. 卖票之前 要检查余票是否充足 方法一 checkBalance

2. 买票之后 更新剩余票数 方法二 updateBalance

通知：TicketAdvice implements MethodBeforeAdvice, AfterReturningAdvice

1. 前置通知 调用 余票检查程序

2. 后置通知 调用 余票更新程序

Spring小结

① 原理 容器 bean工厂 一切都是bean "大管家"

② 依赖注入

a. setter注入

b.构造注入

c.注入

③ AOP

④ 整合Hibernate , Mybatis

# SpringMVC

http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/index.html

spring框架的一部分，轻量级Model 数据操作View jsp界面Controller 控制器

基本概念

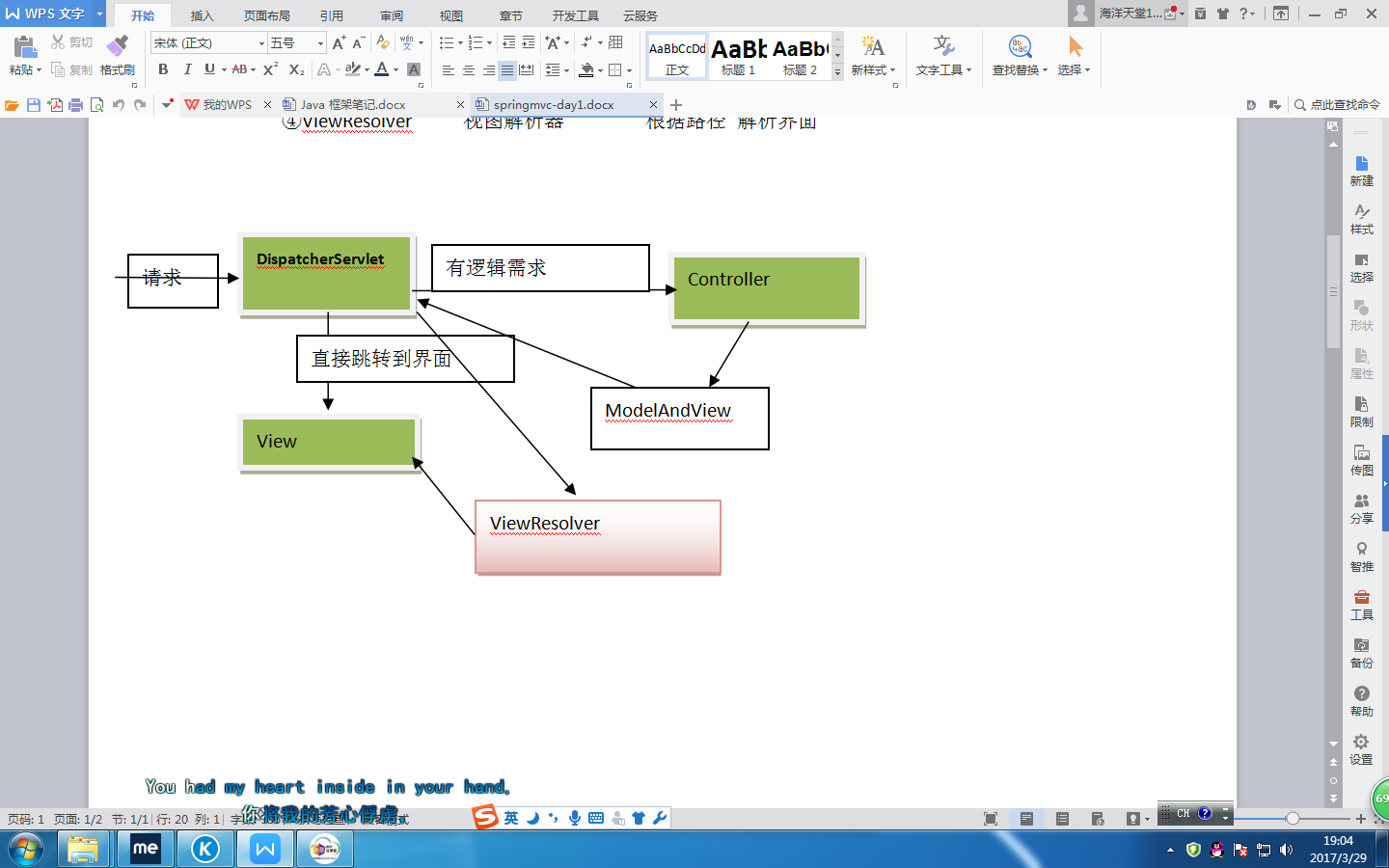
1. DispatcherServlet 请求分发Servlet 拦截请求 分发到不同的Controller

需要在web.xml文件 添加配置

2. Controller 就相当于servlet 自己定义的一个类

需要在springmvc配置文件中配置Controller路径，新建SpringMVC配置文件dispacher-servlet.xml， 放在WEB-INF文件夹下

3. 视图解析器 ViewResolver 需要springmvc配置文件dispacher-servlet.xml中配置视图解析器，需要两个参数1. 界面的路径 2. 匹配的规则



执行顺序

①加载web.xml配置文件

②加载dispatcher-servlet配置文件

如果启动Tomcat时候就报错 ，就要检查上面两个配置文件

③ 发出请求(输入URL; 超链接 ; 提交 form) 将URL交给 DispathcherServlet

运行方式：① 在地址栏里面输入以下地址http://localhost:8080/项目名字/hello.do

② 在index.jsp里面 添加 以下代码 <a href="hello.do">打开hello</a>

**③** 运行index.jsp 然后单击超链接

④ DispatcherServlet 判断如果是.do结尾的路径 就会去 读取 dispatcher-servlet.xml中的

关于 Controller的配置 比较 URL 和 Controller的 name属性 如果一致 就执行Controller中的代码

⑤ Controller中 返回数据，设置ViewName 调用 dispatcher-servlet.xml中配置的视图解析器 根据 配置的 prefix 和 suffix 解析出来 要显示的界面 并显示

Spring MVC注解

写法：①@Controller 将类变成SpringMVC的控制器 无需 实现Contronller接口

②@RequestMapping 用来说明URL

<?xml version="1.0" encoding="UTF-8"?>web.xml文件

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" version="3.0">

<display-name>springmvcDetails</display-name>

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

<context-param>

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

<param-value>classpath:applicationContext.xml</param-value>

</context-param>

<servlet><!-- 配置DispatcherServlet -->

<servlet-name>dispacher</servlet-name><!-- 自定义名字 -->

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

</servlet>

<servlet-mapping>

<servlet-name>dispacher</servlet-name><!-- 自定义名字跟上面的一致 -->

<!-- 表示只有以.do结尾的URL才会被DispatcherServlet拦截 ,有\*.jsp \*.html或/ -->

<url-pattern>\*.do</url-pattern><!-- /表示拦截全部路径 -->

</servlet-mapping>

</web-app>

---------------------------------------------------------------------------------------------------------------------------------

<?xml version="1.0" encoding="UTF-8"?>//dispacher-servlet.xml

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"

xmlns:mvc="http://www.springframework.org/schema/mvc"

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-4.1.xsd

http://www.springframework.org/schema/mvc [http://www.springframework.org/schema/mvc/spring-mvc-4.1.xsd">](http://www.springframework.org/schema/mvc/spring-mvc-4.1.xsd%22%3e)

<!-- 让springmvc 自动扫描Controller 按照base-package所写的路径去 搜索@Controller注解 --> <context:component-scan base-package="springmvcDetails"></context:component-scan>

<!-- 配置controller 告诉springMVC helloController是控制器 指定了helloController的路径是hello.do -->

<bean name="/hello.do" class="com.niit.controller.HelloController"></bean>

<!-- 视图解析器 将接收到的一个View 的name 根据配置文件中的prefix 和suffix生成界面的实际路径 例如viewName是list，实际生成路径就是/WEB-INF/list.jsp -->

<bean id="viewResolver"

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

<property name="prefix" value="/WEB-INF/"></property>1. 界面的路径

<property name="suffix" value=".jsp"></property>2. 匹配的规则

</bean></beans>

---------------------------------------------------------------------------------------------------------------------------------

package springmvcDetails; import java.util.ArrayList; import java.util.List;

import javax.servlet.http.HttpSession;import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.servlet.ModelAndView;

@Controller//将类变成控制器

@RequestMapping("/room")// 表示想要访问当前controller 需要配置的URL

public class RoomController {//控制器的两种形式一，对应两种配置方式

@RequestMapping("/hello.do")//注解用以区分每个方法

public ModelAndView sayhello(HttpSession session) {

ModelAndView mv = new ModelAndView();

List list = new ArrayList<String>();

list.add("张三"); list.add("李四"); list.add("王老五"); mv.addObject("list", list);

mv.setViewName("list");// 视图解析器为/WEB-INF/list.jsp

return mv;}}

package com.niit.controller;

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

import org.springframework.web.servlet.ModelAndView;

import org.springframework.web.servlet.mvc.Controller;

public class HelloController implements Controller {//控制器的两种形式二

// spring mvc 控制器根据请求的路径做出响应

// 2.DispatcherServlet拦截请求中url 然后和HelloController的路径进行匹配

// 如果相同就执行handleRequest方法

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

ModelAndView mvc = new ModelAndView();

mvc.addObject("message", "hello,it is me!");// 定义了变量，输出

mvc.setViewName("hello");// 跳转到hello.jsp

return mvc;

}}

---------------------------------------------------------------------------------------------------------------------------------

<%@ page language="java" import="java.util.\*" pageEncoding="UTF-8"%>//index.jsp发送请求<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html> <head> <title>My JSP 'index.jsp' starting page</title> </head>

<body> <a href="room/hello.do">调用sayhello方法</a> <br> </body>

</html>

---------------------------------------------------------------------------------------------------------------------------------

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>//list.jsp接收结果

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html><head><meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Insert title here</title></head>

<body> <c:forEach items="${list}" var="v">${v}</c:forEach> </body></html>

404 错误 检查以下的代码：

① web.xml DispatcherServlet url-pattern中写的是不是 \*.do

② 检查 dispatcher-servlet.xml 里面 Controller的 name属性

**是**否写了，或者是否跟输入的路径一致

**③** 极少数情况是 配置文件写错了

## Controller中方法的写法

A @RequestMapping 类中指定一级URL，方法指定二级URL

指定响应的请求方式post或get，method=RequestMethod.POST，只能响应post方式提交的请求，安全性考虑，不会响应其他方式的请求

指定URL中的参数，@RequestMapping(value="/testParam.do",params={"id","name"})

B 方法的参数 接收请求中数据的方式 界面--->controller

直接参数（参数名跟request中数据变量名同名，可直接接收request中的数据）

直接用Java类接收（POJO setter和getter）

@Controller @RequestMapping("/room")

@RequestMapping(value = "/testParam1.do", params = { "id", "name" })//RoomController

public String testParam1(int id, String name, HttpServletRequest request) {

request.setAttribute("id", id);// 请求中的数据可直接用同名变量接收

request.setAttribute("name", name);

return "list";}

<a href="room/testParam1.do?id=1001&&name=susu">测试提交中需传参数，带正确参数时</a>

传入的id: ${id} <br /> 传入的name: ${name}//发送请求--controller处理->接收数据

// 请求中的数据可直接用类接收,只要请求中参数名和类User中的字段名一致，则可直接将数据存入User

@RequestMapping(value = "/testPro.do")

public String testPro(User user, HttpServletRequest request) {

request.setAttribute("user", user); return "list";} <br />

<form action="room/testPro.do" method="post">

id:<input name="id" type="text"> name:<input name="name" type="text"><br>

<input type="submit" value="测试用类接收请求传参的数据" />

</form>

用类接收的数据： id:${user.id}<br> name:${user.name}

C 返回数据的方式 controller-->界面

HttpServletRequest request.setAttribute(key, value);

HttpSession session.setAttribute(key, value);

将Map<String ,Object>作为参数

将model接口作为参数

// 只需将数据存入Map，springMVC将数据存入request，带到页面

@RequestMapping(value = "/testMap.do")

public String testMap(Map<String, Object> map) {

map.put("msg", "Map也可将数据带到页面去"); return "list"; }

// 只需将数据存入Model，springMVC将数据存入request，带到页面

@RequestMapping(value = "/testModel.do")

public String testModel(Model model) {

model.addAttribute("msg", "Model也可将数据带到页面去"); return "list";}

D 方法的返回值类型 controller-->界面

Model and View String Map<String ,Object> 无返回值void

// 当Map作为返回值类型时相当于没有指定返回的页面，viewResolver会将按照请求的路径解析界面，例如WEB-INF/room/testMaps.jsp

@RequestMapping(value = "/testMaps.do")

public Map<String, Object> testMap() {

Map<String, Object> map = new HashMap<String, Object>();

map.put("msg", "map作为返回值，可将数据带到页面去");

return map; }

// 当返回值类型为空时相当于没有指定返回的页面，viewResolver会将按照请求的路径解析界面，例如WEB-INF/room/testVoid.jsp

@RequestMapping(value = "/testVoid.do")

public void testVoid() { System.out.println("返回值为空时的方法"); }

过滤器

字符集过滤器

每次请求之前都拦截请求 强制设置字符集Filter打开请求进行过滤，在web.xml中进行配置

<filter><!-- 1 定义一个过滤器 --><!-- 配置字符集过滤器 解决乱码springMVC自带的 -->

<filter-name>encodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param><!-- 初始化参数 就是当过滤类 被创建的时候 就自带的一些参数 -->

<param-name>encoding</param-name><!-- 固定的名字 -->

<param-value>utf-8</param-value>

</init-param>

<init-param>

<param-name>forceEncoding</param-name><!-- 固定的名字 -->

<param-value>true</param-value>

</init-param>

</filter>

<filter-mapping><!-- mapping 说明 过滤器 过滤请求的条件 -->

<filter-name>encodingFilter</filter-name> <url-pattern>/\*</url-pattern>

</filter-mapping>

Session过滤器

登录之后将用户名存在session中，过滤器在每次请求之前session是否还有用户名，若没有不能进行任何操作，跳转到登录页

# Springboot

## controller的使用

@Controller 处理http请求 需配合模板使用 不推荐因为使用模板会降低性能且现开发模式前后端分离，后端提供restAPI即可

Pom.xml中添加模板依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

Resource文件夹的template夹下建index.html

Controller中 return “index”

@RestController 等同于@ResponseBody 配合@Controller

@RequestMapping 配置URL映射

@RequestMapping(value = {"/higirl",”/hi”}, method = RequestMethod.GET)

@PathVariable 获取URL中的数据

@PathVariable ("id") 请求路径为http://localhost:8080/girl/higirl/say/123

@RequestParam 获取请求参数的值

@RequestParam (value="id",required=false,defaultValue="11")

请求路径为http://localhost:8080/girl/higirl/speak?id=456

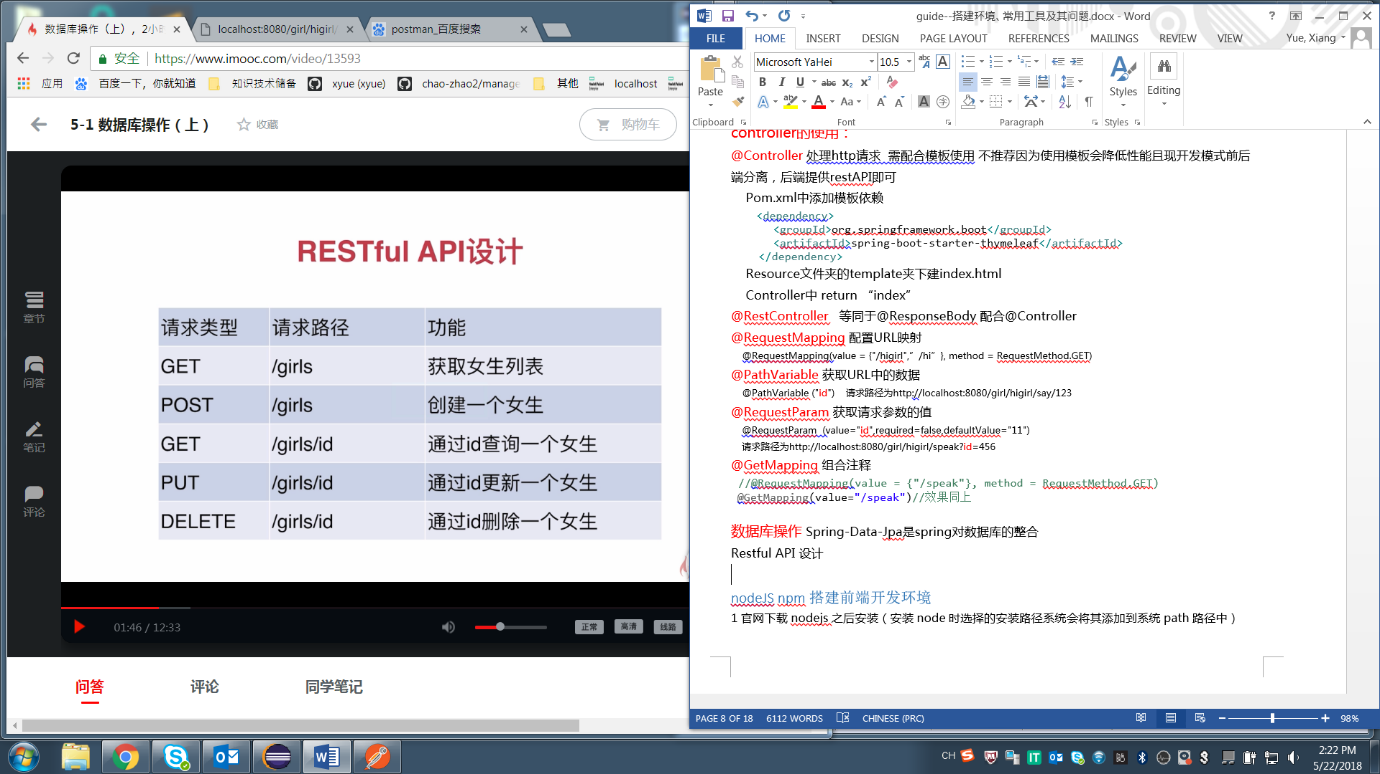
@GetMapping 组合注释如POSTMapping PUTMapping等

//@RequestMapping(value = {"/speak"}, method = RequestMethod.GET)

@GetMapping(value="/speak")//效果同上

## 数据库操作

Spring-Data-Jpa是spring对数据库的整合

1 添加数据库依赖（pom.xml）

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

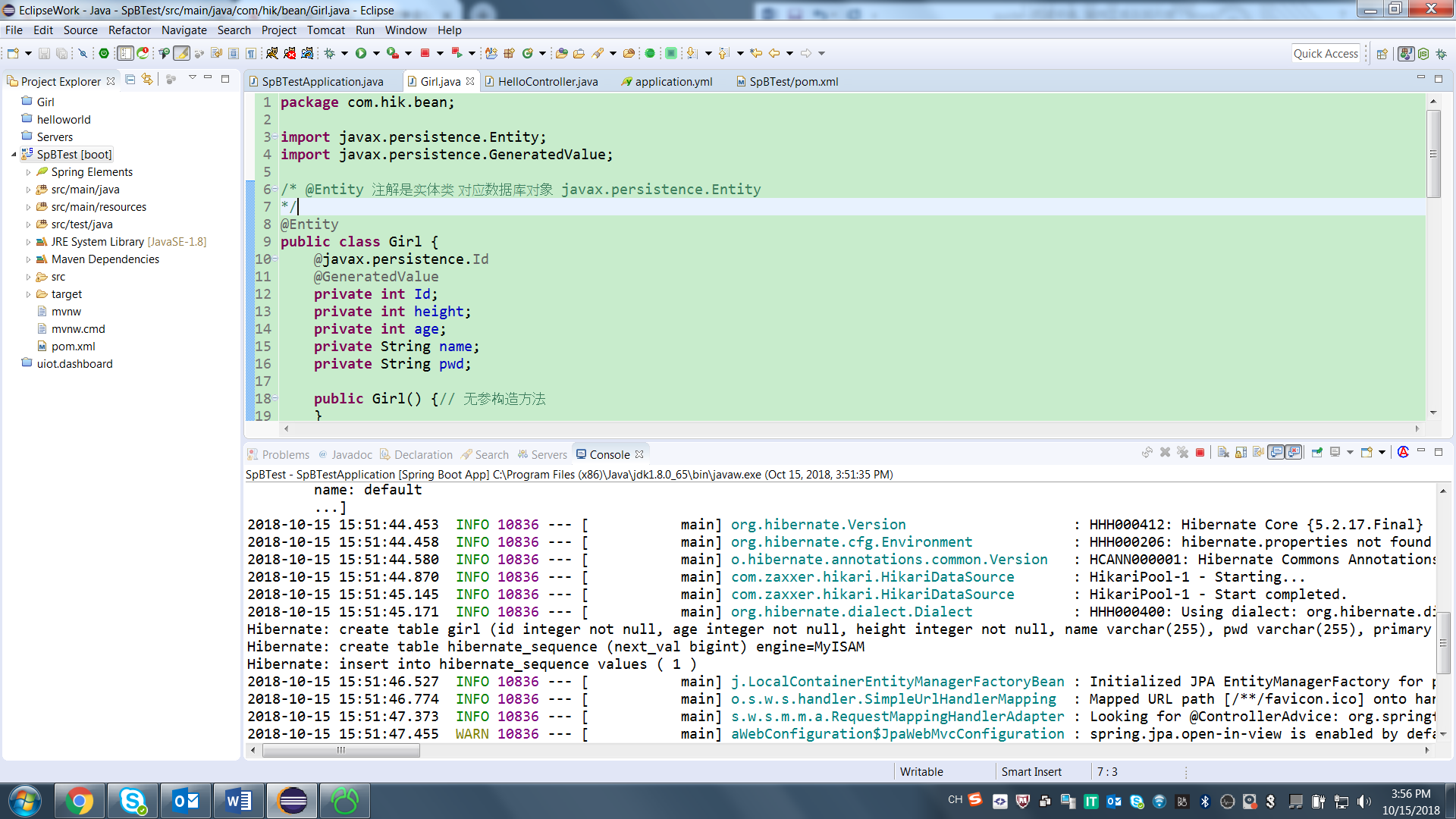
<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

2 添加配置（application.yml）

[SpringBoot连接Mysql数据库警告：Establishing SSL connection without server's identity verification is not recommend](https://www.cnblogs.com/songyao2018/p/8996077.html)：在mysql连接字符串url中加入ssl=true或者false即可 url=jdbc:mysql://127.0.0.1:3306/framework?characterEncoding=utf8&useSSL=true

实体类对数据库创建不成功—引入注解不对 应都是javax.persistence



自动注入失败girlRest required a bean of type 'com.hik.restAPI.GirlRest' that could not be found.—启动类加上注解@ComponentScan(basePackages={"com.hik.restAPI"})

java.sql.SQLException:Unable to load authentication plugin'caching\_sha2\_password'.

法一、my.ini的default\_authentication\_plugin=mysql\_native\_password改caching\_sha2\_password—不好使

3 新建restAPI包及接口com.hik.service及com.hik.service.impl

4 controller调用com.hik.controller

## 事务管理

@Transactional 注解失效---GirlService.java

1 首先要看数据库本身对应的库、表所设置的引擎是什么。MyIsam不支持事务，如果需要，则必须改为InnnoDB。

mysql修改配置文件my.ini我将my-small.ini在[mysqld]最后添加为上default-storage-engine=InnoDB，重启服务，数据库默认的引擎修改为InnoDB

配置文件application.yml properties.hibernate.dialect: org.hibernate.dialect.MySQL5InnoDBDialect

2 @Transactional所注解的方法是否为public，如果应用在protected、private或者 package可见度的方法上，也不会报错，不过事务设置不会起作用。

3 @Transactional所注解的方法所在的类，是否已经被注解@Service或@Component等。

4 需要调用该方法，且需要支持事务特性的调用方是在在 @Transactional所在的类的外面。注意：类内部的其他方法调用这个注解了@Transactional的方法，事务是不会起作用的。

注意：service类标签(一般不建议在接口上)上添加@Transactional，可将整个类纳入spring事务管理，在每个业务方法执行时都会开启一个事务，不过这些事务采用相同的管理方式。

5、注解为事务范围的方法中，事务的回滚仅仅对于unchecked的异常有效。对于checked异常无效。也就是说事务回滚仅仅发生在出现RuntimeException或Error的时候。

6 . 注意仅仅 @Transactional 注解的出现不足于开启事务行为，它仅仅 是一种元数据。必须在配置文件中使用配置元素，才真正开启了事务行为

## Web进阶

表单验证valid

AOP---将通用逻辑从业务逻辑中分离出来

统一处理请求日志(eg 打印日志记录每一个HTTP请求)

添加依赖<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-aop</artifactId>

</dependency>

Aspect包—HttpAspect.java

统一异常处理

Com.hik.enums 统一定义异常说明(code--msg)

Com.hik.exception 自定义异常类型(GirlException)

Com.hik.handle 统一异常处理类(ExceptionHandle)

单元测试

添加依赖<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

</dependency>

Src/tst/java对应类测试

1需指定启动类 @SpringBootTest(classes=SpBTestApplication.class)

2需开启事务管理@Transactional，不然报JPA之"could not initialize proxy - no Session"的错

项目打包会进行单元测试：mvn clean package

打包时跳过单元测试：mvn clean package -Dmaven.test.skip=true

微信小程序(json配置 wxml wcss及js交互)---Area对象操作

官网：https://developers.weixin.qq.com/miniprogram/dev/

微信开发者工具—账号：qq邮箱,密码：RMhuijia？0

列表功能

Springboot DEV热部署—devtools

添加依赖<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional>

</dependency><!-- dev热部署 optional=true依赖不传递-->

配置文件application-dev.yml

项目配置文件注入service、controller

添加依赖<dependency>

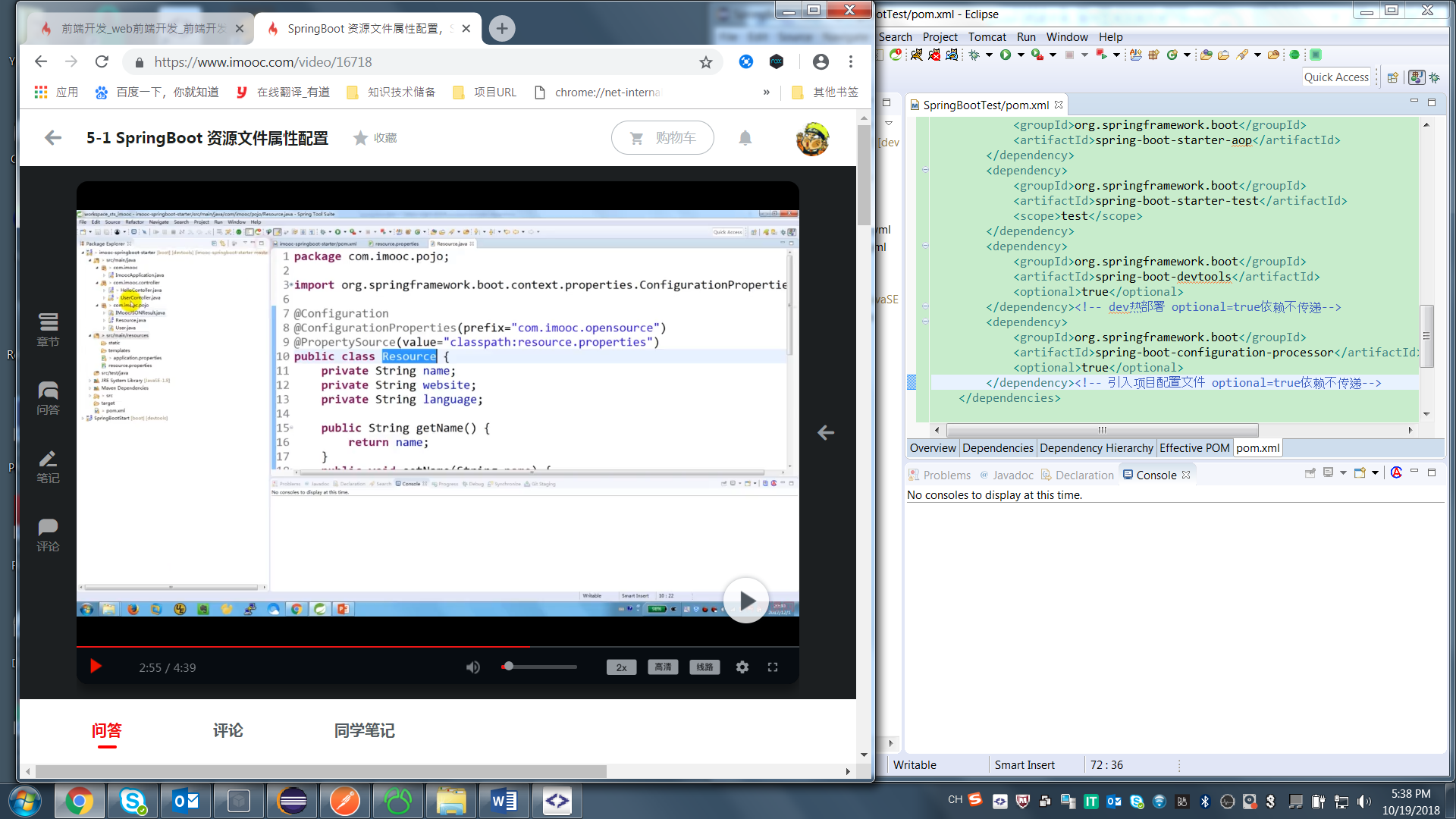
<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-configuration-processor</artifactId>

<optional>true</optional>

</dependency><!-- 引入项目配置文件 optional=true依赖不传递-->

配置文件实体类注解



整合Redis缓存管理

<dependency> <!-- 引入 redis 依赖 -->

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

添加配置application.yml-🡪RedisController

引入Redis封装工具类RedisOperator

解决运行查看Redis结果超时io.lettuce.core.RedisCommandTimeoutException: Command timed out---连接超时时间设置的过于短暂，将spring.redis.timeout=5000即可

下载Redis Desktop Mananger 码云 qq登录 域名：<http://gete/TestRedis> 密码：RMhuijia？1

整合定时任务和异步任务

使用注解@EnableScheduling 开启定时任务(项目启动类)

使用注解@Component 扫描执行定时任务组件

Com.hik.task—TestTask.java

使用注解@EnableAsync 开启异步任务(项目启动类)

使用注解@Component @Async扫描执行定时任务组件

Com.hik.task—AsyncTask.java及DoTask.java测试

使用异步的场景(节省时间)：发送短信或邮件、app消息推送、运维凌晨发布任务

使用拦截器

使用@Configuration配置拦截器

继承WebMvcConfigurerAdapter

重写addInterceptors添加需要拦截地址

com.hik.config—WebMvcConfigurer.java

com.hik.controller.interceptor//拦截定义及拦击controller使用

# 对比

## Struts2和springmvc对比

**1、**Struts2是类级别的拦截， 一个类对应一个request上下文，SpringMVC是方法级别的拦截，一个方法对应一个request上下文，而方法同时又跟一个url对应,所以说从[架构](http://lib.csdn.net/base/16)本身上SpringMVC就容易实现restful url,而struts2的架构实现起来要费劲，因为Struts2中Action的一个方法可以对应一个url，而其类属性却被所有方法共享，这也就无法用注解或其他方式标识其所属方法了。

**2、**由上边原因，SpringMVC的方法之间基本上独立的，独享request response数据，请求数据通过参数获取，处理结果通过ModelMap交回给框架，方法之间不共享变量，而Struts2搞的就比较乱，虽然方法之间也是独立的，但其所有Action变量是共享的，这不会影响程序运行，却给我们编码 读程序时带来麻烦，每次来了请求就创建一个Action，一个Action对象对应一个request上下文。  
**3、**由于Struts2需要针对每个request进行封装，把request，session等servlet生命周期的变量封装成一个一个Map，供给每个Action使用，并保证线程安全，所以在原则上，是比较耗费内存的。

**4、** 拦截器实现机制上，Struts2有以自己的interceptor机制，SpringMVC用的是独立的AOP方式，这样导致Struts2的配置文件量还是比SpringMVC大。

**5、**SpringMVC的入口是servlet，而Struts2是filter（这里要指出，filter和servlet是不同的。以前认为filter是servlet的一种特殊），这就导致了二者的机制不同，这里就牵涉到servlet和filter的区别了。

**6、**SpringMVC集成了Ajax，使用非常方便，只需一个注解@ResponseBody就可以实现，然后直接返回响应文本即可，而Struts2拦截器集成了Ajax，在Action中处理时一般必须安装插件或者自己写代码集成进去，使用起来也相对不方便。

**7、**SpringMVC验证支持JSR303，处理起来相对更加灵活方便，而Struts2验证比较繁琐，感觉太烦乱。

**8、**Spring MVC和Spring是无缝的。从这个项目的管理和安全上也比Struts2高（当然Struts2也可以通过不同的目录结构和相关配置做到SpringMVC一样的效果，但是需要xml配置的地方不少）。

**9、** 设计思想上，Struts2更加符合OOP的编程思想， SpringMVC就比较谨慎，在servlet上扩展。

**10、**SpringMVC开发效率和性能高于Struts2。  
**11、**SpringMVC可以认为已经100%零配置。

Struts: 在struts web应用程序中，当web应用程序启动的时候，就会初始化ActionServlet在初始化ActionServlet的时候会加载struts-config.xml配置文件，在加载成功后会把这些URL和控制器映射关系存放在ActionMapping对象或者其他对象中。当ActionServlet接收到用户请求的时候，就会按照下面的流程对用户请求进行处理。

（1）ActionServlet接收到用户的请求后，会根据请求URL寻找匹配的ActionMapping对象，如果匹配失败，说明用户请求的URL路径信息有误，所以返回请求路径无效的信息，当找到匹配的ActionMapping的时候，进入到下一步。

（2）当ActionServlet找到匹配的ActionMapping对象的时候，会根据ActionMapping中的映射信息判断对应的ActionForm对象是否存在，如果不存在对应的ActionForm对象就创建一个新的ActionForm对应，并把用户提交的表单信息保存到这个ActionForm对象中。

（3）在struts-config.xml中这个配置文件，可以配置表单是否需要验证，如果需要验证，就调用ActionForm中的validate（）方法对用户输入的表单进行验证。

（4）如果ActionForm的validate（）方法返回了ActionErrors对象，则表明验证失败，ActionServlet把这个页面返回到用户输入的界面，提示用户重新输入。如果方法的返回值为null，就表明验证已经通过，可以进入下一步处理。

（5）ActionServlet可以根据ActionMapping对象查找用户请求转发给哪个控制器Action，如果对应的Action对象不存在，就创建这个对象，并调用这个Action的excute（）方法。

（6）业务逻辑控制器Action的execute（）方法就会返回一个ActionForward对象，ActionServlet把控制器处理的结果转发到ActionForward对象指定的JSP页面。

（7）ActionForward对象指定的JSP页面根据返回的处理结果，用合适形式把服务器处理的结果展示给用户，到这里为止，一个客户请求的整个过程完毕。

struts2是webwork升级版，跟struts1没有关系。struts1和servlet耦合度高，各层之间耦合度高，单元测试困难，表示层技术单一，而struts2没有和servlet耦合，更容易测试，支持更多表示层技术。

struts2工作流程

1）  客户端初始化一个指向Servlet容器（例如Tomcat）的请求。2）  这个请求经过一系列的过滤器（Filter）。3）  接着FilterDispatcher被调用，FilterDispatcher询问ActionMapper来决定这个请是否需要调用某个Action。4）  如果ActionMapper决定需要调用某个Action，FilterDispatcher把请求的处理交给ActionProxy。5）  ActionProxy通过Configuration Manager询问框架的配置文件，找到需要调用的Action类。6）  ActionProxy创建一个ActionInvocation的实例。7）  ActionInvocation实例使用命名模式来调用，在调用Action的过程前后，涉及到相关拦截器（Intercepter）的调用。8）  一旦Action执行完毕，ActionInvocation负责根据struts.xml中的配置找到对应的返回结果。Struts2的核心就是拦截器。Struts.xml中所有的package都要extends="struts-default"。同理与所有的Java类都要extends自Object一样。struts-default.xml里面就是要做以上事情。

## Spring springmvc springboot对比

Spring 框架就像一个家族，有众多衍生产品例如 boot、security、jpa等等。但他们的基础都是Spring 的 ioc和 aop ioc 提供了依赖注入的容器 aop ，解决了面向横切面的编程，然后在此两者的基础上实现了其他延伸产品的高级功能。Spring MVC是基于 Servlet 的一个 MVC 框架 主要解决 WEB 开发的问题，因为 Spring 的配置非常复杂，各种XML、 JavaConfig、hin处理起来比较繁琐。于是为了简化开发者的使用，从而创造性地推出了Spring boot，约定优于配置，简化了spring的配置流程。

Spring MVC的功能:提供了一种轻度耦合的方式来开发web应用。Spring MVC是Spring的一个模块，是一个web框架。通过Dispatcher Servlet, ModelAndView 和 View Resolver，开发web应用变得很容易。解决的问题领域是网站应用程序或者服务开发——URL路由、Session、模板引擎、静态Web资源等等。

Spring Boot的功能：实现了自动配置，降低了项目搭建的复杂度。Spring Boot本身并不提供Spring框架的核心特性以及扩展功能，只是用于快速、敏捷地开发新一代基于Spring框架的应用程序，是和Spring框架紧密结合用于提升Spring开发者体验的工具。同时它集成了大量常用的第三方库配置(例如Jackson, JDBC, Mongo, Redis, Mail等等)，Spring Boot应用中这些第三方库几乎可以零配置的开箱即用(out-of-the-box)，大部分的Spring Boot应用都只需要非常少量的配置代码，开发者能够更加专注于业务逻辑。

Spring Boot只是承载者，辅助你简化项目搭建过程的。如果承载的是WEB项目，使用Spring MVC作为MVC框架，那么工作流程和你上面描述的是完全一样的，因为这部分工作是Spring MVC做的而不是Spring Boot。

对使用者来说，换用Spring Boot以后，项目初始化方法变了，配置文件变了，另外就是不需要单独安装Tomcat这类容器服务器了，maven打出jar包直接跑起来就是个网站，但你最核心的业务逻辑实现与业务流程实现没有任何变化。

所以Spring 是一个“引擎”；Spring MVC 是基于Spring的一个 MVC 框架 ；

Spring Boot 是基于Spring4的条件注册的一套快速开发整合包。

# Hibernate与 MyBatis的比较

一 简介

Hibernate 是当前最流行的O/R mapping框架，它出身于sf.net，现在已经成为Jboss的一部分。 Mybatis 是另外一种优秀的O/R mapping框架。目前属于apache的一个子项目。

Hibernate对数据库结构提供了较为完整的封装，Hibernate的O/R Mapping实现了POJO 和数据库表之间的映射，以及SQL 的自动生成和执行。程序员往往只需定义好了POJO 到数据库表的映射关系，即可通过Hibernate 提供的方法完成持久层操作。程序员甚至不需要对SQL 的熟练掌握， Hibernate/OJB 会根据制定的存储逻辑，自动生成对应的SQL 并调用JDBC 接口加以执行。

iBATIS 的着力点，则在于POJO 与SQL之间的映射关系。然后通过映射配置文件，将SQL所需的参数，以及返回的结果字段映射到指定POJO。 相对Hibernate“O/R”而言，iBATIS 是一种“Sql Mapping”的ORM实现。

二 开发对比

Hibernate和MyBatis都有相应的代码生成工具。可以生成简单基本的DAO层方法。

针对高级查询，Mybatis需要手动编写SQL语句，以及ResultMap。而Hibernate有良好的映射机制，开发者无需关心SQL的生成与结果映射，可以更专注于业务流程。

三 系统调优对比

Hibernate的调优方案

制定合理的缓存策略；尽量使用延迟加载特性；采用合理的Session管理机制；

使用批量抓取，设定合理的批处理参数（batch\_size）;进行合理的O/R映射设计

Mybatis调优方案

MyBatis在Session方面和Hibernate的Session生命周期是一致的，同样需要合理的Session管理机制。MyBatis同样具有二级缓存机制。 MyBatis可以进行详细的SQL优化设计。

SQL优化方面

Hibernate的查询会将表中的所有字段查询出来，这一点会有性能消耗。Hibernate也可以自己写SQL来指定需要查询的字段，但这样就破坏了Hibernate开发的简洁性。而Mybatis的SQL是手动编写的，所以可以按需求指定查询的字段。Hibernate HQL语句的调优需要将SQL打印出来，而Hibernate的SQL被很多人嫌弃因为太丑了。MyBatis的SQL是自己手动写的所以调整方便。但Hibernate具有自己的日志统计。Mybatis本身不带日志统计，使用Log4j进行日志记录。

扩展性方面

Hibernate与具体数据库的关联只需在XML文件中配置即可，所有的HQL语句与具体使用的数据库无关，移植性很好。MyBatis项目中所有的SQL语句都是依赖所用的数据库的，所以不同数据库类型的支持不好。

两者相同点

Hibernate与MyBatis都可以是通过SessionFactoryBuider由XML配置文件生成SessionFactory，然后由SessionFactory 生成Session，最后由Session来开启执行事务和SQL语句。其中SessionFactoryBuider，SessionFactory，Session的生命周期都是差不多的。

Hibernate和MyBatis都支持JDBC和JTA事务处理。

Mybatis优势

MyBatis可以进行更为细致的SQL优化，可以减少查询字段。

MyBatis容易掌握，而Hibernate门槛较高。

Hibernate优势

Hibernate的DAO层开发比MyBatis简单，Mybatis需要维护SQL和结果映射。

Hibernate对对象的维护和缓存要比MyBatis好，对增删改查的对象的维护要方便。

Hibernate数据库移植性很好，MyBatis的数据库移植性不好，不同的数据库需要写不同SQL。

Hibernate有更好的二级缓存机制，可以使用第三方缓存。MyBatis本身提供的缓存机制不佳。

他人总结

Hibernate功能强大，数据库无关性好，O/R映射能力强，如果你对Hibernate相当精通，而且对Hibernate进行了适当的封装，那么你的项目整个持久层代码会相当简单，需要写的代码很少，开发速度很快，非常爽。

Hibernate的缺点就是学习门槛不低，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡取得平衡，以及怎样用好Hibernate方面需要你的经验和能力都很强才行。

iBATIS入门简单，即学即用，提供了数据库查询的自动对象绑定功能，而且延续了很好的SQL使用经验，对于没有那么高的对象模型要求的项目来说，相当完美。

iBATIS的缺点就是框架还是比较简陋，功能尚有缺失，虽然简化了数据绑定代码，但是整个底层数据库查询实际还是要自己写的，工作量也比较大，而且不太容易适应快速数据库修改。

# JVMJava虚拟机

JVM的设计目标是提供一个基于抽象规格描述的计算机模型，为解释程序开发人员提供很好的灵活性，同时也确保Java代码可在符合该规范的任何系统上运行。JVM是java的核心和基础，在java编译器和os平台之间的虚拟处理器。它是一种基于下层的操作系统和硬件平台并利用软件方法来实现的抽象的计算机，可以在上面执行java的字节码程序。

java编译器只需面向JVM，生成JVM能理解的代码或字节码文件。Java源文件经编译器，编译成字节码程序，通过JVM将每一条指令翻译成不同平台机器码，通过特定平台运行。

**JVM执行程序的过程 :**

I.加载.class文件

II.管理并分配内存

III.执行垃圾收集

JRE(java运行时环境)包含JVM的java程序的运行环境 [[1]](https://baike.so.com/doc/1063579-1125177.html" \l "refff_1063579-1125177-1)

JVM是Java程序运行的容器,但是他同时也是操作系统的一个进程,因此他也有他自己的运行的生命周期,也有自己的代码和数据空间。

JVM在整个jdk中处于最底层,负责与操作系统的交互,用来屏蔽操作系统环境,提供一个完整的Java运行环境,因此也叫虚拟计算机.操作系统装入JVM是通过jdk中Java.exe来完成,通过下面4步来完成JVM环境。

1.创建JVM装载环境和配置

2.装载JVM.dll

3.初始化JVM.dll并挂接到JNIENV(JNI调用接口)实例

4.调用JNIEnv实例装载并处理[class类](https://baike.so.com/doc/4126483-4325917.html)。[[2]](https://baike.so.com/doc/1063579-1125177.html" \l "refff_1063579-1125177-2)

JRE和JVM的区别

JRE(JavaRuntimeEnvironment，Java运行环境)，也就是Java平台。所有的Java程序都要在JRE下才能运行。JDK的工具也是Java程序，也需要JRE才能运行。为了保持JDK的独立性和完整性，在JDK的安装过程中，JRE也是安装的一部分。所以，在JDK的安装目录下有一个名为jre的目录，用于存放JRE文件。

JVM(JavaVirtualMachine，Java虚拟机)是JRE的一部分。它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。JVM有自己完善的硬件架构，如处理器、堆栈、寄存器等，还具有相应的指令系统。Java语言最重要的特点就是跨平台运行。使用JVM就是为了支持与操作系统无关，实现跨平台。[[](https://baike.so.com/doc/1063579-1125177.html" \l "refff_1063579-1125177-4)