# 第一章 基本常识

# 1.面试与工作准备

## idea下载安装破解

1. 面试的时候带纸和笔，方便作图展示
2. 下载安装：

下载地址（网盘已存最终版）：

[Jetbrains公司](https://www.jetbrains.com/idea/)

路径要选择英文路径，不要含带中文路径

安装时关键页面的选择：

​ Create assodations：全不选。

（3）破解：

激活码 网址：<http://idea.lanyus.com/>

​ 激活码目前为博主分享的自用激活码，到期前会更换新的激活码。

​ 使用前请将“0.0.0.0 account.jetbrains.com”添加到hosts文件中（这一步要提前操作）。

​ hosts文件：C:\Windows\System32\drivers\etc

​ 然后进入激活码网址获取注册码

​ 打开软件，在license Activation对话框选择第二项Activation code

​ 把注册码复制到框里

​ 破解完成

## 1.2 idea初始化和JDK配置

1. 初始化：

破解完后首先选择视图，试图区别为背景色分别为黑和白。然后全点击继续类按钮直到最后一个页面点击创建新项目。

​ 打开后，先关联JDK：New---c:Program Files\java\jdk1.8.0\_131

​ 然后点next---再点next--设置路径--finish--取消左下角Show Tips on Startup选项--Next Tip--点击左下角Pla。

# 2.小知识总结

## 2.1git与github与TortoiseGit

1. 上传文件时，target .idea im要忽略不上传
2. TortoiseGit右键无菜单问题：重启一下explorer.exe即可-：任务管理器--选中explorer---结束任务---windows键+R--输入explorer.exe---再右键就看到了
3. 如果git安装失败，可以尝试以管理员身份启动试试，想使用TortoiseGit，就必须先安装git

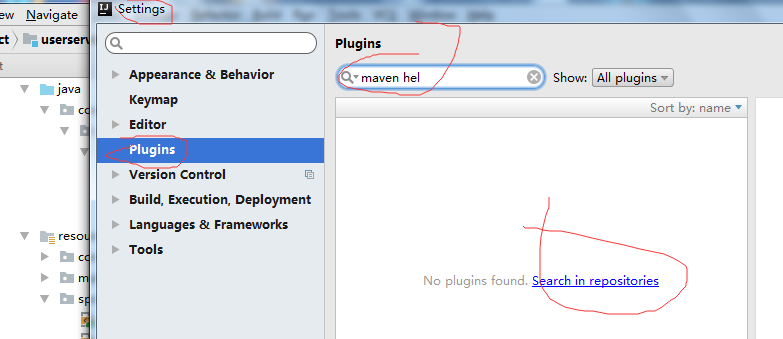
## 2.2idea使用技巧与maven

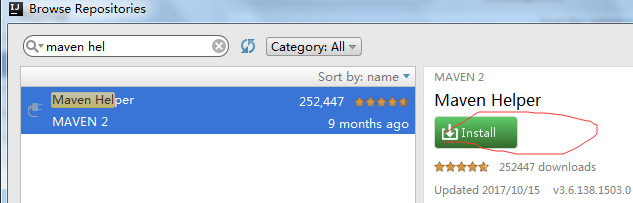
### 2.2.1maven

#### 2.2.1.1插件

1. maven插件安装，作用于全局的install

搜索maven helper





#### 2.2.1.2 jar

##### 2.2.1.2.1 pom技巧

（1）<classifier>标签的使用

另类pom，有的时候jar pom就是依赖不进来，可能是因为版本的问题，此时需要用到classfile标签

|  |
| --- |
| <dependency>  <groupId>net.sf.json-lib</groupId>  <artifactId>json-lib</artifactId>  <version>2.4</version>  <classifier>jdk15</classifier>  </dependency> |

（2）另类引包方式

只有一个jar文件时，可以使用下面导入方式，将jar文件放到webapp\WEB-INF文件夹下，然后引入的时候使用下面pom：

|  |
| --- |
| <dependency>  <groupId>ValidateCode</groupId>  <artifactId>ValidateCode</artifactId>  <scope>system</scope>  <systemPath>${project.basedir}/src/main/webapp/WEB-INF/lib/ValidateCode.jar</systemPath>  <version>1.0</version>  </dependency> |



（3）约定版本

|  |
| --- |
| <!--规定相关文件版本-->  <properties>  <spring.version>4.3.13.RELEASE</spring.version>  <hibernate.version>5.2.10.Final</hibernate.version>  <!--日志文件管理包版本-->  <slf4j.version>1.7.21</slf4j.version>  <log4j.version>1.2.17</log4j.version>  </properties> |

##### 2.2.1.2.2jar包功能介绍

（1）锁定依赖标签，用于锁定版本号，比如springpom锁定之后，再往后引入的所有的springpom版本都将没法设置，全都使用锁定标签之内的版本号，但需要注意的是，pom中锁定标签仅仅是锁定pom,但并没把依赖引进来。所以要是想使用某依赖，还需要在<dependencyManagement>外再引pom，但是不用写版本号了。

|  |
| --- |
| <!--锁定依赖,如果你使用了这个依赖,那么将被锁定-->  <dependencyManagement>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>${spring.version}</version>  </dependency>  </dependencyManagement>  <!--使用依赖,注意,锁定的依赖仅仅是锁定,并没依赖进来-->  <dependencies>  <!--这是spring相关的依赖包-->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  </dependency>  </dependencies> |

（2）dubbl和zookeeper

|  |
| --- |
| <!--用于和zookeeper通信的分布式依赖-->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>dubbo</artifactId>  <version>2.6.0</version>  </dependency>  <!--zookeeper客户端,持有协议-->  <dependency>  <groupId>com.101tec</groupId>  <artifactId>zkclient</artifactId>  <version>0.10</version>  </dependency> |

### 2.2.2idea使用技巧

（1）解决引入pom时不提示的问题

当你自己把自己的程序打包到本地仓库后，发现引用的时候却不提示，此时可以刷新一下本地仓库。

右键项目

-----setting-----Build,Executiom,Deployment--BuildTools---Maven---Repositories---选中给本地仓库路径，点击更新

## 2.3项目经验

### 2.3.1数据库，CRUD,SQL

#### 2.3.1.1不同表的关联信息问题

##### 2.3.1.1.1冗余数据的使用

冗余数据的使用：当表2中的一列和表1有关联时，不巧表一表二要放到两个服务器上，那总不能请求的时候访问两个地址吧，此时解决的方法就是在表一添加冗余数据，把表2相关的列直接加到表一后面的新列上，这样查询的时候直接查询表一即可，但是要注意的是，当表二该列变化时，别忘记更新表一相关列

##### 2.3.1.1.2数据预热

数据预热即通过监听器，在容器初始化时，就把数据放到缓存中，其监听器一般配置在service层的实现中的web.xml中，其执行流程以request的监听器为例，就是tomcat启动时会执行request相关方法创建request，此时如果看到web.xml中有监听器，就会逐一的根据监听器的配置，把需要初始化的方法初始化执行一下。

当表一表二有关联查询时，恰巧表二数据比较少，但是使用却很频繁，此时可以考虑数据预热，提前把表二的数据放到缓存中（如redis），这样用的时候直接在缓存中取就可以了，就用不到使用表二数据时去关联查询了。

#### 2.3.1.2

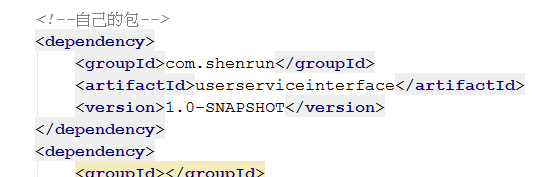
### 2.3.2命名规范

1. 通用打包程序

|  |
| --- |
| <context:component-scan base-package="com.common.validcode.controller"/> |

如上图，如果该项目需要放到服务器上被访问，那么testmaincontroller类的包名最好设置为controller。

1. 如图引入jar包的时候，对应的<groupId>，<artifactId>分别对应于建项目的时候两个名字，故以后全要小写，大写了多难看。多个单词汇集一起的时候用中横线分割。



1. 前端请求的属性名和我们定义的实体类属性名要一致

比如前端将参数封装到user对象中传递给我们，但是他的属性名写的是username，而我们定义的实体类user中的属性名是userName,那么我们输出user.userName的时候可能输出的会是空值，所以最好都用驼峰命名法，前端对应属性的name也定义成userName。

（4）如下图所示，controller请求接口最好全小写



### 2.3.3定接口，开项目注意事项

1. 比如注册时，哪些数据用户是复选框类型的，这样就不用判断空值或者空字符了。
2. 有些东西不能写死，例如图片的访问地址要把域名抽象出来，要不然换个地址就需要改掉所有地址。
3. 所有的spring配置文件都放在spring文件夹下，，开头都是以application开头，便于导入依赖时好扫描
4. 如下图所示，controller请求接口最好全小写



### 2.3.4项目优化

#### 2.3.4.1sql优化

1. sql语句关键字要大写，这样可以省去执行sql语句时，数据库转化大写字母的时间
2. Select \* 执行时，数据库会先去扫描识别都是有哪些列需要查询出来，所以不要直接select \*，用具体的列名代替\*

#### 2.3.4.2数据预热

（1）数据预热的实现：把数据在系统初始化的时候就放到缓存中，tomcat容器启动web.xml时执行监听器，spring执行的时候加载初始化方法调用mapper

（2）能做数据预热操作的数据特点:不会经常改变（比如12306的出发站城市，这个数据基本是不会变的），数据查询比较频繁

（3）数据预热即通过监听器，在容器初始化时，就把数据放到缓存中，其监听器一般配置在service层的实现中的web.xml中，其执行流程以request的监听器为例，就是tomcat启动时会执行request相关方法创建request，此时如果看到web.xml中有监听器，就会逐一的根据监听器的配置，把需要初始化的方法初始化执行一下。

当表一表二有关联查询时，恰巧表二数据比较少，但是使用却很频繁，此时可以考虑数据预热，提前把表二的数据放到缓存中（如redis），这样用的时候直接在缓存中取就可以了，就用不到使用表二数据时去关联查询了。

## 2.4Java监听器

监听器用于监听web应用中某些对象、信息的创建、销毁、增加，修改，删除等动作的发生，然后作出相应的响应处理。当范围对象的状态发生变化的时候，服务器自动调用监听器对象中的方法。常用于统计在线人数和在线用户，系统加载时进行信息初始化，统计网站的访问量等等。

一共8个

按监听的对象划分，可以分为

ServletContext对象监听器（监听创建销毁的ServletContextListener ，监听属性发生变化ServletContextAttributeListener）：程序启动的时候就触发

HttpSession对象监听器

ServletRequest对象监听器

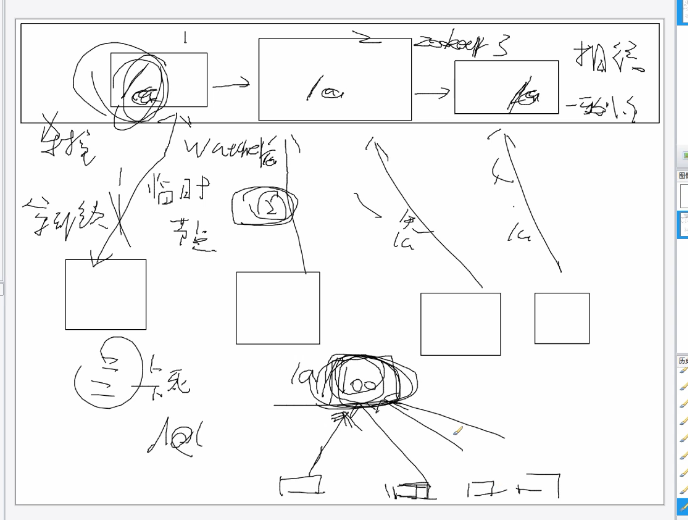
按监听的事件划分

对象自身的创建和销毁的监听器

对象中属性的创建和消除的监听器

session中的某个对象的状态变化的监听器

## 2.5zookeeper



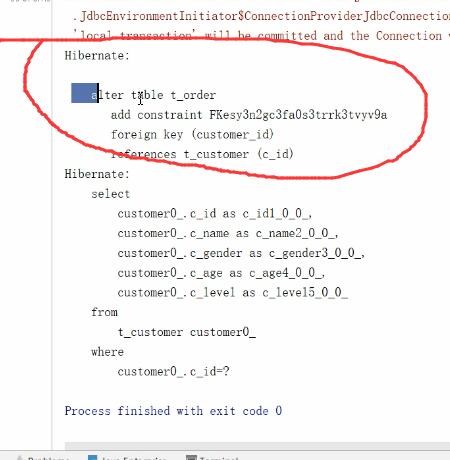
1. zookeeper相当于一个集群，不同zookeeper服务器之间保持一致性，当机器1向zookeeper1发送消息创建节点a时，其他机器向任意zookeeper发送消息创建节点a都将失败。
2. 机器与zookeeper之间的联系分为临时节点和永久节点两种，当建立的是永久节点时，即使重启服务器，该节点也不会消失
3. 当很多服务器请求a节点时，如果a节点坏掉了，不需要消息通知，所有的请求会知道这个节点已经坏掉，都会去自动请求新节点。

# 第二章 报错集锦

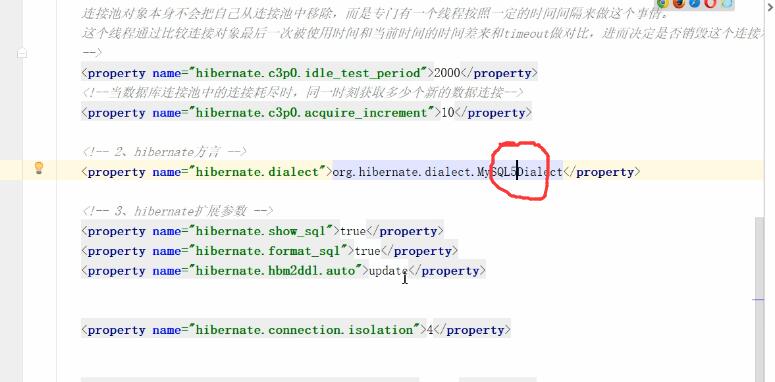
# 1.IDEA

## 1.1idea-hibernate

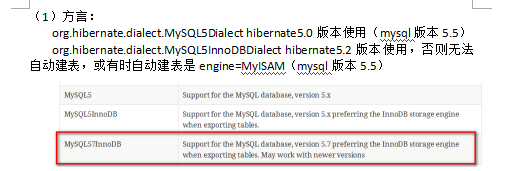
（1）重复打印设置外键语句问题：



如上图所示，每次执行程序都会打印创建外键语句，出现这种情况，首先要想到是表中的外键创建没有成功，所以每次执行语句，hibernate都会检测到程序与数据库表格有差异，需要建立外键，从而执行创建外键的Sql语句。

所以首先打开Sqlyog查看数据库对应表的设计表是否有外键生成，如果没有生成，则说明外键一直都是创建失败，造成这种情况的原因之一在于hibernate核心配置文件中方言版本设置有误，如下图所示设置的版本是5。 

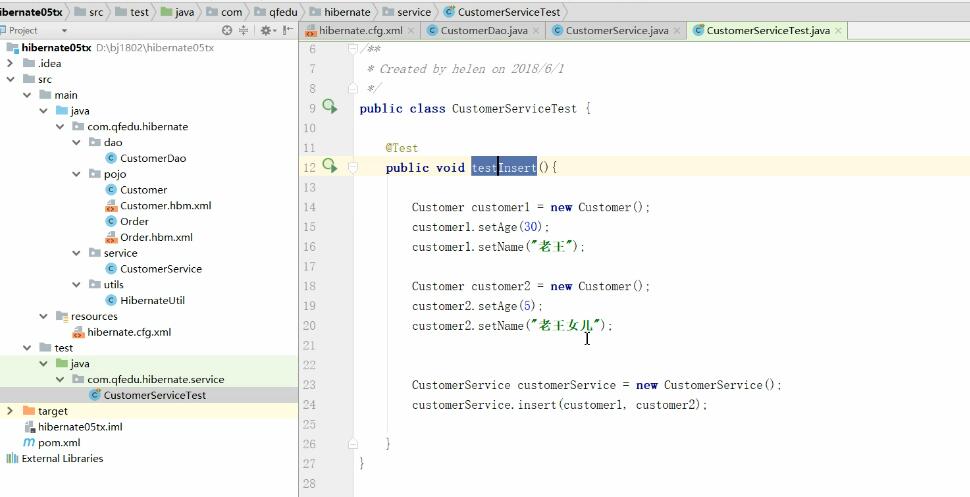
根据下图中方言的版本信息，可以看出，方言版本选择的是5（而一般都是使用57的或者5InnoDB），存储引擎会使用MyISAM，而这种存储方式不支持事务，不支持建立外键，所以才会每次执行数据都打印建立外键的Sql语句。



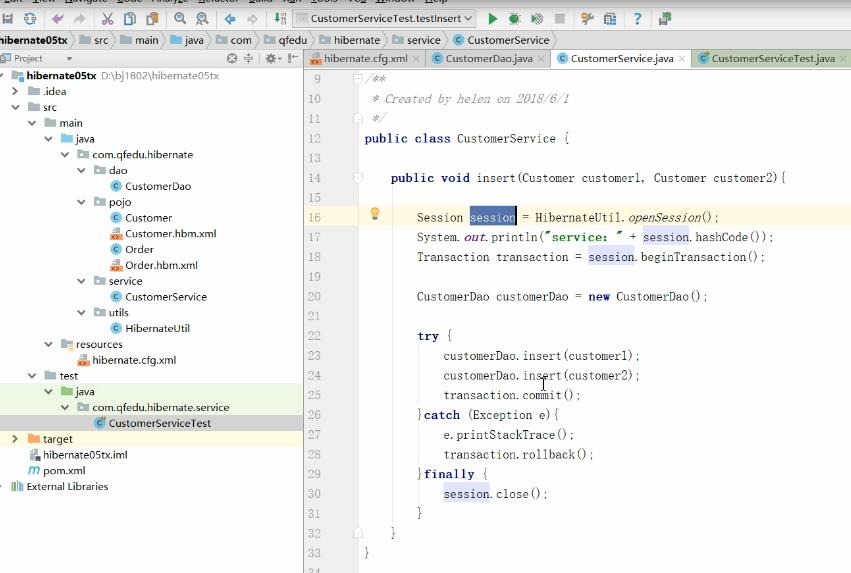
只需要把方言版本修改成57或者5InnoDB即可。

1. 执行下面的程序，发现虽然事务层和实体层session不一样，但仍然能成功的把数据插入数据库，这是怎么回事？

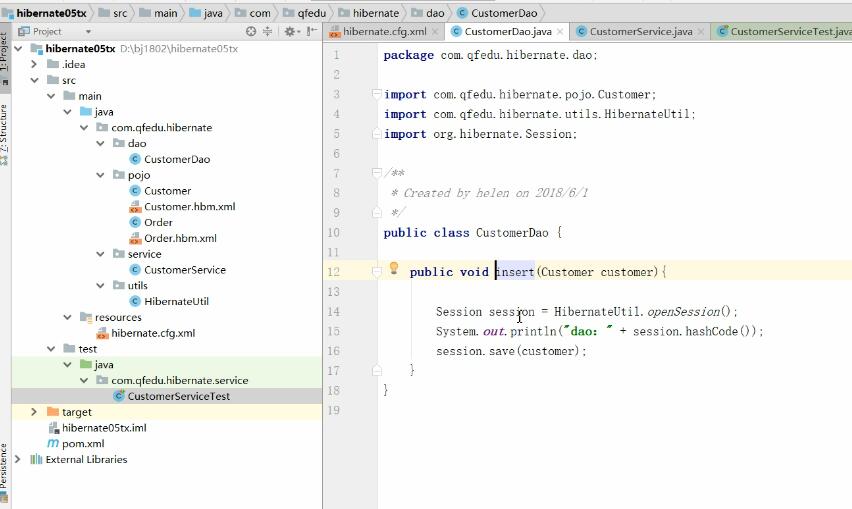
Test函数：



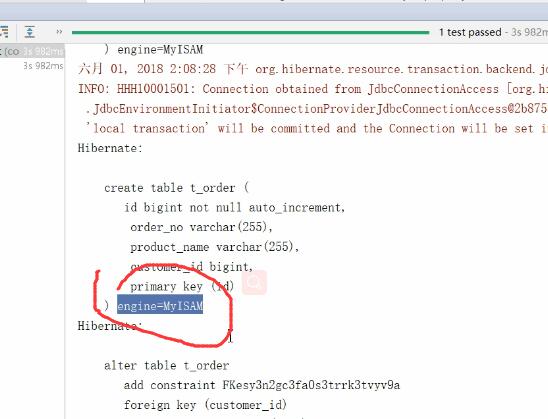
业务层函数：



Dao层函数：

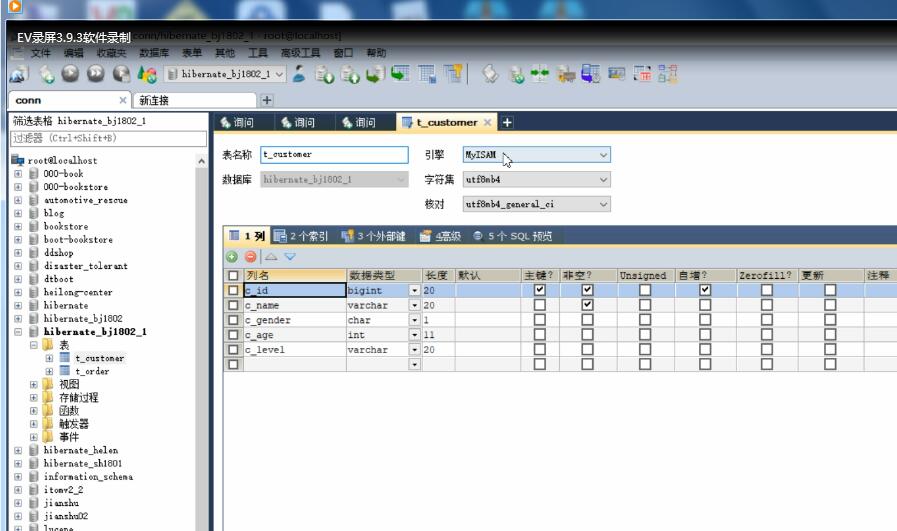


将hibernate的生成表方式改为create，打印会发现以下控制台信息：

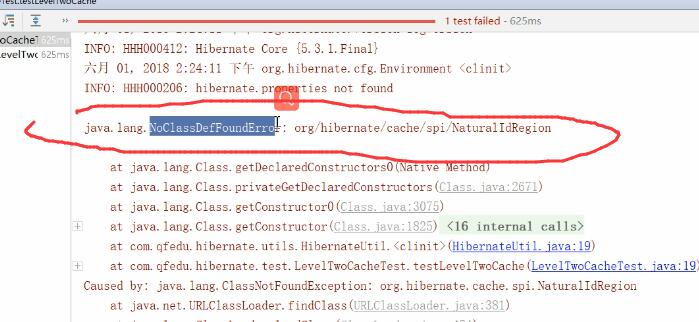


从执行结果可以看出，和上面一个题的问题是一样的，方言设置有问题，储存引擎是MyISAM，而这种存储方式不支持事务，所以事务设置无效，当在Dao层运行session.save时，数据就已经储存在了数据库中。

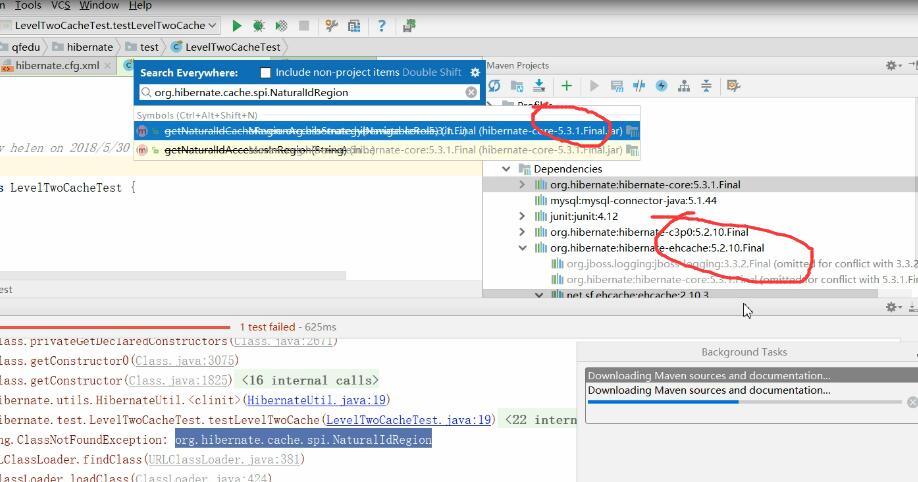
备注：右键数据库对应表格，点击改变表出现以下图所示表，可以看储存引擎。



（3）出现如下图所示异常怎么回事？



异常中出现的找不到的类不是我们自己建立的类，这种情况百分之八十是因为导入的jar包版本出现了冲突，如下图所示，双击shift键会出现搜索框，在框里输入包名搜索出jar包信息会发现版本和缓存jar包等包的信息中间那个数查了一个等级，所以两者不匹配，方法就是修改导入jar包的版本号（当版本号是RELEASE的时候表示要使用最新版）。



## 1.2idea-pom-插件

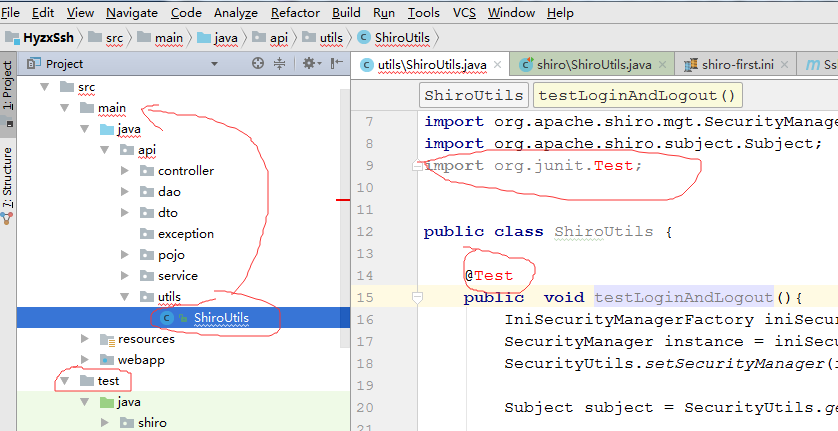
### 1.2.1juint@Test

1. 引入了juint的pom，但是@Test就是不能用的问题

如下图所示，juint已经被引入了，但是仍然不能使用@Test，原因在于你测试的位置不再test文件夹下，但是查看一下pom中junit的坐标注明了一句：

<scope>test</scope>（规定只能在test文件下测试使用）

所以此时只需要把这句话删除就可以了



（2）Junit测试出现异常：Exception in thread "main" java.lang.NoSuchMethodError: org.junit.platform.commons.util.

此时查看一下异常中是否有类似于

At com.intellij.junit5.JUnit5IdeaTestRunner.createListeners(JUnit5IdeaTestRunner)

的提示，如果有，就说明系统正在尝试用junit5或者其他型号进行测试，而你引入了4.12junit，想用4.12测试，所以可以看看你的pom中是否同时引入了以下两个jar：

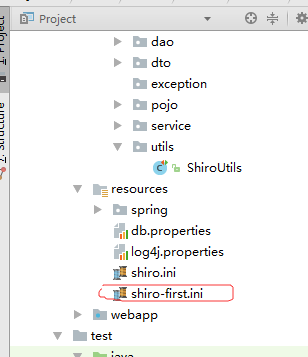
|  |
| --- |
| <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  </dependency>  <dependency>  <groupId>org.junit.jupiter</groupId>  <artifactId>junit-jupiter-api</artifactId>  <version>RELEASE</version>  </dependency>  </dependencies> |

如果有，删除junit-jupiter-api即可。

### 1.2.2拷贝资源文件resources插件

1. Resources中添加了文件，但是编译的时候找不到问题：

如下图所示，文件明明已经加在resources文件下了，结果一运行，出现了如下异常：org.apache.shiro.config.ConfigurationException: java.io.IOException: Resource [classpath:shiro-first.ini] could not be found.

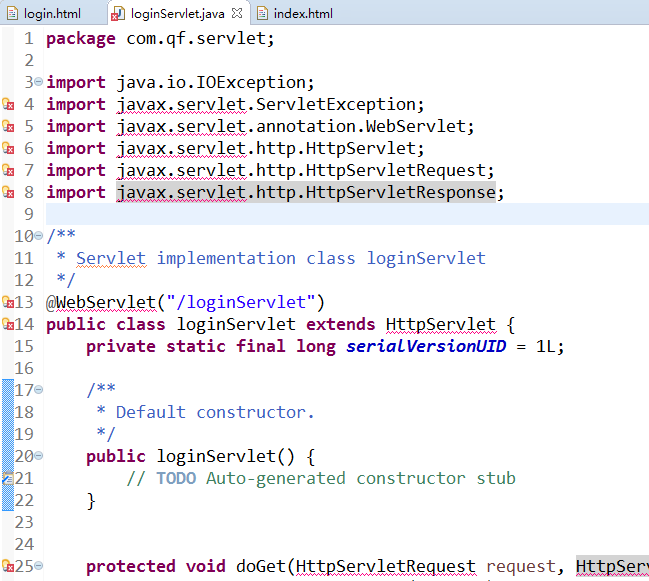


这个问题的罪魁祸首是由于pom中引入了下面的拷贝资源插件，且该插件配置不对，解决办法：要么删除这个插件，要么在插件中后半部分的 <includes>中更改设置，增加代码：<include>\*\*/\*.ini</include>

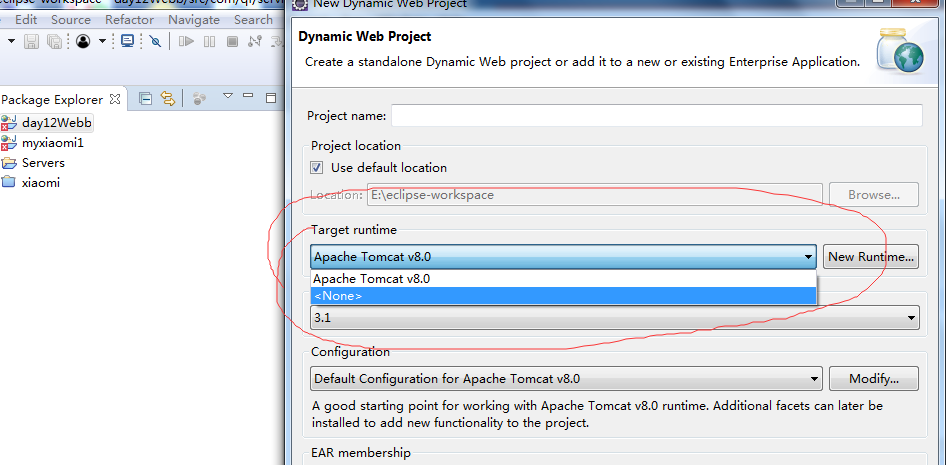
|  |
| --- |
| <!-- 拷贝资源文件 -->  <!-- 添加此节点将mybatis的XxxMapper.xml文件发布到target路径下 -->  <resources>  <resource>  <directory>src/main/java</directory>  <includes>  <!-- <include>\*\*/\*.properties</include> -->  <include>\*\*/\*.xml</include>  </includes>  <filtering>false</filtering>  </resource>  <!-- 上面的配置导致src/main/resources的所有文件都不能被扫描，因此还要配多一个 -->  <resource>  <directory>src/main/resources</directory>  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.xml</include>  </includes>  <filtering>false</filtering>  </resource>  </resources> |

# Eclipse

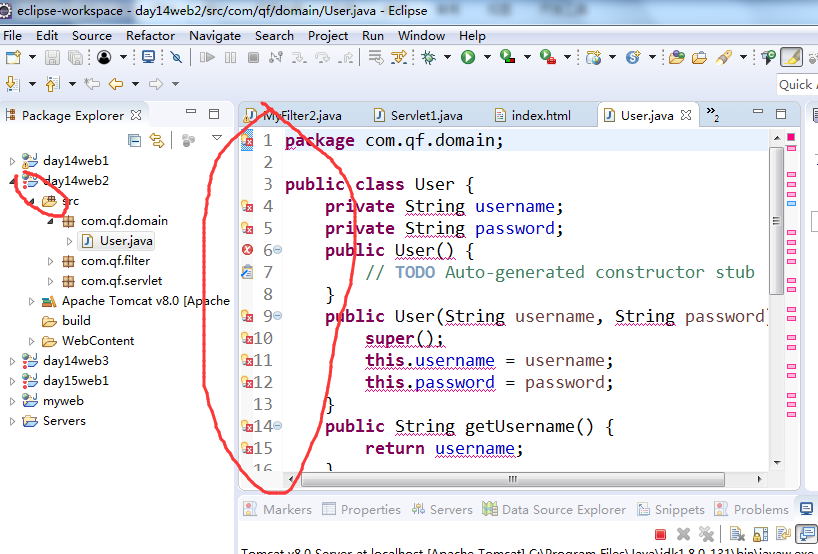
1.新建Web项目，新建Servlet文件，自动生成的程序竟然有错，这是为何？



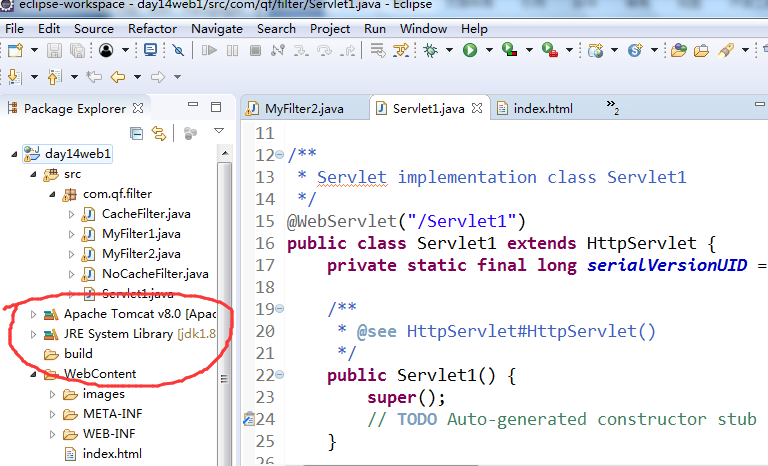
答：这是由于在创建Web项目时，没有导入包，即如下图所示Target runtime栏选的是None。 解决办法：选中项目右键，选中properties，选中Target runtimes，在后面的服务器上打勾



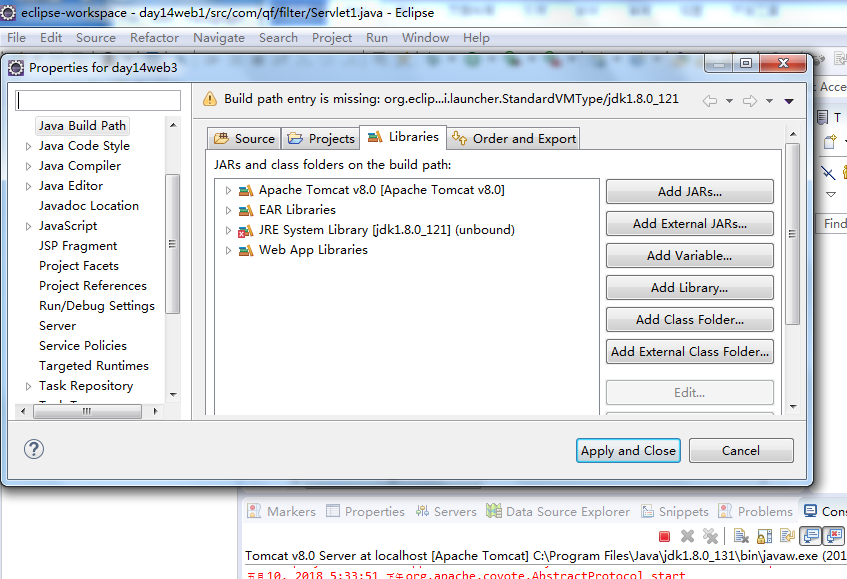
2.导入Web项目后，竟然满篇都是错误如下图，怎么办？



答：（1）先看项目下是否少了JRE System Library，正常项目下都该有这个jar包，如图所示：

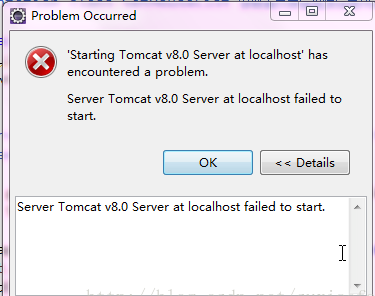


若没有，则选中项目右键点击Build Path->configer build path,出现如下图所示窗口：



选中带红叉号的JRE System Library，然后点击右边出现的remove--->然后点击右边的add Library-->选中JRE System Library—>点击next-->点击finish-->点击应用。

3.遇到如下问题如何解决？



答：应该文件有问题，[比如过滤器中@WebFilter("\*.jsp")](mailto:比如过滤器中@WebFilter(%22*.jsp%22))[写成了@WebFilter("/\*.jsp")](mailto:写成了@WebFilter(%22/*.jsp%22))就会出现如上问题，主要就是看每个文件的这类注释是否有问题。

3.注册模块测试时出现

com.mysql.jdbc.exceptions.MySQLSyntaxErrorException

错误时，该怎么解决：

答：要考虑是不是database.properties文件中的username=root ；password=123 是不是自己数据库设置的数据。

# 经典知识

# String详解

## 1.1String类常量池详解

|  |
| --- |
| public class test1 {  public static void main(String[] args){  String a = "a1";  String b = "a"+ 1;  System.out.println(a==b);  }//true  } |
| public class test2 {  public static void main(String[] args){  String a = "ab";  String bb = "b";  String b = "a"+ bb; //编译器不能确定为常量  System.out.println(a==b);  }//false  } |
| public class test3 {  public static void main(String[] args){  String a = "ab";  final String bb = "b";  String b = "a"+ bb; //bb加final后是常量，可以在编译器确定b  System.out.println(a==b);  }//true  } |
| public class test4 {  public static void main(String[] args){  String a = "ab";  final String bb = getBB();  String b = "a"+ bb;//bb是通过函数返回的，虽然知道它是final的，但不知道具体是啥，要到运行期才知道bb的值  System.out.println(a==b);  }//false  private static String getBB(){ return "b"; }  } |
| public class test6 {  private static String a = new String("ab");  public static void main(String[] args){  String s1 = "a";  String s2 = "b";  String s = s1 + s2;  System.out.println(s == a);  System.out.println(s.intern() == a);  System.out.println(s.intern() == a.intern());  }//flase false true  } |

（1）String常量池详解：

　　1.String使用private final char value[]来实现字符串的存储,也就是说String对象创建之后,就不能再修改此对象中存储的字符串内容,就是因为如此,才说String类型是不可变的(immutable)。String类有一个特殊的创建方法,就是使用""双引号来创建.例如new String("i am")实际创建了2个

　　String对象,一个是"i am"通过""双引号创建的,另一个是通过new创建的.只不过他们创建的时期不同,

通过""双引号创建的是编译期,创建在了常量池中，通过new创建的是运行期，创建在了堆中。所以想向string的常量池中添加数据需要在编译器完成，但是运行期调用String类的intern()方法的话，也可以向String Pool中动态添加对象。另外，常量池不能添加重复数据。

java对String类型重载了+操作符,可以直接使用+操作符对两个字符串进行连接。

所以：

|  |
| --- |
| String s1 = "sss111";  String s2 = "sss111";  System.out.println(s1 == s2); //结果为true  因为引号创建是在编译器创建在了常量池中，而常量池不允许存储相同的数据，所以s1，s2必定指向了同一个地址，所以结果是true。 |
| String s1 = new String("sss111");  String s2 = "sss111";  System.out.println(s1 == s2); //结果为false  因为new是在运行期创建在堆中，而s2在编译期创建在了常量池中，所以s1，s2指向的不会是一个对象，更不会是同一个地址，所以false。 |
| String s1 = new String("sss111");  s1 = s1.intern();  String s2 = "sss111";  System.out.println(s1 == s2);//结果为true |
| String s1 = new String("111");  String s2 = "sss111";  String s3 = "sss" + "111";  String s4 = "sss" + s1;  System.out.println(s2 == s3); //true  System.out.println(s2 == s4); //false  System.out.println(s2 == s4.intern()); //true |

结果上面分析,总结如下:

1.单独使用""引号创建的字符串都是常量,编译期就已经确定存储到String Pool中；

　 2，使用new String("")创建的对象会存储到heap中,是运行期新创建的；

3，使用只包含常量的字符串连接符如"aa" + "aa"创建的也是常量,编译期就能确定,已经确定存储到String Pool中；

4，使用包含变量的字符串连接符如"aa" + s1创建的对象是运行期才创建的,存储在heap中；

　　还有几个经常考的面试题:

　　 String s1 = new String("s1") ;

　　 String s2 = new String("s1") ;

　　上面创建了几个String对象?

　　答案:3个 ,编译期Constant Pool中创建1个,运行期heap中创建2个.（用new创建的每new一次就在堆上创建一个对象，用引号创建的如果在常量池中已有就直接指向，不用创建）

　　String s1 = "s1";

　　String s2 = s1;

　　s2 = "s2";

　　s1指向的对象中的字符串是什么?

　　答案: "s1"。（永远不要忘了String不可变的，s2 = "s2";实际上s2的指向就变了，因为你不可以去改变一个String，）

（2）String是一个特殊的包装类数据。可以用：

String str = new String("abc"); String str = "abc";

两种的形式来创建，第一种是用new()来新建对象的，它会在存放于堆中。每调用一次就会创建一个新的对象。

而第二种是先在栈中创建一个对String类的对象引用变量str，然后通过符号引用去字符串常量池里找有没有"abc",如果没有，则将"abc"存放进字符串常量池，并令str指向”abc”，如果已经有”abc” 则直接令str指向“abc”。

比较类里面的数值是否相等时，用equals()方法；当测试两个包装类的引用是否指向同一个对象时，用==，下面用例子说明上面的理论。

String str1 = "abc";

String str2 = "abc";

System.out.println(str1==str2); //true

可以看出str1和str2是指向同一个对象的。

String str1 =new String ("abc");

String str2 =new String ("abc");

System.out.println(str1==str2); // false

用new的方式是生成不同的对象。每一次生成一个。

因此用String str1 = "abc"; 方式创建多个”abc”字符串,在内存中其实只存在一个对象而已. 这种写法有利与节省内存空间. 同时它可以在一定程度上提高程序的运行速度，因为JVM会自动根据栈中数据的实际情况来决定是否有必要创建新对象。而对于String str = new String("abc")；的代码，则一概在堆中创建新对象，而不管其字符串值是否相等，是否有必要创建新对象，从而加重了程序的负担。

另一方面, 要注意: 我们在使用诸如String str = "abc"；的格式定义类时，总是想当然地认为，创建了String类的对象str。担心陷阱！对象可能并没有被创建！而可能只是指向一个先前已经创建的 对象。只有通过new()方法才能保证每次都创建一个新的对象。

（3）由于String类的immutable性质，当String变量需要经常变换其值时，应该考虑使用StringBuffer类，以提高程序效率。

1. 首先String不属于8种基本数据类型，String是一个对象。 因为对象的默认值是null，所以String的默认值也是null；但它又是一种特殊的对象，有其它对象没有的一些特性。

2. new String()和new String(”")都是申明一个新的空字符串，是空串不是null；

（4） String.intern()：

再补充介绍一点：存在于.class文件中的常量池，在运行期被JVM装载，并且可以扩充。String的intern()方法就是扩充常量池的一个方法；当一个String实例str调用intern()方法时，Java查找常量池中是否有相同Unicode的字符串常量，如果有，则返回其的引用，如果没有，则在常量池中增加一个Unicode等于str的字符串并返回它的引用；看例3就清楚了

例3：

String s0= "kvill";

String s1=new String("kvill");

String s2=new String("kvill");

System.out.println( s0==s1 );

System.out.println( "\*\*\*\*\*\*\*\*\*\*" );

s1.intern();

s2=s2.intern(); //把常量池中"kvill"的引用赋给s2

System.out.println( s0==s1);

System.out.println( s0==s1.intern() );

System.out.println( s0==s2 );

结果为：

false

\*\*\*\*\*\*\*\*\*\*

false //虽然执行了s1.intern(),但它的返回值没有赋给s1

true //说明s1.intern()返回的是常量池中"kvill"的引用

true

（5）我再破除一个错误的理解：有人说，“使用 String.intern() 方法则可以将一个 String 类的保存到一个全局 String 表中 ，如果具有相同值的 Unicode 字符串已经在这个表中，那么该方法返回表中已有字符串的地址，如果在表中没有相同值的字符串，则将自己的地址注册到表中”如果我把他说的这个全局的 String 表理解为常量池的话，他的最后一句话，”如果在表中没有相同值的字符串，则将自己的地址注册到表中”是错的：

看例4：

String s1=new String("kvill");

String s2=s1.intern();

System.out.println( s1==s1.intern() );

System.out.println( s1+" "+s2 );

System.out.println( s2==s1.intern() );

结果：

false

kvill kvill

true

在这个类中我们没有声名一个”kvill”常量，所以常量池中一开始是没有”kvill”的，当我们调用s1.intern()后就在常量池中新添加了一个”kvill”常量，原来的不在常量池中的”kvill”仍然存在，也就不是“将自己的地址注册到常量池中”了。 s1==s1.intern()为false说明原来的”kvill”仍然存在；s2现在为常量池中”kvill”的地址，所以有s2==s1.intern()为true。

# 2.Spring

## 2.1注入理解，注入继承问题

（1）知识补充：

（1.1）当使用property--set属性注入的时候

<property name="jt" ref="jdbcTemplate"></property>，

Spring操作时是根据property注入而不是field，所以不管bean中定义的被注入类有无定义该属性private JdbcTemplate jt;，只要有对应的set方法

public void setJt(JdbcTemplate jt) {

this.jt = jt;

}

就会成功注入。

（1.2）注入后在spring空间内就会生成带有依赖关系的set属性对应的对象，如：

<bean name="jt" class="org.springframework.jdbc.core.JdbcTemplate">

<property name="dataSource" ref="dataSource">

</property>

空间中就会留有已经注入dataSource的JdbcTemplate对象，所以即使在另一个daoimpl类中使用 private JdbcTemplate jt建立JdbcTemplate对象，只要配置文件中配置了JdbcTemplate的自动注入：

<bean name="userDao" class="com.qf.springjdbc.dao.impl.UserDaoImpl">

<property name="jt" ref="jdbcTemplate"></property>

</bean>

jt中就会带有dataSource的信息，这就为什么不设置自动注入而产生空指针异常，另外如果定义private JdbcTemplate jt=new JdbcTemplate（）；那么这就建立了新对象，就不会和spring空间带有dataSource的JdbcTemplate对象进行匹配了。

（1.3）JdbcDaoSupport的使用

当我们实现层继承JdbcDaoSupport后，spring初始化的时候就会对该函数里的JdbcTemplate进行检查，所以虽然我们在daoimpl中没有使用JdbcDaoSupport的JdbcTemplate，而是自己定义了对象jt，

private JdbcTemplate jt;

public void setJt(JdbcTemplate jt) {

this.jt = jt;

}

但如果我们不给JdbcDaoSupport的JdbcTemplate注入dataSource，就会报错，

|  |
| --- |
| java.lang.IllegalStateException: Failed to load ApplicationContext  Caused by: java.lang.IllegalArgumentException: 'dataSource' or 'jdbcTemplate' is required  Caused by: org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'userDao' defined in class path resource [applicationContext.xml]: Invocation of init method failed; nested exception is java.lang.IllegalArgumentException: 'dataSource' or 'jdbcTemplate' is required |

由于daoimpl继承了JdbcDaoSupport，所以也就继承了它的set方法，因此在bean对象为daoimpl类中就可以为JdbcDaoSupport注入带有dataSource的jdbcTemplate对象或者dataSource（两者只需注入一个就可以，JdbcDaoSupport底层将jdbcTemplate和dataSource已经做了关联）

<bean name="userDao" class="com.qf.springjdbc.dao.impl.UserDaoImpl">

<property name="jt" ref="jdbcTemplate"></property>

<property name="dataSource" ref="dataSource" />

<!--<property name="jdbcTemplate" ref="jdbcTemplate" />-->

</bean>

（2）示例演示

（2.1）创建user.java

|  |
| --- |
| package com.qf.springjdbc.bean;  public class user {  private Integer id;  private String username;  private String gender;    //......自生成方法.......  } |

（2）创建 Dao.java 和 DaoImpl.java

|  |
| --- |
| package com.qf.springjdbc.dao;  import com.qf.springjdbc.bean.User;  import java.util.List;  public interface UserDao {  void save(User user);  void delete(Integer id);  void update(User user);  void getById(Integer id);  List<User> getAll();  int getTotalCount();  } |
| package com.qf.springjdbc.dao.impl;  import com.qf.springjdbc.bean.User;  import com.qf.springjdbc.dao.UserDao;  import org.springframework.jdbc.core.JdbcTemplate;  import org.springframework.jdbc.core.RowMapper;  import org.springframework.jdbc.core.support.JdbcDaoSupport;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.util.List;  public class UserDaoImpl extends JdbcDaoSupport implements UserDao{  /\*在继承的类JdbcDaoSupport中有一个final的setJdbcTemplate方法，所以我们自己定义的私有成员变量的名字不能是jdbcTemplate，否则会因为名字冲突而无法添加自己定义的setter方法  private JdbcTemplate jdbcTemplate;  public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {  this.jdbcTemplate = jdbcTemplate;  }\*/  private JdbcTemplate jt;  public void setJt(JdbcTemplate jt) {  this.jt = jt;  }  @Override  public void save(User user) {  String sql="insert into tb\_user(username,gender)values(?,?)";  jt.update(sql,user.getUsername(),user.getGender());  }  @Override  public void delete(Integer id) {  String sql="delete from tb\_user where id=?";  jt.update(sql,id);  }  @Override  public void update(User user) {  String sql="update tb\_user set username=?,gender=? where id=?";  jt.update(sql,user.getUsername(),user.getGender(),user.getId());  }  @Override  public User getById(Integer id) {  String sql = "select \* from tb\_user where id=?";  User user=jt.queryForObject(sql, new RowMapper<User>() {  @Override  public User mapRow(ResultSet rs, int rowNum) throws SQLException {  return mapUserHandle(rs);  }  },id);  return user;  }  @Override  public List<User> getAll() {  String sql="select \* from tb\_user";  List<User> userList=jt.query(sql,new RowMapper<User>() {  public User mapRow(ResultSet rs, int rowNum) throws SQLException {  return mapUserHandle(rs);  }  });  return userList;  }  @Override  public int getTotalCount() {  String sql="select count(\*) from tb\_user";  Integer integer=jt.queryForObject(sql,Integer.class);  return integer;  }  //将查询语句的公共部分抽取出来  public User mapUserHandle(ResultSet rs) throws SQLException {  User user=new User();  user.setUsername(rs.getString("username"));  user.setGender(rs.getString("gender"));  user.setId(rs.getInt("id"));  return user;  }  } |
| 优化dao.impl  进一步抽取查询语句，定义一个类来实现RowMapper接口的方法，由于该接口只有该函数使用，故可以考虑建成内部类。  原函数：  public User getById(Integer id) {  String sql = "select \* from tb\_user where id=?";  User user=jt.queryForObject(sql, new RowMapper<User>() {  @Override  public User mapRow(ResultSet rs, int rowNum) throws SQLException {  return mapUserHandle(rs);  }  },id);  return user;  }  public User mapUserHandle(ResultSet rs) throws SQLException {  User user=new User();  user.setUsername(rs.getString("username"));  user.setGender(rs.getString("gender"));  user.setId(rs.getInt("id"));  return user;  } |
| 优化修改后函数：  public List<User> getAll() {  String sql="select \* from tb\_user";  List<User> userList=jt.query(sql,new InnerClass());  return userList;  }  public User mapUserHandle(ResultSet rs) throws SQLException {  User user=new User();  user.setUsername(rs.getString("username"));  user.setGender(rs.getString("gender"));  user.setId(rs.getInt("id"));  return user;  }  private class InnerClass implements RowMapper{  public Object mapRow(ResultSet rs, int rowNum) throws SQLException {  return mapUserHandle(rs);  }  } |

（3）创建properties文件

|  |
| --- |
| jdbc.jdbcUrl=jdbc:mysql://localhost:3306/qf  jdbc.driverClass=com.mysql.jdbc.Driver  jdbc.user=root  jdbc.password=123 |

（4）配置applicationContext.xml

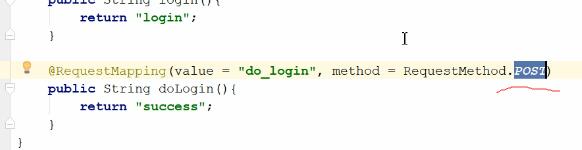
|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">  <!--指定数据源和配置文件-->  <context:property-placeholder location="db.properties"/>  <!--数据库连接池,class必须是全称-->  <bean name="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="driverClass" value="${jdbc.driverClass}"></property>  <property name="jdbcUrl" value="${jdbc.jdbcUrl}"></property>  <property name="user" value="${jdbc.user}"></property>  <property name="password" value="${jdbc.password}"></property>  </bean>  <!--注入区,构造函数用constructor-arg注入，set函数用property注入-->  <!--jdbcTemplate需要注入datasource-->  <bean name="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  <!-- name要和函数中的set后面的属性一致,ref关联的是上边连接池的id，即name-->  <property name="dataSource" ref="dataSource"></property>  </bean>  <!--daoimpl需要注入jdbcTemplate-->  <!--要注入到实现层，不要注入到接口层，接口层不能创建对象，没用-->  <bean name="userDao" class="com.qf.springjdbc.dao.impl.UserDaoImpl">  <!--将带有dataSource的jdbcTemplate关联到Daoimpl类的jt属性-->  <property name="jt" ref="jdbcTemplate"></property>  <!--为父类JdbcDaoSupport的jdbcTemplate或dataSource属性注入依赖，注入其中之一即可，JdbcDaoSupport底层为这两个属性做了关联-->  <!--为父类JdbcDaoSupport注入，为什么在UserDaoImpl类中写：  UserDaoImpl继承了JdbcDaoSupport的set方法，所以在子类里注入完全可以。  -->  <property name="dataSource" ref="dataSource" />  <!--<property name="jdbcTemplate" ref="jdbcTemplate" />-->  </bean>  </beans> |

（5）测试配置和daoimpl可行性：

|  |
| --- |
| package com.qf.springjdbc.test;  import com.qf.springjdbc.bean.User;  import com.qf.springjdbc.dao.UserDao;  import com.qf.springjdbc.dao.impl.UserDaoImpl;  import org.aspectj.apache.bcel.util.ClassPath;  import org.junit.Test;  import org.junit.runner.RunWith;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.test.context.ContextConfiguration;  import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  import java.util.List;  /\*\*  \* Created by Administrator on 2018/6/9.  \*/  @RunWith(SpringJUnit4ClassRunner.class)  @ContextConfiguration("classpath:applicationContext.xml")  public class TestXml {  @Autowired  UserDao userDao;  @Test  public void testSave(){  User user=new User();  user.setGender("男10");  user.setUsername("葫芦娃10");  userDao.save(user);  }  @Test  public void testDelete(){  userDao.delete(9);  }  @Test  public void testUpdate(){  User user=new User();  user.setGender("男1");  user.setUsername("葫芦娃1");  user.setId(1);  userDao.update(user);  }  @Test  public void testGetById(){  User user=userDao.getById(1);  System.out.println(user);  }  @Test  public void testGetAll(){  List<User> users=userDao.getAll();  System.out.println(users);  }  @Test  public void testGetTotalCount(){  int totalCount=userDao.getTotalCount();  System.out.println(totalCount);  }  } |

# 3.HTTP常见状态响应码

（1）405 是指服务器设定的接收请求方式与客户端设定的发送请求方式不一致。比如服务器设置只接收post请求方式，



而jsp页面设置的发送请求方式为get，



则就会报405错误。



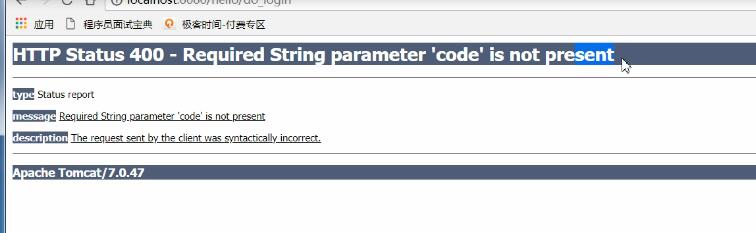
（2）200 成功访问

（3）404 页面找不到

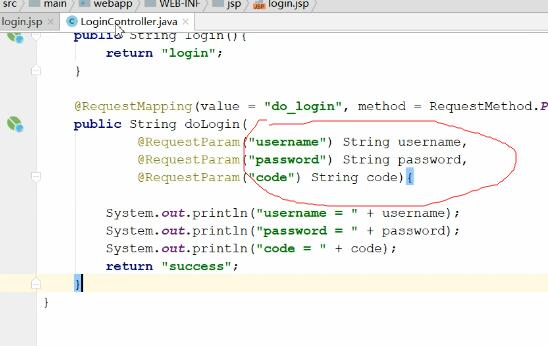
（4）500服务器错误

（5）302缓存

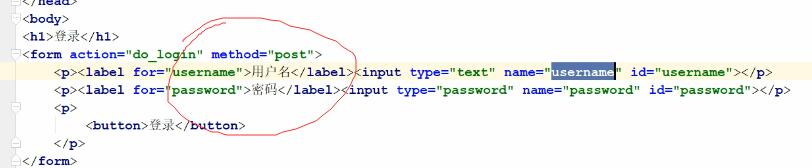
（6）400参数错误，凡是400错误，就一定是参数出现问题，比如参数丢了找不到了，参数类型不对应等等。



比如：后台请求参数中有验证码code的接收：



而前端页面并没有设置验证码的发送请求的功能：

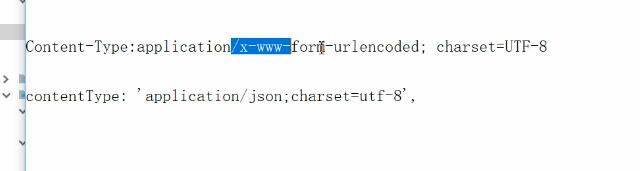


这时候测试必然会报400错误。

（7）415 错误如下图所示：



造成的原因是框架提供的格式与页面接收的格式不匹配，所以需要在返回json语句的程序中要定义contentType:"application/json;charset=utf-8",



# 4.资源信息集锦

（1）架构师之路微信公众号 沈剑 58架构工程师 数据库30条军规

## 4.1前端学习资源

（1）https://v3.bootcss.com/

（2）Bootstrap表格参数属性列表：

http://bootstrap-table.wenzhixin.net.cn/zh-cn/documentation/

## 4.2后端学习资源

### 4.2.1shiro

（1）项目整合shiro的过滤器的完整参考：

<http://blog.csdn.net/jadyer/article/details/12172839>

1. shiro验证授权缓存参考：

Ehcache配置文件的详细说明

http://blog.csdn.net/mlitsn/article/details/1909192

缓存存活时间和失效时间：

http://www.cnblogs.com/sprinkle/p/6539086.html

设置缓存的大小

http://elim.iteye.com/blog/2116749

### 4.2.2webService

（1）Webservice服务网站： http://www.webxml.com.cn（首页被黑，还是用下面的博客吧）

<https://blog.csdn.net/qq_20545159/article/details/47903513>

### 4.2.3dubbo

<http://dubbo.apache.org/#/docs/user/quick-start.md?lang=zh-cn>

### 4.2.4maven

<http://mvnrepository.com/>

### 4.2.5Quartz

定时任务cro在线生成器：例如0/1 32-33 \* \* \* ? \*

<http://cron.qqe2.com/>

# 5 Java 面试 36 题精讲

## 5.1谈谈你对Java平台的理解？

（1）从你接触 Java 开发到现在，你对 Java 最直观的印象是什么呢？是它宣传的 “Write once, run anywhere”，还是目前看已经有些过于形式主义的语法呢？你对于 Java 平台到底了解到什么程度？请你先停下来总结思考一下。

今天我要问你的问题是，谈谈你对 Java 平台的理解？“Java 是解释执行”，这句话正确吗？

（2）典型回答

Java 本身是一种面向对象的语言，最显著的特性有两个方面，一是所谓的“**书写一次，到处运行**”（Write once, run anywhere），能够非常容易地获得跨平台能力；另外就是**垃圾收集**（GC, Garbage Collection），Java 通过垃圾收集器（Garbage Collector）回收分配内存，大部分情况下，程序员不需要自己操心内存的分配和回收。

我们日常会接触到 JRE（Java Runtime Environment）或者 JDK（Java Development Kit）。 JRE，也就是 Java 运行环境，包含了 JVM 和 Java 类库，以及一些模块等。而 JDK 可以看作是 JRE 的一个超集，提供了更多工具，比如编译器、各种诊断工具等。

对于“Java 是解释执行”这句话，这个说法不太准确。我们开发的 Java 的源代码，首先通过 Javac 编译成为字节码（bytecode），然后，在运行时，通过 Java 虚拟机（JVM）内嵌的解释器将字节码转换成为最终的机器码。但是常见的 JVM，比如我们大多数情况使用的 Oracle JDK 提供的 Hotspot JVM，都提供了 JIT（Just-In-Time）编译器，也就是通常所说的动态编译器，JIT 能够在运行时将热点代码编译成机器码，这种情况下部分热点代码就属于**编译执行**，而不是解释执行了。

（3）考点分析

其实这个问题，问得有点笼统。题目本身是非常开放的，往往考察的是多个方面，比如，基础知识理解是否很清楚；是否掌握 Java 平台主要模块和运行原理等。很多面试者会在这种问题上吃亏，稍微紧张了一下，不知道从何说起，就给出个很简略的回答。

对于这类笼统的问题，你需要尽量**表现出自己的思维深入并系统化，Java 知识理解得也比较全面**，一定要避免让面试官觉得你是个“知其然不知其所以然”的人。毕竟明白基本组成和机制，是日常工作中进行问题诊断或者性能调优等很多事情的基础，相信没有招聘方会不喜欢“热爱学习和思考”的面试者。

即使感觉自己的回答不是非常完善，也不用担心。我个人觉得这种笼统的问题，有时候回答得稍微片面也很正常，大多数有经验的面试官，不会因为一道题就对面试者轻易地下结论。通常会尽量引导面试者，把他的真实水平展现出来，这种问题就是做个开场热身，面试官经常会根据你的回答扩展相关问题。

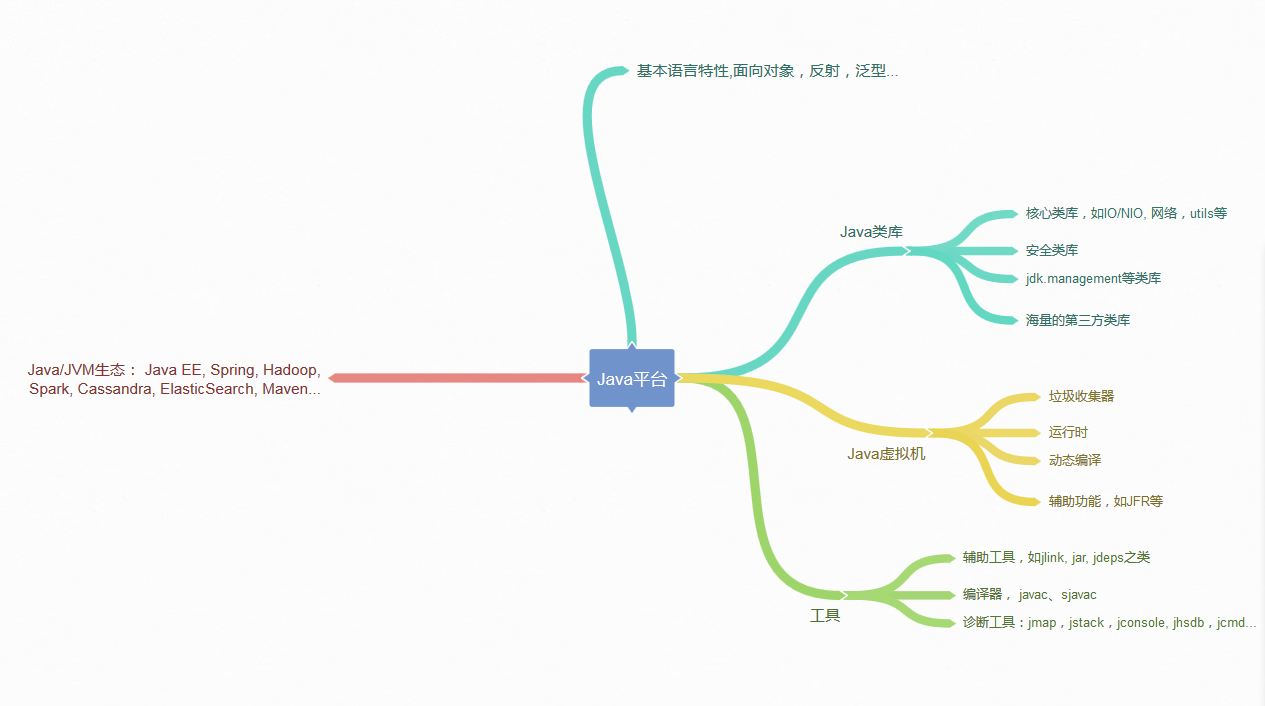
（3）知识扩展

回归正题，对于 Java 平台的理解，可以从很多方面简明扼要地谈一下，例如：Java 语言特性，包括泛型、Lambda 等语言特性；基础类库，包括集合、IO/NIO、网络、并发、安全等基础类库。对于我们日常工作应用较多的类库，面试前可以系统化总结一下，有助于临场发挥。

或者谈谈 JVM 的一些基础概念和机制，比如 Java 的类加载机制，常用版本 JDK（如 JDK 8）内嵌的 Class-Loader，例如 Bootstrap、 Application 和 Extension Class-loader；类加载大致过程：加载、验证、链接、初始化（这里参考了周志明的《深入理解 Java 虚拟机》，非常棒的 JVM 上手书籍）；自定义 Class-Loader 等。还有垃圾收集的基本原理，最常见的垃圾收集器，如 SerialGC、Parallel GC、 CMS、 G1 等，对于适用于什么样的工作负载最好也心里有数。这些都是可以扩展开的领域，我会在后面的专栏对此进行更系统的介绍。

当然还有 JDK 包含哪些工具或者 Java 领域内其他工具等，如编译器、运行时环境、安全工具、诊断和监控工具等。这些基本工具是日常工作效率的保证，对于我们工作在其他语言平台上，同样有所帮助，很多都是触类旁通的。

下图是我总结的一个相对宽泛的蓝图供你参考。



不再扩展了，回到前面问到的解释执行和编译执行的问题。有些面试官喜欢在特定问题上“刨根问底儿”，因为这是进一步了解面试者对知识掌握程度的有效方法，我稍微深入探讨一下。

众所周知，我们通常把 Java 分为编译期和运行时。这里说的 Java 的编译和 C/C++ 是有着不同的意义的，Javac 的编译，编译 Java 源码生成“.class”文件里面实际是字节码，而不是可以直接执行的机器码。Java 通过字节码和 Java 虚拟机（JVM）这种跨平台的抽象，屏蔽了操作系统和硬件的细节，这也是实现“一次编译，到处执行”的基础。

在运行时，JVM 会通过类加载器（Class-Loader）加载字节码，解释或者编译执行。就像我前面提到的，主流 Java 版本中，如 JDK 8 实际是解释和编译混合的一种模式，即所谓的混合模式（-Xmixed）。通常运行在 server 模式的 JVM，会进行上万次调用以收集足够的信息进行高效的编译，client 模式这个门限是 1500 次。Oracle Hotspot JVM 内置了两个不同的 JIT compiler，C1 对应前面说的 client 模式，适用于对于启动速度敏感的应用，比如普通 Java 桌面应用；C2 对应 server 模式，它的优化是为长时间运行的服务器端应用设计的。默认是采用所谓的分层编译（TieredCompilation）。这里不再展开更多 JIT 的细节，没必要一下子就钻进去，我会在后面介绍分层编译的内容。

Java 虚拟机启动时，可以指定不同的参数对运行模式进行选择。 比如，指定“-Xint”，就是告诉 JVM 只进行解释执行，不对代码进行编译，这种模式抛弃了 JIT 可能带来的性能优势。毕竟解释器（interpreter）是逐条读入，逐条解释运行的。与其相对应的，还有一个“-Xcomp”参数，这是告诉 JVM 关闭解释器，不要进行解释执行，或者叫作最大优化级别。那你可能会问这种模式是不是最高效啊？简单说，还真未必。“-Xcomp”会导致 JVM 启动变慢非常多，同时有些 JIT 编译器优化方式，比如分支预测，如果不进行 profiling，往往并不能进行有效优化。

除了我们日常最常见的 Java 使用模式，其实还有一种新的编译方式，即所谓的 AOT（Ahead-of-Time Compilation），直接将字节码编译成机器代码，这样就避免了 JIT 预热等各方面的开销，比如 Oracle JDK 9 就引入了实验性的 AOT 特性，并且增加了新的 jaotc 工具。利用下面的命令把某个类或者某个模块编译成为 AOT 库。

jaotc --output libHelloWorld.so HelloWorld.class

jaotc --output libjava.base.so --module java.base

然后，在启动时直接指定就可以了。

java -XX:AOTLibrary=./libHelloWorld.so,./libjava.base.so HelloWorld

而且，Oracle JDK 支持分层编译和 AOT 协作使用，这两者并不是二选一的关系。如果你有兴趣，可以参考相关文档：<http://openjdk.java.net/jeps/295>。AOT 也不仅仅是只有这一种方式，业界早就有第三方工具（如 GCJ、Excelsior JET）提供相关功能。

另外，JVM 作为一个强大的平台，不仅仅只有 Java 语言可以运行在 JVM 上，本质上合规的字节码都可以运行，Java 语言自身也为此提供了便利，我们可以看到类似 Clojure、Scala、Groovy、JRuby、Jython 等大量 JVM 语言，活跃在不同的场景。

今天，我简单介绍了一下 Java 平台相关的一些内容，目的是提纲挈领地构建一个整体的印象，包括 Java 语言特性、 核心类库与常用第三方类库、Java 虚拟机基本原理和相关工具，希望对你有所帮助。

**Helen的总结：**

**热点代码？？**

热点代码是指频繁运行的某个方法或者是方法块，为了优化热点代码的执行效率，使用JIT编译器将这些代码编译成机器码。

****解释执行和编译执行的区别？？****

计算机并不能直接地接受和执行用高级语言编写的源程序，源程序在输入计算机时，通过"翻译程序"翻译成机器语言形式的目标程序，计算机才能识别和执行。这种"翻译"通常有两种方式，即编译方式和解释方式。

**编译方式**是指利用事先编好的一个称为编译程序的机器语言程序，作为系统软件存放在计算机内，当用户将高级语言编写的源程序输入计算机后，编译程序便把源程序整个地翻译成用机器语言表示的与之等价的目标程序，然后计算机再执行该目标程序，以完成源程序要处理的运算并取得结果。

**解释方式**是指源程序进入计算机后，解释程序边扫描边解释，逐句输入逐句翻译，计算机一句句执行，并不产生目标程序。

**我理解的java程序执行步骤：（嗯嗯，简单易懂）**  
 首先javac编译器将源代码编译成字节码。  
 然后jvm类加载器加载字节码文件，然后通过解释器逐行解释执行，这种方式的执行速度相对会比较慢。有些方法和代码块是高频率调用的，也就是所谓的热点代码，所以引进jit技术，提前将这类字节码直接编译成本地机器码。这样类似于缓存技术，运行时再遇到这类代码直接可以执行，而不是先解释后执行。

## 5.2Exception和Error有什么区别？

### 5.2.1引言

（1）世界上存在永远不会出错的程序吗？也许这只会出现在程序员的梦中。随着编程语言和软件的诞生，异常情况就如影随形地纠缠着我们，只有正确处理好意外情况，才能保证程序的可靠性。

Java 语言在设计之初就提供了相对完善的异常处理机制，这也是 Java 得以大行其道的原因之一，因为这种机制大大降低了编写和维护可靠程序的门槛。如今，异常处理机制已经成为现代编程语言的标配。

今天我要问你的问题是，请对比 Exception 和 Error，另外，运行时异常与一般异常有什么区别？

### 5.2.2典型回答

Exception 和 Error 都是继承了 Throwable 类，在 Java 中只有 Throwable 类型的实例才可以被抛出（throw）或者捕获（catch），它是异常处理机制的基本组成类型。

Exception 和 Error 体现了 Java 平台设计者对不同异常情况的分类。

Exception 是程序正常运行中，可以预料的意外情况，可能并且应该被捕获，进行相应处理。

Error 是指在正常情况下，不大可能出现的情况，绝大部分的 Error 都会导致程序（比如 JVM 自身）处于非正常的、不可恢复状态。既然是非正常情况，所以不便于也不需要捕获，常见的比如 OutOfMemoryError 之类，都是 Error 的子类。

Exception 又分为**可检查**（checked）异常和**不检查**（unchecked）异常，可检查异常在源代码里必须显式地进行捕获处理，这是编译期检查的一部分。前面我介绍的不可查的 Error，是 Throwable 不是 Exception。

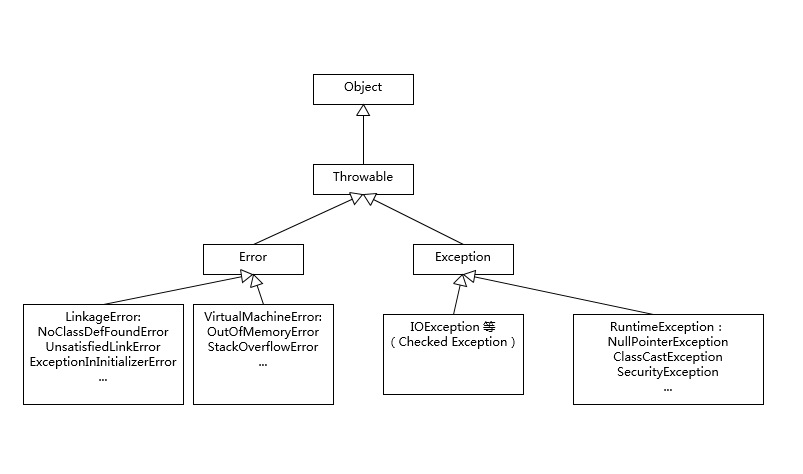
不检查异常就是所谓的运行时异常，类似 NullPointerException、ArrayIndexOutOfBoundsException 之类，通常是可以编码避免的逻辑错误，具体根据需要来判断是否需要捕获，并不会在编译期强制要求。

### 5.2.3考点分析

分析 Exception 和 Error 的区别，是从概念角度考察了 Java 处理机制。总的来说，还处于理解的层面，面试者只要阐述清楚就好了。我们在日常编程中，如何处理好异常是比较考验功底的，我觉得需要掌握两个方面。

第一，**理解 Throwable、Exception、Error 的设计和分类**。比如，掌握那些应用最为广泛的子类，以及如何自定义异常等。

很多面试官会进一步追问一些细节，比如，你了解哪些 Error、Exception 或者 RuntimeException？我画了一个简单的类图，并列出来典型例子，可以给你作为参考，至少做到基本心里有数。



其中有些子类型，最好重点理解一下，比如 NoClassDefFoundError 和 ClassNotFoundException 有什么区别，这也是个经典的入门题目。

第二，**理解 Java 语言中操作 Throwable 的元素和实践**。掌握最基本的语法是必须的，如 try-catch-finally 块，throw、throws 关键字等。与此同时，也要懂得如何处理典型场景。

异常处理代码比较繁琐，比如我们需要写很多千篇一律的捕获代码，或者在 finally 里面做一些资源回收工作。随着 Java 语言的发展，引入了一些更加便利的特性，比如 try-with-resources 和 multiple catch，具体可以参考下面的代码段。在编译时期，会自动生成相应的处理逻辑，比如，自动按照约定俗成 close 那些扩展了 AutoCloseable 或者 Closeable 的对象。

try (BufferedReader br = new BufferedReader(…);

BufferedWriter writer = new BufferedWriter(…)) {// Try-with-resources

// do something

catch ( IOException | XEception e) {// Multiple catch

// Handle it

}

### 5.2.4知识扩展

前面谈的大多是概念性的东西，下面我来谈些实践中的选择，我会结合一些代码用例进行分析。

**例1：**先开看第一个吧，下面的代码反映了异常处理中哪些不当之处？

try {

// 业务代码

// …

Thread.sleep(1000L);

} catch (Exception e) {

// Ignore it

}

这段代码虽然很短，但是已经违反了异常处理的两个基本原则。

第一，**尽量不要捕获类似 Exception 这样的通用异常，而是应该捕获特定异常**，在这里是 Thread.sleep() 抛出的 InterruptedException。

这是因为在日常的开发和合作中，我们读代码的机会往往超过写代码，软件工程是门协作的艺术，所以我们有义务让自己的代码能够直观地体现出尽量多的信息，而泛泛的 Exception 之类，恰恰隐藏了我们的目的。另外，我们也要保证程序不会捕获到我们不希望捕获的异常。比如，你可能更希望 RuntimeException 被扩散出来，而不是被捕获。

进一步讲，除非深思熟虑了，否则不要捕获 Throwable 或者 Error，这样很难保证我们能够正确程序处理 OutOfMemoryError。

第二，**不要生吞（swallow）异常**。这是异常处理中要特别注意的事情，因为很可能会导致非常难以诊断的诡异情况。

生吞异常，往往是基于假设这段代码可能不会发生，或者感觉忽略异常是无所谓的，但是千万不要在产品代码做这种假设！

如果我们不把异常抛出来，或者也没有输出到日志（Logger）之类，程序可能在后续代码以不可控的方式结束。没人能够轻易判断究竟是哪里抛出了异常，以及是什么原因产生了异常。

**例2：**再来看看第二段代码

try {

// 业务代码

// …

} catch (IOException e) {

e.printStackTrace();

}

这段代码作为一段实验代码，它是没有任何问题的，但是在产品代码中，通常都不允许这样处理。你先思考一下这是为什么呢？

我们先来看看[printStackTrace()](https://docs.oracle.com/javase/9/docs/api/java/lang/Throwable.html" \l "printStackTrace--)的文档，开头就是“Prints this throwable and its backtrace to the **standard error stream**”。问题就在这里，在稍微复杂一点的生产系统中，标准出错（STERR）不是个合适的输出选项，因为你很难判断出到底输出到哪里去了。

尤其是对于分布式系统，如果发生异常，但是无法找到堆栈轨迹（stacktrace），这纯属是为诊断设置障碍。所以，最好使用产品日志，详细地输出到日志系统里。

**例3：**我们接下来看下面的代码段，体会一下**Throw early, catch late 原则**。

public void readPreferences(String fileName){

//...perform operations...

InputStream in = new FileInputStream(fileName);

//...read the preferences file...

}

如果 fileName 是 null，那么程序就会抛出 NullPointerException，但是由于没有第一时间暴露出问题，堆栈信息可能非常令人费解，往往需要相对复杂的定位。这个 NPE 只是作为例子，实际产品代码中，可能是各种情况，比如获取配置失败之类的。在发现问题的时候，第一时间抛出，能够更加清晰地反映问题。

我们可以修改一下，让问题“throw early”，对应的异常信息就非常直观了。

public void readPreferences(String filename) {

Objects. requireNonNull(filename);

//...perform other operations...

InputStream in = new FileInputStream(filename);

//...read the preferences file...

}

至于“catch late”，其实是我们经常苦恼的问题，捕获异常后，需要怎么处理呢？最差的处理方式，就是我前面提到的“生吞异常”，本质上其实是掩盖问题。如果实在不知道如何处理，可以选择保留原有异常的 cause 信息，直接再抛出或者构建新的异常抛出去。在更高层面，因为有了清晰的（业务）逻辑，往往会更清楚合适的处理方式是什么。

有的时候，我们会根据需要自定义异常，这个时候除了保证提供足够的信息，还有两点需要考虑：

* 是否需要定义成 Checked Exception，因为这种类型设计的初衷更是为了从异常情况恢复，作为异常设计者，我们往往有充足信息进行分类。
* 在保证诊断信息足够的同时，也要考虑避免包含敏感信息，因为那样可能导致潜在的安全问题。如果我们看 Java 的标准类库，你可能注意到类似 java.net.ConnectException，出错信息是类似“ Connection refused (Connection refused)”，而不包含具体的机器名、IP、端口等，一个重要考量就是信息安全。类似的情况在日志中也有，比如，用户数据一般是不可以输出到日志里面的。

业界有一种争论（甚至可以算是某种程度的共识），Java 语言的 Checked Exception 也许是个设计错误，反对者列举了几点：

Checked Exception 的假设是我们捕获了异常，然后恢复程序。但是，其实我们大多数情况下，根本就不可能恢复。Checked Exception 的使用，已经大大偏离了最初的设计目的。

Checked Exception 不兼容 functional 编程，如果你写过 Lambda/Stream 代码，相信深有体会。

很多开源项目，已经采纳了这种实践，比如 Spring、Hibernate 等，甚至反映在新的编程语言设计中，比如 Scala 等。 如果有兴趣，你可以参考：

<http://literatejava.com/exceptions/checked-exceptions-javas-biggest-mistake/>。

当然，很多人也觉得没有必要矫枉过正，因为确实有一些异常，比如和环境相关的 IO、网络等，其实是存在可恢复性的，而且 Java 已经通过业界的海量实践，证明了其构建高质量软件的能力。我就不再进一步解读了，感兴趣的同学可以点击[链接](http://v.qq.com/x/page/d0635rf5x0o.html)，观看 Bruce Eckel 在 2018 年全球软件开发大会 QCon 的分享 Failing at Failing: How and Why We've Been Nonchalantly Moving Away From Exception Handling。

我们从性能角度来审视一下 Java 的异常处理机制，这里有两个可能会相对昂贵的地方：

try-catch 代码段会产生额外的性能开销，或者换个角度说，它往往会影响 JVM 对代码进行优化，所以建议仅捕获有必要的代码段，尽量不要一个大的 try 包住整段的代码；与此同时，利用异常控制代码流程，也不是一个好主意，远比我们通常意义上的条件语句（if/else、switch）要低效。

Java 每实例化一个 Exception，都会对当时的栈进行快照，这是一个相对比较重的操作。如果发生的非常频繁，这个开销可就不能被忽略了。

所以，对于部分追求极致性能的底层类库，有种方式是尝试创建不进行栈快照的 Exception。这本身也存在争议，因为这样做的假设在于，我创建异常时知道未来是否需要堆栈。问题是，实际上可能吗？小范围或许可能，但是在大规模项目中，这么做可能不是个理智的选择。如果需要堆栈，但又没有收集这些信息，在复杂情况下，尤其是类似微服务这种分布式系统，这会大大增加诊断的难度。

当我们的服务出现反应变慢、吞吐量下降的时候，检查发生最频繁的 Exception 也是一种思路。关于诊断后台变慢的问题，我会在后面的 Java 性能基础模块中系统探讨。

**总结：**

从一个常见的异常处理概念问题，简单总结了 Java 异常处理的机制。并结合代码，分析了一些普遍认可的最佳实践，以及业界最新的一些异常使用共识。最后，分析了异常性能开销。

（5）一课一练

关于今天我们讨论的题目你做到心中有数了吗？可以思考一个问题，对于异常处理编程，不同的编程范式也会影响到异常处理策略，比如，现在非常火热的反应式编程（Reactive Stream），因为其本身是异步、基于事件机制的，所以出现异常情况，决不能简单抛出去；另外，由于代码堆栈不再是同步调用那种垂直的结构，这里的异常处理和日志需要更加小心，我们看到的往往是特定 executor 的堆栈，而不是业务方法调用关系。对于这种情况，你有什么好的办法吗？

### 5.2.5精彩评论

在Java世界里，异常的出现让我们编写的程序运行起来更加的健壮，同时为程序在调试、运行期间发生的一些意外情况，提供了补救机会；即使遇到一些严重错误而无法弥补，异常也会非常忠实的记录所发生的这一切。以下是文章心得感悟:

不要推诿或延迟处理异常，就地解决最好，并且需要实实在在的进行处理，而不是只捕捉，不动作。

一个函数尽管抛出了多个异常，但是只有一个异常可被传播到调用端。最后被抛出的异常时唯一被调用端接收的异常，其他异常都会被吞没掩盖。如果调用端要知道造成失败的最初原因，程序之中就绝不能掩盖任何异常。

不要在finally代码块中处理返回值。

按照我们程序员的惯性认知：当遇到return语句的时候，执行函数会立刻返回。但是，在Java语言中，如果存在finally就会有例外。除了return语句，try代码块中的break或continue语句也可能使控制权进入finally代码块。

请勿在try代码块中调用return、break或continue语句。万一无法避免，一定要确保finally的存在不会改变函数的返回值。

函数返回值有两种类型：值类型与对象引用。对于对象引用，要特别小心，如果在finally代码块中对函数返回的对象成员属性进行了修改，即使不在finally块中显式调用return语句，这个修改也会作用于返回值上。

勿将异常用于控制流。

如无必要，勿用异常。

.异常：这种情况下的异常，可以通过完善任务重试机制，当执行异常时，保存当前任务信息加入重试队列。重试的策略根据业务需要决定，当达到重试上限依然无法成功，记录任务执行失败，同时发出告警。

日志：类比消息中间件，处在不同线程之间的同一任务，简单高效一点的做法可能是用traceId/requestId串联。有些日志系统本身支持MDC/NDC功能，可以串联相关联的日志。

Error:系统错误，虚拟机出错，我们处理不了，也不需要我们来处理。

Exception，可以捕获的异常，且作出处理。也就是要么捕获异常并作出处理，要么继续抛出异常。

RuntimeException，经常性出现的错误，可以捕获，并作出处理，可以不捕获，也可以不用抛出。ArrayIndexOutOfBoundsException像这种异常可以不捕获，为什么呢？在一个程序里，使用很多数组，如果使用一次捕获一次，则很累。

继承某个异常时，重写方法时，要么不抛出异常，要么抛出一模一样的异常。

当一个try后跟了很多个catch时，必须先捕获小的异常再捕获大的异常。

假如一个异常发生了，控制台打印了许多行信息，是因为程序中进行多层方法调用造成的。关键是看类型和行号。

上传下载不能抛异常。上传下载一定要关流。

异常不是错误。异常控制代码流程不利于代码简单易读。

try catch finally执行流程，与 return，break，continue等混合使用注意代码执行顺序。不是不可以，而是越是厉害的人，代码越容易理解。 NoClassDefFoundError是一个错误(Error)，而ClassNOtFoundException是一个异常，在Java中对于错误和异常的处理是不同的，我们可以从异常中恢复程序但却不应该尝试从错误中恢复程序。

ClassNotFoundException的产生原因：

Java支持使用Class.forName方法来动态地加载类，任意一个类的类名如果被作为参数传递给这个方法都将导致该类被加载到JVM内存中，如果这个类在类路径中没有被找到，那么此时就会在运行时抛出ClassNotFoundException异常。

解决该问题需要确保所需的类连同它依赖的包存在于类路径中，常见问题在于类名书写错误。

另外还有一个导致ClassNotFoundException的原因就是：当一个类已经某个类加载器加载到内存中了，此时另一个类加载器又尝试着动态地从同一个包中加载这个类。通过控制动态类加载过程，可以避免上述情况发生。

NoClassDefFoundError产生的原因在于：

如果JVM或者ClassLoader实例尝试加载（可以通过正常的方法调用，也可能是使用new来创建新的对象）类的时候却找不到类的定义。要查找的类在编译的时候是存在的，运行的时候却找不到了。这个时候就会导致NoClassDefFoundError.

造成该问题的原因可能是打包过程漏掉了部分类，或者jar包出现损坏或者篡改。解决这个问题的办法是查找那些在开发期间存在于类路径下但在运行期间却不在类路径下的类。

NoClassDefFoundError 和 ClassNotFoundException 有什么区别

前者是找不到类的定义，类文件还在；后者是找不到class文件；前者可能是在类初始化的时候（比如初始化某个变量报错）发生异常；后者可能是找不到引用的类文件，可能是某个包没有导入.前者是运行时异常，后者是可检查异常

## 5.3谈谈final、finally、 finalize有什么不同？

##### 1.3.1.3.1典型回答

（1）引入

Java 语言有很多看起来很相似，但是用途却完全不同的语言要素，这些内容往往容易成为面试官考察你知识掌握程度的切入点。今天，我要问你的是一个经典的 Java 基础题目，谈谈 final、finally、 finalize 有什么不同？

（2）典型回答

final 可以用来修饰类、方法、变量，分别有不同的意义，final 修饰的 class 代表不可以继承扩展，final 的变量是不可以修改的，而 final 的方法也是不可以重写的（override）。

finally 则是 Java 保证重点代码一定要被执行的一种机制。我们可以使用 try-finally 或者 try-catch-finally 来进行类似关闭 JDBC 连接、保证 unlock 锁等动作。

finalize 是基础类 java.lang.Object 的一个方法，它的设计目的是保证对象在被垃圾收集前完成特定资源的回收。finalize 机制现在已经不推荐使用，并且在 JDK 9 开始被标记为 deprecated。

##### 1.3.1.3.2**考点分析**

这是一个非常经典的 Java 基础问题，我上面的回答主要是从语法和使用实践角度出发的，其实还有很多方面可以深入探讨，面试官还可以考察你对性能、并发、对象生命周期或垃圾收集基本过程等方面的理解。

推荐使用 final 关键字来明确表示我们代码的语义、逻辑意图，这已经被证明在很多场景下是非常好的实践，比如：

我们可以将方法或者类声明为 final，这样就可以明确告知别人，这些行为是不许修改的。

如果你关注过 Java 核心类库的定义或源码， 有没有发现 java.lang 包下面的很多类，相当一部分都被声明成为 final class？在第三方类库的一些基础类中同样如此，这可以有效避免 API 使用者更改基础功能，某种程度上，这是保证平台安全的必要手段。

使用 final 修饰参数或者变量，也可以清楚地避免意外赋值导致的编程错误，甚至，有人明确推荐将所有方法参数、本地变量、成员变量声明成 final。

final 变量产生了某种程度的不可变（immutable）的效果，所以，可以用于保护只读数据，尤其是在并发编程中，因为明确地不能再赋值 final 变量，有利于减少额外的同步开销，也可以省去一些防御性拷贝的必要。

final 也许会有性能的好处，很多文章或者书籍中都介绍了可在特定场景提高性能，比如，利用 final 可能有助于 JVM 将方法进行内联，可以改善编译器进行条件编译的能力等等。坦白说，很多类似的结论都是基于假设得出的，比如现代高性能 JVM（如 HotSpot）判断内联未必依赖 final 的提示，要相信 JVM 还是非常智能的。类似的，final 字段对性能的影响，大部分情况下，并没有考虑的必要。

从开发实践的角度，我不想过度强调这一点，这是和 JVM 的实现很相关的，未经验证比较难以把握。我的建议是，在日常开发中，除非有特别考虑，不然最好不要指望这种小技巧带来的所谓性能好处，程序最好是体现它的语义目的。如果你确实对这方面有兴趣，可以查阅相关资料，我就不再赘述了，不过千万别忘了验证一下。

对于 finally，明确知道怎么使用就足够了。需要关闭的连接等资源，更推荐使用 Java 7 中添加的 try-with-resources 语句，因为通常 Java 平台能够更好地处理异常情况，编码量也要少很多，何乐而不为呢。

另外，我注意到有一些常被考到的 finally 问题（也比较偏门），至少需要了解一下。比如，下面代码会输出什么？

try {

// do something

System.exit(1);

} finally{

System.out.println(“Print from finally”);

}

上面 finally 里面的代码可不会被执行的哦，这是一个特例。

对于 finalize，我们要明确它是不推荐使用的，业界实践一再证明它不是个好的办法，在 Java 9 中，甚至明确将 Object.finalize() 标记为 deprecated！如果没有特别的原因，不要实现 finalize 方法，也不要指望利用它来进行资源回收。

为什么呢？简单说，你无法保证 finalize 什么时候执行，执行的是否符合预期。使用不当会影响性能，导致程序死锁、挂起等。

通常来说，利用上面的提到的 try-with-resources 或者 try-finally 机制，是非常好的回收资源的办法。如果确实需要额外处理，可以考虑 Java 提供的 Cleaner 机制或者其他替代方法。接下来，我来介绍更多设计考虑和实践细节。

##### 1.3.1.3.3**知识扩展**

**1. 注意，final 不是 immutable！**

我在前面介绍了 final 在实践中的益处，需要注意的是，**final 并不等同于 immutable**，比如下面这段代码：

final List<String> strList = new ArrayList<>();

strList.add("Hello");

strList.add("world");

List<String> unmodifiableStrList = List.of("hello", "world");

unmodifiableStrList.add("again");

final 只能约束 strList 这个引用不可以被赋值，但是 strList 对象行为不被 fi例子中，[List.of 方法](http://openjdk.java.net/jeps/269)创建的本身nal 影响，添加元素等操作是完全正常的。如果我们真的希望对象本身是不可变的，那么需要相应的类支持不可变的行为。在上面这个就是不可变 List，最后那句 add 是会在运行时抛出异常的。

Immutable 在很多场景是非常棒的选择，某种意义上说，Java 语言目前并没有原生的不可变支持，如果要实现 immutable 的类，我们需要做到：

将 class 自身声明为 final，这样别人就不能扩展来绕过限制了。

将所有成员变量定义为 private 和 final，并且不要实现 setter 方法。

通常构造对象时，成员变量使用深度拷贝来初始化，而不是直接赋值，这是一种防御措施，因为你无法确定输入对象不被其他人修改。

如果确实需要实现 getter 方法，或者其他可能会返回内部状态的方法，使用 copy-on-write 原则，创建私有的 copy。

这些原则是不是在并发编程实践中经常被提到？的确如此。

关于 setter/getter 方法，很多人喜欢直接用 IDE 一次全部生成，建议最好是你确定有需要时再实现。

**finalize 真的那么不堪？**

前面简单介绍了 finalize 是一种已经被业界证明了的非常不好的实践，那么为什么会导致那些问题呢？

finalize 的执行是和垃圾收集关联在一起的，一旦实现了非空的 finalize 方法，就会导致相应对象回收呈现数量级上的变慢，有人专门做过 benchmark，大概是 40~50 倍的下降。

因为，finalize 被设计成在对象**被垃圾收集前**调用，这就意味着实现了 finalize 方法的对象是个“特殊公民”，JVM 要对它进行额外处理。finalize 本质上成为了快速回收的阻碍者，可能导致你的对象经过多个垃圾收集周期才能被回收。

有人也许会问，我用 System.runFinalization() 告诉 JVM 积极一点，是不是就可以了？也许有点用，但是问题在于，这还是不可预测、不能保证的，所以本质上还是不能指望。实践中，因为 finalize 拖慢垃圾收集，导致大量对象堆积，也是一种典型的导致 OOM 的原因。

从另一个角度，我们要确保回收资源就是因为资源都是有限的，垃圾收集时间的不可预测，可能会极大加剧资源占用。这意味着对于消耗非常高频的资源，千万不要指望 finalize 去承担资源释放的主要职责，最多让 finalize 作为最后的“守门员”，况且它已经暴露了如此多的问题。这也是为什么我推荐，**资源用完即显式释放，或者利用资源池来尽量重用**。

finalize 还会掩盖资源回收时的出错信息，我们看下面一段JDK 的源代码，截取自 java.lang.ref.Finalizer

private void runFinalizer(JavaLangAccess jla) {

// ... 省略部分代码

try {

Object finalizee = this.get();

if (finalizee != null && !(finalizee instanceof java.lang.Enum)) {

jla.invokeFinalize(finalizee);

// Clear stack slot containing this variable, to decrease

// the chances of false retention with a conservative GC

finalizee = null;

}

} catch (Throwable x) { }

super.clear();

}

结合我上期专栏介绍的异常处理实践，你认为这段代码会导致什么问题？

是的，你没有看错，这里的**Throwable 是被生吞了的！**也就意味着一旦出现异常或者出错，你得不到任何有效信息。况且，Java 在 finalize 阶段也没有好的方式处理任何信息，不然更加不可预测。

**3. 有什么机制可以替换 finalize 吗？**

Java 平台目前在逐步使用 java.lang.ref.Cleaner 来替换掉原有的 finalize 实现。Cleaner 的实现利用了幻象引用（PhantomReference），这是一种常见的所谓 post-mortem 清理机制。我会在后面的专栏系统介绍 Java 的各种引用，利用幻象引用和引用队列，我们可以保证对象被彻底销毁前做一些类似资源回收的工作，比如关闭文件描述符（操作系统有限的资源），它比 finalize 更加轻量、更加可靠。

吸取了 finalize 里的教训，每个 Cleaner 的操作都是独立的，它有自己的运行线程，所以可以避免意外死锁等问题。

实践中，我们可以为自己的模块构建一个 Cleaner，然后实现相应的清理逻辑。下面是 JDK 自身提供的样例程序：

public class CleaningExample implements AutoCloseable {

// A cleaner, preferably one shared within a library

private static final Cleaner cleaner = <cleaner>;

static class State implements Runnable {

State(...) {

// initialize State needed for cleaning action

}

public void run() {

// cleanup action accessing State, executed at most once

}

}

private final State;

private final Cleaner.Cleanable cleanable

public CleaningExample() {

this.state = new State(...);

this.cleanable = cleaner.register(this, state);

}

public void close() {

cleanable.clean();

}

}

注意，从可预测性的角度来判断，Cleaner 或者幻象引用改善的程度仍然是有限的，如果由于种种原因导致幻象引用堆积，同样会出现问题。所以，Cleaner 适合作为一种最后的保证手段，而不是完全依赖 Cleaner 进行资源回收，不然我们就要再做一遍 finalize 的噩梦了。

我也注意到很多第三方库自己直接利用幻象引用定制资源收集，比如广泛使用的 MySQL JDBC driver 之一的 mysql-connector-j，就利用了幻象引用机制。幻象引用也可以进行类似链条式依赖关系的动作，比如，进行总量控制的场景，保证只有连接被关闭，相应资源被回收，连接池才能创建新的连接。

另外，这种代码如果稍有不慎添加了对资源的强引用关系，就会导致循环引用关系，前面提到的 MySQL JDBC 就在特定模式下有这种问题，导致内存泄漏。上面的示例代码中，将 State 定义为 static，就是为了避免普通的内部类隐含着对外部对象的强引用，因为那样会使外部对象无法进入幻象可达的状态。

今天，我从语法角度分析了 final、finally、finalize，并从安全、性能、垃圾收集等方面逐步深入，探讨了实践中的注意事项，希望对你有所帮助。

关于今天我们讨论的题目你做到心中有数了吗？也许你已经注意到了，JDK 自身使用的 Cleaner 机制仍然是有缺陷的，你有什么更好的建议吗？

#### 1.3.1.4强引用、软引用、弱引用、幻象引用有什么区别？

##### 1.3.1.4.1典型回答

在 Java 语言中，除了原始数据类型的变量，其他所有都是所谓的引用类型，指向各种不同的对象，理解引用对于掌握 Java 对象生命周期和 JVM 内部相关机制非常有帮助。

今天我要问你的问题是，强引用、软引用、弱引用、幻象引用有什么区别？具体使用场景是什么？

不同的引用类型，主要体现的是**对象不同的可达性（reachable）状态和对垃圾收集的影响**。

所谓强引用（"Strong" Reference），就是我们最常见的普通对象引用，只要还有强引用指向一个对象，就能表明对象还“活着”，垃圾收集器不会碰这种对象。对于一个普通的对象，如果没有其他的引用关系，只要超过了引用的作用域或者显式地将相应（强）引用赋值为 null，就是可以被垃圾收集的了，当然具体回收时机还是要看垃圾收集策略。

软引用（SoftReference），是一种相对强引用弱化一些的引用，可以让对象豁免一些垃圾收集，只有当 JVM 认为内存不足时，才会去试图回收软引用指向的对象。JVM 会确保在抛出 OutOfMemoryError 之前，清理软引用指向的对象。软引用通常用来实现内存敏感的缓存，如果还有空闲内存，就可以暂时保留缓存，当内存不足时清理掉，这样就保证了使用缓存的同时，不会耗尽内存。

弱引用（WeakReference）并不能使对象豁免垃圾收集，仅仅是提供一种访问在弱引用状态下对象的途径。这就可以用来构建一种没有特定约束的关系，比如，维护一种非强制性的映射关系，如果试图获取时对象还在，就使用它，否则重现实例化。它同样是很多缓存实现的选择。

对于幻象引用，有时候也翻译成虚引用，你不能通过它访问对象。幻象引用仅仅是提供了一种确保对象被 finalize 以后，做某些事情的机制，比如，通常用来做所谓的 Post-Mortem 清理机制，我在专栏上一讲中介绍的 Java 平台自身 Cleaner 机制等，也有人利用幻象引用监控对象的创建和销毁。

##### 1.3.1.4.2**考点分析**

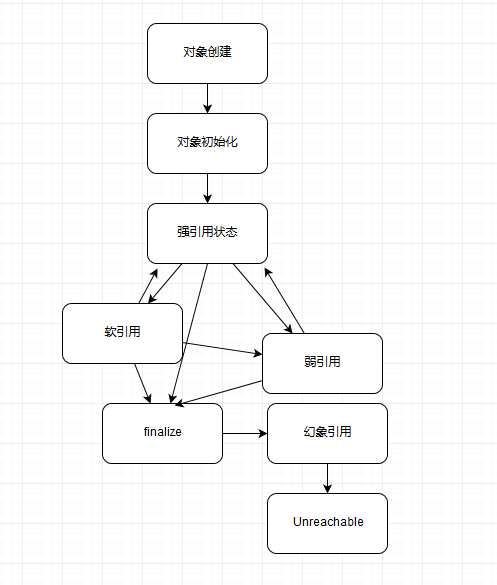
这道面试题，属于既偏门又非常高频的一道题目。说它偏门，是因为在大多数应用开发中，很少直接操作各种不同引用，虽然我们使用的类库、框架可能利用了其机制。它被频繁问到，是因为这是一个综合性的题目，既考察了我们对基础概念的理解，也考察了对底层对象生命周期、垃圾收集机制等的掌握。

充分理解这些引用，对于我们设计可靠的缓存等框架，或者诊断应用 OOM 等问题，会很有帮助。比如，诊断 MySQL connector-j 驱动在特定模式下（useCompression=true）的内存泄漏问题，就需要我们理解怎么排查幻象引用的堆积问题。

##### 1.3.1.4.3**知识扩展**

1. 对象可达性状态流转分析

首先，请你看下面流程图，我这里简单总结了对象生命周期和不同可达性状态，以及不同状态可能的改变关系，可能未必 100% 严谨，来阐述下可达性的变化。



我来解释一下上图的具体状态，这是 Java 定义的不同可达性级别（reachability level），具体如下：

强可达（Strongly Reachable），就是当一个对象可以有一个或多个线程可以不通过各种引用访问到的情况。比如，我们新创建一个对象，那么创建它的线程对它就是强可达。

软可达（Softly Reachable），就是当我们只能通过软引用才能访问到对象的状态。

弱可达（Weakly Reachable），类似前面提到的，就是无法通过强引用或者软引用访问，只能通过弱引用访问时的状态。这是十分临近 finalize 状态的时机，当弱引用被清除的时候，就符合 finalize 的条件了。

幻象可达（Phantom Reachable），上面流程图已经很直观了，就是没有强、软、弱引用关联，并且 finalize 过了，只有幻象引用指向这个对象的时候。

当然，还有一个最后的状态，就是不可达（unreachable），意味着对象可以被清除了。

判断对象可达性，是 JVM 垃圾收集器决定如何处理对象的一部分考虑。

所有引用类型，都是抽象类 java.lang.ref.Reference 的子类，你可能注意到它提供了 get() 方法：

IMG_258

除了幻象引用（因为 get 永远返回 null），如果对象还没有被销毁，都可以通过 get 方法获取原有对象。这意味着，利用软引用和弱引用，我们可以将访问到的对象，重新指向强引用，也就是人为的改变了对象的可达性状态！这也是为什么我在上面图里有些地方画了双向箭头。

所以，对于软引用、弱引用之类，垃圾收集器可能会存在二次确认的问题，以保证处于弱引用状态的对象，没有改变为强引用。

但是，你觉得这里有没有可能出现什么问题呢？

不错，如果我们错误的保持了强引用（比如，赋值给了 static 变量），那么对象可能就没有机会变回类似弱引用的可达性状态了，就会产生内存泄漏。所以，检查弱引用指向对象是否被垃圾收集，也是诊断是否有特定内存泄漏的一个思路，如果我们的框架使用到弱引用又怀疑有内存泄漏，就可以从这个角度检查。

2. 引用队列（ReferenceQueue）使用

谈到各种引用的编程，就必然要提到引用队列。我们在创建各种引用并关联到响应对象时，可以选择是否需要关联引用队列，JVM 会在特定时机将引用 enqueue 到队列里，我们可以从队列里获取引用（remove 方法在这里实际是有获取的意思）进行相关后续逻辑。尤其是幻象引用，get 方法只返回 null，如果再不指定引用队列，基本就没有意义了。看看下面的示例代码。利用引用队列，我们可以在对象处于相应状态时（对于幻象引用，就是前面说的被 finalize 了，处于幻象可达状态），执行后期处理逻辑。

Object counter = new Object();

ReferenceQueue refQueue = new ReferenceQueue<>();

PhantomReference<Object> p = new PhantomReference<>(counter, refQueue);

counter = null;

System.gc();

try {

// Remove 是一个阻塞方法，可以指定 timeout，或者选择一直阻塞

Reference<Object> ref = refQueue.remove(1000L);

if (ref != null) {

// do something

}

} catch (InterruptedException e) {

// Handle it

}

3. 显式地影响软引用垃圾收集

前面泛泛提到了引用对垃圾收集的影响，尤其是软引用，到底 JVM 内部是怎么处理它的，其实并不是非常明确。那么我们能不能使用什么方法来影响软引用的垃圾收集呢？

答案是有的。软引用通常会在最后一次引用后，还能保持一段时间，默认值是根据堆剩余空间计算的（以 M bytes 为单位）。从 Java 1.3.1 开始，提供了 -XX:SoftRefLRUPolicyMSPerMB 参数，我们可以以毫秒（milliseconds）为单位设置。比如，下面这个示例就是设置为 3 秒（3000 毫秒）。

-XX:SoftRefLRUPolicyMSPerMB=3000

这个剩余空间，其实会受不同 JVM 模式影响，对于 Client 模式，比如通常的 Windows 32 bit JDK，剩余空间是计算当前堆里空闲的大小，所以更加倾向于回收；而对于 server 模式 JVM，则是根据 -Xmx 指定的最大值来计算。

本质上，这个行为还是个黑盒，取决于 JVM 实现，即使是上面提到的参数，在新版的 JDK 上也未必有效，另外 Client 模式的 JDK 已经逐步退出历史舞台。所以在我们应用时，可以参考类似设置，但不要过于依赖它。

4. 诊断 JVM 引用情况

如果你怀疑应用存在引用（或 finalize）导致的回收问题，可以有很多工具或者选项可供选择，比如 HotSpot JVM 自身便提供了明确的选项 （PrintReferenceGC）去获取相关信息，我指定了下面选项去使用 JDK 8 运行一个样例应用：

-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintReferenceGC

这是 JDK 8 使用 ParrallelGC 收集的垃圾收集日志，各种引用数量非常清晰。

0.403: [GC (Allocation Failure) 0.871: [SoftReference, 0 refs, 0.0000393 secs]0.871: [WeakReference, 8 refs, 0.0000138 secs]0.871: [FinalReference, 4 refs, 0.0000094 secs]0.871: [PhantomReference, 0 refs, 0 refs, 0.0000085 secs]0.871: [JNI Weak Reference, 0.0000071 secs][PSYoungGen: 76272K->10720K(141824K)] 128286K->128422K(316928K), 0.4683919 secs] [Times: user=1.17 sys=0.03, real=0.47 secs]

**注意：JDK 9 对 JVM 和垃圾收集日志进行了广泛的重构**，类似 PrintGCTimeStamps 和 PrintReferenceGC 已经不再存在，我在专栏后面的垃圾收集主题里会更加系统的阐述。

5.Reachability Fence

除了我前面介绍的几种基本引用类型，我们也可以通过底层 API 来达到强引用的效果，这就是所谓的设置**reachability fence**。

为什么需要这种机制呢？考虑一下这样的场景，按照 Java 语言规范，如果一个对象没有指向强引用，就符合垃圾收集的标准，有些时候，对象本身并没有强引用，但是也许它的部分属性还在被使用，这样就导致诡异的问题，所以我们需要一个方法，在没有强引用情况下，通知 JVM 对象是在被使用的。说起来有点绕，我们来看看 Java 9 中提供的案例。

class Resource {

private static ExternalResource[] externalResourceArray = ...

int myIndex; Resource(...) {

myIndex = ...

externalResourceArray[myIndex] = ...;

...

}

protected void finalize() {

externalResourceArray[myIndex] = null;

...

}

public void action() {

try {

// 需要被保护的代码

int i = myIndex;

Resource.update(externalResourceArray[i]);

} finally {

// 调用 reachbilityFence，明确保障对象 strongly reachable

Reference.reachabilityFence(this);

}

}

private static void update(ExternalResource ext) {

ext.status = ...;

}

}

方法 action 的执行，依赖于对象的部分属性，所以被特定保护了起来。否则，如果我们在代码中像下面这样调用，那么就可能会出现困扰，因为没有强引用指向我们创建出来的 Resource 对象，JVM 对它进行 finalize 操作是完全合法的。

new Resource().action()

类似的书写结构，在异步编程中似乎是很普遍的，因为异步编程中往往不会用传统的“执行 -> 返回 -> 使用”的结构。

在 Java 9 之前，实现类似类似功能相对比较繁琐，有的时候需要采取一些比较隐晦的小技巧。幸好，java.lang.ref.Reference 给我们提供了新方法，它是 JEP 193: Variable Handles 的一部分，将 Java 平台底层的一些能力暴露出来：

static void reachabilityFence(Object ref)

在 JDK 源码中，reachabilityFence 大多使用在 Executors 或者类似新的 HTTP/2 客户端代码中，大部分都是异步调用的情况。编程中，可以按照上面这个例子，将需要 reachability 保障的代码段利用 try-finally 包围起来，在 finally 里明确声明对象强可达。

今天，我总结了 Java 语言提供的几种引用类型、相应可达状态以及对于 JVM 工作的意义，并分析了引用队列使用的一些实际情况，最后介绍了在新的编程模式下，如何利用 API 去保障对象不被以为意外回收，希望对你有所帮助。

关于今天我们讨论的题目你做到心中有数了吗？给你留一道练习题，你能从自己的产品或者第三方类库中找到使用各种引用的案例吗？它们都试图解决什么问题？

#### 1.3.1.5String、StringBuffer、StringBuilder有什么区别？

##### 1.3.1.5.1典型回答

今天我会聊聊日常使用的字符串，别看它似乎很简单，但其实字符串几乎在所有编程语言里都是个特殊的存在，因为不管是数量还是体积，字符串都是大多数应用中的重要组成。

今天我要问你的问题是，理解 Java 的字符串，String、StringBuffer、StringBuilder 有什么区别？

String 是 Java 语言非常基础和重要的类，提供了构造和管理字符串的各种基本逻辑。它是典型的 Immutable 类，被声明成为 final class，所有属性也都是 final 的。也由于它的不可变性，类似拼接、裁剪字符串等动作，都会产生新的 String 对象。由于字符串操作的普遍性，所以相关操作的效率往往对应用性能有明显影响。

StringBuffer 是为解决上面提到拼接产生太多中间对象的问题而提供的一个类，它是 Java 1.5 中新增的，我们可以用 append 或者 add 方法，把字符串添加到已有序列的末尾或者指定位置。StringBuffer 本质是一个线程安全的可修改字符序列，它保证了线程安全，也随之带来了额外的性能开销，所以除非有线程安全的需要，不然还是推荐使用它的后继者，也就是 StringBuilder。

StringBuilder 在能力上和 StringBuffer 没有本质区别，但是它去掉了线程安全的部分，有效减小了开销，是绝大部分情况下进行字符串拼接的首选。

##### 1.3.1.5.2**考点分析**

几乎所有的应用开发都离不开操作字符串，理解字符串的设计和实现以及相关工具如拼接类的使用，对写出高质量代码是非常有帮助的。关于这个问题，我前面的回答是一个通常的概要性回答，至少你要知道 String 是 Immutable 的，字符串操作不当可能会产生大量临时字符串，以及线程安全方面的区别。

如果继续深入，面试官可以从各种不同的角度考察，比如可以：

通过 String 和相关类，考察基本的线程安全设计与实现，各种基础编程实践。

考察 JVM 对象缓存机制的理解以及如何良好地使用。

考察 JVM 优化 Java 代码的一些技巧。

String 相关类的演进，比如 Java 9 中实现的巨大变化。

针对上面这几方面，我会在知识扩展部分与你详细聊聊。

##### 1.3.1.5.3**知识扩展**

1. 字符串设计和实现考量

我在前面介绍过，String 是 Immutable 类的典型实现，原生的保证了基础线程安全，因为你无法对它内部数据进行任何修改，这种便利甚至体现在拷贝构造函数中，由于不可变，Immutable 对象在拷贝时不需要额外复制数据。

我们再来看看 StringBuffer 实现的一些细节，它的线程安全是通过把各种修改数据的方法都加上 synchronized 关键字实现的，非常直白。其实，这种简单粗暴的实现方式，非常适合我们常见的线程安全类实现，不必纠结于 synchronized 性能之类的，有人说“过早优化是万恶之源”，考虑可靠性、正确性和代码可读性才是大多数应用开发最重要的因素。

为了实现修改字符序列的目的，StringBuffer 和 StringBuilder 底层都是利用可修改的（char，JDK 9 以后是 byte）数组，二者都继承了 AbstractStringBuilder，里面包含了基本操作，区别仅在于最终的方法是否加了 synchronized。

另外，这个内部数组应该创建成多大的呢？如果太小，拼接的时候可能要重新创建足够大的数组；如果太大，又会浪费空间。目前的实现是，构建时初始字符串长度加 16（这意味着，如果没有构建对象时输入最初的字符串，那么初始值就是 16）。我们如果确定拼接会发生非常多次，而且大概是可预计的，那么就可以指定合适的大小，避免很多次扩容的开销。扩容会产生多重开销，因为要抛弃原有数组，创建新的（可以简单认为是倍数）数组，还要进行 arraycopy。

前面我讲的这些内容，在具体的代码书写中，应该如何选择呢？

在没有线程安全问题的情况下，全部拼接操作是应该都用 StringBuider 实现吗？毕竟这样书写的代码，还是要多敲很多字的，可读性也不理想，下面的对比非常明显。

String strByBuilder = new

StringBuilder().append("aa").append("bb").append("cc").append

("dd").toString();

String strByConcat = "aa" + "bb" + "cc" + "dd";

其实，在通常情况下，没有必要过于担心，要相信 Java 还是非常智能的。

我们来做个实验，把下面一段代码，利用不同版本的 JDK 编译，然后再反编译，例如：

public class StringConcat {

public static void main(String[] args) {

String myStr = "aa" + "bb" + "cc" + "dd";

System.out.println("My String:" + myStr);

}

}

先编译再反编译，比如使用 JDK 9：

${JAVA9\_HOME}/bin/javac StringConcat.java

${JAVA9\_HOME}/bin/javap -v StringConcat.class

JDK 8 的输出片段是：

6: new #4 // class java/lang/StringBuilder

9: dup

10: invokespecial #5 // Method java/lang/StringBuilder."<init>":()V

13: ldc #6 // String My String:

15: invokevirtual #7 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;

18: aload\_1

19: invokevirtual #7 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;

22: invokevirtual #8 // Method java/lang/StringBuilder.toString:()Ljava/lang/String;

而在 JDK 9 中，反编译的结果就非常简单了，片段是：

7: invokedynamic #4, 0 // InvokeDynamic #0:makeConcatWithConstants:(Ljava/lang/String;)Ljava/lang/String;

你可以看到，在 JDK 8 中，字符串拼接操作会自动被 javac 转换为 StringBuilder 操作，而在 JDK 9 里面则是因为 Java 9 为了更加统一字符串操作优化，提供了 StringConcatFactory，作为一个统一的入口。javac 自动生成的代码，虽然未必是最优化的，但普通场景也足够了，你可以酌情选择。

2. 字符串缓存

我们粗略统计过，把常见应用进行堆转储（Dump Heap），然后分析对象组成，会发现平均 25% 的对象是字符串，并且其中约半数是重复的。如果能避免创建重复字符串，可以有效降低内存消耗和对象创建开销。

String 在 Java 6 以后提供了 intern() 方法，目的是提示 JVM 把相应字符串缓存起来，以备重复使用。在我们创建字符串对象并调用 intern() 方法的时候，如果已经有缓存的字符串，就会返回缓存里的实例，否则将其缓存起来。一般来说，JVM 会将所有的类似“abc”这样的文本字符串，或者字符串常量之类缓存起来。

看起来很不错是吧？但实际情况估计会让你大跌眼镜。一般使用 Java 6 这种历史版本，并不推荐大量使用 intern，为什么呢？魔鬼存在于细节中，被缓存的字符串是存在所谓 PermGen 里的，也就是臭名昭著的“永久代”，这个空间是很有限的，也基本不会被 FullGC 之外的垃圾收集照顾到。所以，如果使用不当，OOM 就会光顾。

在后续版本中，这个缓存被放置在堆中，这样就极大避免了永久代占满的问题，甚至永久代在 JDK 8 中被 MetaSpace（元数据区）替代了。而且，默认缓存大小也在不断地扩大中，从最初的 1009，到 7u40 以后被修改为 60013。你可以使用下面的参数直接打印具体数字，可以拿自己的 JDK 立刻试验一下。

-XX:+PrintStringTableStatistics

你也可以使用下面的 JVM 参数手动调整大小，但是绝大部分情况下并不需要调整，除非你确定它的大小已经影响了操作效率。

-XX:StringTableSize=N

Intern 是一种**显式地排重机制**，但是它也有一定的副作用，因为需要开发者写代码时明确调用，一是不方便，每一个都显式调用是非常麻烦的；另外就是我们很难保证效率，应用开发阶段很难清楚地预计字符串的重复情况，有人认为这是一种污染代码的实践。

幸好在 Oracle JDK 8u20 之后，推出了一个新的特性，也就是 G1 GC 下的字符串排重。它是通过将相同数据的字符串指向同一份数据来做到的，是 JVM 底层的改变，并不需要 Java 类库做什么修改。

注意这个功能目前是默认关闭的，你需要使用下面参数开启，并且记得指定使用 G1 GC：

-XX:+UseStringDeduplication

前面说到的几个方面，只是 Java 底层对字符串各种优化的一角，在运行时，字符串的一些基础操作会直接利用 JVM 内部的 Intrinsic 机制，往往运行的就是特殊优化的本地代码，而根本就不是 Java 代码生成的字节码。Intrinsic 可以简单理解为，是一种利用 native 方式 hard-coded 的逻辑，算是一种特别的内联，很多优化还是需要直接使用特定的 CPU 指令，具体可以看相关[源码](http://hg.openjdk.java.net/jdk/jdk/file/44b64fc0baa3/src/hotspot/share/classfile/vmSymbols.hpp)，搜索“string”以查找相关 Intrinsic 定义。当然，你也可以在启动实验应用时，使用下面参数，了解 intrinsic 发生的状态。

-XX:+PrintCompilation -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining

// 样例输出片段

180 3 3 java.lang.String::charAt (25 bytes)

@ 1 java.lang.String::isLatin1 (19 bytes)

...

@ 7 java.lang.StringUTF16::getChar (60 bytes) intrinsic

可以看出，仅仅是字符串一个实现，就需要 Java 平台工程师和科学家付出如此大且默默无闻的努力，我们得到的很多便利都是来源于此。

我会在专栏后面的 JVM 和性能等主题，详细介绍 JVM 内部优化的一些方法，如果你有兴趣可以再深入学习。即使你不做 JVM 开发或者暂时还没有使用到特别的性能优化，这些知识也能帮助你增加技术深度。

3.String 自身的演化

如果你仔细观察过 Java 的字符串，在历史版本中，它是使用 char 数组来存数据的，这样非常直接。但是 Java 中的 char 是两个 bytes 大小，拉丁语系语言的字符，根本就不需要太宽的 char，这样无区别的实现就造成了一定的浪费。密度是编程语言平台永恒的话题，因为归根结底绝大部分任务是要来操作数据的。

其实在 Java 6 的时候，Oracle JDK 就提供了压缩字符串的特性，但是这个特性的实现并不是开源的，而且在实践中也暴露出了一些问题，所以在最新的 JDK 版本中已经将它移除了。

在 Java 9 中，我们引入了 Compact Strings 的设计，对字符串进行了大刀阔斧的改进。将数据存储方式从 char 数组，改变为一个 byte 数组加上一个标识编码的所谓 coder，并且将相关字符串操作类都进行了修改。另外，所有相关的 Intrinsic 之类也都进行了重写，以保证没有任何性能损失。

虽然底层实现发生了这么大的改变，但是 Java 字符串的行为并没有任何大的变化，所以这个特性对于绝大部分应用来说是透明的，绝大部分情况不需要修改已有代码。

当然，在极端情况下，字符串也出现了一些能力退化，比如最大字符串的大小。你可以思考下，原来 char 数组的实现，字符串的最大长度就是数组本身的长度限制，但是替换成 byte 数组，同样数组长度下，存储能力是退化了一倍的！还好这是存在于理论中的极限，还没有发现现实应用受此影响。

在通用的性能测试和产品实验中，我们能非常明显地看到紧凑字符串带来的优势，**即更小的内存占用、更快的操作速度**。

今天我从 String、StringBuffer 和 StringBuilder 的主要设计和实现特点开始，分析了字符串缓存的 intern 机制、非代码侵入性的虚拟机层面排重、Java 9 中紧凑字符的改进，并且初步接触了 JVM 的底层优化机制 intrinsic。从实践的角度，不管是 Compact Strings 还是底层 intrinsic 优化，都说明了使用 Java 基础类库的优势，它们往往能够得到最大程度、最高质量的优化，而且只要升级 JDK 版本，就能零成本地享受这些益处。

关于今天我们讨论的题目你做到心中有数了吗？限于篇幅有限，还有很多字符相关的问题没有来得及讨论，比如编码相关的问题。可以思考一下，很多字符串操作，比如 getBytes()/[String](https://docs.oracle.com/javase/9/docs/api/java/lang/String.html" \l "String-byte:A-)(byte[] bytes) 等都是隐含着使用平台默认编码，这是一种好的实践吗？是否有利于避免乱码？

来源： <https://time.geekbang.org/column/article/7349>

# 6数据库

（1）只要执行了增添的sql语句，无论是否执行提交，数据都会存到数据库中，只不过是可视化软件隔离级别高，看不到而已。

测试代码如下：

Pom引入：

<dependencies>

<dependency>

<groupId>mysql</groupId>

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

<version>5.1.30</version>

</dependency>

<dependency>

<groupId>com.mchange</groupId>

<artifactId>c3p0</artifactId>

<version>0.9.5.2</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

</dependencies>

测试代码：

import com.mchange.v2.c3p0.ComboPooledDataSource;

import org.junit.Before;

import org.junit.Test;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

/\*\*

\* Created by jackiechan on 18-7-10/下午4:00.

\*/

public class TsetMain {

private ComboPooledDataSource comboPooledDataSource;

@Before

public void before() throws Exception {

comboPooledDataSource = new ComboPooledDataSource();

comboPooledDataSource.setUser("root");

comboPooledDataSource.setDriverClass("com.mysql.jdbc.Driver");

comboPooledDataSource.setPassword("qishimeiyoumima");

comboPooledDataSource.setJdbcUrl("jdbc:mysql:///solr");

}

public void testInsert(String title1,String content1) throws SQLException {

Connection connection = comboPooledDataSource.getConnection();

connection.setAutoCommit(false);

PreparedStatement preparedStatement = connection.prepareStatement("INSERT INTO solr (title,content) VALUES (?,?)");

//statement.execute("INSERT INTO solr (title,content) VALUES ("+title1+","+content1+")");

preparedStatement.setString(1, title1);

preparedStatement.setString(2, content1);

preparedStatement.execute();

connection.commit();

connection.close();

}

@Test

public void test() throws SQLException {

String title="title1";

String content = "content1";

testInsert(title,content);

}

@Test

public void test2() throws SQLException {

String title="title2";

String content = "content2";

testInsert(title,content);

}

测试流程，在test1和2调用testInsert方法的地方打断点，然后同时执行两个test，然后先放行test2，发现数据库增加了第58条语句，然后放行test1，发现数据库增加了第57条语句，这就说明实质上test1的数据在test2之前就已经存到了数据库。综上（1）只要执行了增添的sql语句，无论是否执行提交，数据都会存到数据库中，只不过是可视化软件隔离级别高，看不到而已。

# 第四章 工具类与配置文件集锦

# 1.数据库相关

## 1.1C3P0连接池

（1）DataSourceUtil

|  |
| --- |
| package com.qf.springjdbc.utils;  import com.mchange.v2.c3p0.ComboPooledDataSource;  import javax.sql.DataSource;  import java.sql.Connection;  import java.sql.SQLException;  /\*\*  \* idea C3P0连接池  \*/  public class DataSourceUtil {  private static ThreadLocal<Connection> t1=new ThreadLocal<Connection>();  //C3P0连接池对象  private static DataSource dataSource=new ComboPooledDataSource();  public static DataSource getDataSource(){  return dataSource;  }  //从ThreadLocal中获取Connection对象，如果不存在则把取到的Connection对象放到ThreadLocal  public static Connection getConnection(){  //从绑定线程获得连接  Connection con=t1.get();  if(con==null){  //从连接池获得连接  try {  con=dataSource.getConnection();  } catch (SQLException e) {  e.printStackTrace();  }  t1.set(con);  }  return con;  }  //事务开启  public static void startTransaction(){  Connection con=getConnection();  if(con!=null){  try {  con.setAutoCommit(false);  } catch (SQLException e) {  e.printStackTrace();  }  }  }  //事务回滚  public static void rollback(){  Connection con=getConnection();  if(con!=null){  try {  con.rollback();  } catch (SQLException e) {  e.printStackTrace();  }  }  }  //事务提交  public static void commit(){  Connection con=getConnection();  if(con!=null){  try {  con.commit();  } catch (SQLException e) {  e.printStackTrace();  }  }  }  //关闭资源及从ThreadLocal中释放  public static void release(){  Connection con=getConnection();  if(con!=null){  try {  con.close();//放回连接池  } catch (SQLException e) {  e.printStackTrace();  }  t1.remove();//从线程绑定中移除  }  }  } |

（2）C3P0配置xml文件(resources文件夹下)

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <c3p0-config>  <default-config>  <property name="driverClass">com.mysql.jdbc.Driver</property>  <property name="jdbcUrl">jdbc:mysql://localhost:3306/qf</property>  <property name="user">root</property>  <property name="password">123</property>  </default-config>  </c3p0-config> |

## 1.2db.properties

|  |
| --- |
| jdbc.jdbcUrl=jdbc:mysql://localhost:3306/qf  jdbc.driverClass=com.mysql.jdbc.Driver  jdbc.user=root  jdbc.password=123 |

# Spring

## 2.1spring配置文件（applicationContext.xml）

（1）spring配置文件：applicationContext.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/aop [http://www.springframework.org/schema/aop/spring-aop.xsd">](http://www.springframework.org/schema/aop/spring-aop.xsd\">)  <!--设置自动扫描范围-->  <context:component-scan base-package="com.qf.springjdbc"/>  <!--&lt;!&ndash;也可以按以下方式设置&ndash;&gt;-->  <!--<context:component-scan base-package="com.qf.springjdbc.dao"/>-->  <!--<context:component-scan base-package="com.qf.springjdbc.service"/>-->  <!--指定数据源和配置文件-->  <context:property-placeholder location="db.properties"/>  <!--数据库连接池,class必须是全称-->  <!--&lt;!&ndash;配置方式1，不使用db.properties文件&ndash;&gt;-->  <!--<bean name="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">-->  <!--&lt;!&ndash; name要和函数中的set后面的属性一致：dataSource.setDriverClass("com.mysql.jdbc.Driver");&ndash;&gt;-->  <!--<property name="driverClass" value="com.mysql.jdbc.Driver"></property>-->  <!--<property name="jdbcUrl" value="jdbc:mysql://localhost:3306/qf"></property>-->  <!--<property name="user" value="root"></property>-->  <!--<property name="password" value="123"></property>-->  <!--</bean>-->  <!--配置方式2，使用properties文件，与方式一区别在于values获取方式-->  <bean name="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="driverClass" value="${jdbc.driverClass}"></property>  <property name="jdbcUrl" value="${jdbc.jdbcUrl}"></property>  <property name="user" value="${jdbc.user}"></property>  <property name="password" value="${jdbc.password}"></property>  </bean>  <!--注入区,构造函数用constructor-arg注入，set函数用property注入-->  <!--jdbcTemplate需要注入datasource-->  <bean name="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  <!-- name要和函数中的set后面的属性一致,ref关联的是上边连接池的id，即name-->  <property name="dataSource" ref="dataSource"></property>  </bean>  <!--daoimpl需要注入jdbcTemplate-->  <!--要注入到实现层，不要注入到接口层，接口层不能创建对象，没用-->  <bean name="userDao" class="com.qf.springjdbc.dao.impl.UserDaoImpl">  <property name="jdbcTemplate" ref="jdbcTemplate"></property>  </bean>  <!--事务管理方式一：XML管理-->  <!--1事务管理器(事务需要注入数据源，另外这是第三方程序，只能用javaconfig或者xml配置，不能使用注解)-->  <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource"/>  </bean>  <!--2通知advice引入的是带有tx的那个-->  <tx:advice id="txAdvice" transaction-manager="transactionManager">  <tx:attributes>  <!--name表示你要执行的方法，save\*包括saveUser等等所有save开头的方法-->  <!--propagation表示传播方式，详情见第三部葵花宝典5.7.3-->  <tx:method name="save\*" propagation="REQUIRED"/>  <tx:method name="update\*" propagation="REQUIRED"/>  <tx:method name="delete\*" propagation="REQUIRED"/>  <!--read-on;y设置为true会增加查询效率-->  <tx:method name="get\*" propagation="SUPPORTS" read-only="true"/>  </tx:attributes>  </tx:advice>  <!--3切面:在哪用事物-->  <aop:config>  <aop:pointcut id="txPointcut" expression="execution(\* com.qf.springjdbc.service..\*.\*(..))"/>  <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>  </aop:config>  <!--事务管理方式二：使用注解管理-->  <!--1事务管理器(事务需要注入数据源，另外这是第三方程序，只能用javaconfig或者xml配置，不能使用注解)-->  <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource"/>  </bean>  <!--2开启使用注解管理aop事务-->  <tx:annotation-driven/>  <!--&lt;!&ndash;aop设置&ndash;&gt;-->  <!--<context:component-scan base-package="com.qf.springaop"/>-->  <!--<aop:aspectj-autoproxy/>-->  <!--<aop:config>-->  <!--<aop:aspect ref="sysLog">-->  <!--<aop:before method="before" pointcut="execution(\* com.qf.springaop.service..\*ServiceImpl.\*(..))" />-->  <!--<aop:after-returning method="afterReturning" pointcut="execution(\* com.qf.springaop.service..\*ServiceImpl.\*(..))" />-->  <!--<aop:after method="after" pointcut="execution(\* com.qf.springaop.service..\*ServiceImpl.\*(..))"/>-->  <!--<aop:after-throwing method="afterThrowing" pointcut="execution(\* com.qf.springaop.service..\*ServiceImpl.\*(..))"/>-->  <!--<aop:around method="around" pointcut="execution(\* com.qf.springaop.service..\*ServiceImpl.\*(..))" />-->  <!--</aop:aspect>-->  <!--</aop:config>-->  </beans> |

## 2.2 springmvc配置文件

### 2.2.1springmvc-servlet.xml

ssm整合中放在resources/spring文件夹下

P：空间的前缀后缀的斜杠是个关键，要和Controller中的return路径配合好

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context" xmlns:p="http://www.springframework.org/schema/p"  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">  <!--配置包扫描-->  <context:component-scan base-package="com.qf.springmvc.controller"/>  <!--视图解析器-->  <!--InternalResourceViewResolver是针对于JSP的解析器，视图解析器作用在于实现controller中return路径到网页的跳转-->  <!--p:\*\*是定义controller中return路径前缀和后缀，定义完成后，controller中return路径 return "/Web-INF/jsp/index.jsp";可以改为return "index";-->  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver" p:prefix="/WEB-INF/jsp/" p:suffix=".jsp"/>  <!-- 配置注解驱动 -->  <mvc:annotation-driven />    <!-- 处理静态资源 -->  <!-- 所有以/css路径开头的请求经过springMVC的DispatcherServlet不进入controller找@RequestMapping，而是进入DispatcherServlet的ResourceHttpRequestHandler 进行处理-->  <mvc:resources mapping="/css/\*\*" location="/css/"/>  <mvc:resources mapping="/image/\*\*" location="/image/"/>  <mvc:resources mapping="/js/\*\*" location="/js/"/>  <!--<!--default-servlet-handler设置是自动检测到静态资源类型，如果是css，就转到css对应的容器处理，如果是jsp，就转到jsp对应容器处理，不用单独分开罗列所有resources设置，但是缺点是效率低。-->  <mvc:default-servlet-handler/> -->  <!--配置验证码生成器-->  <bean id="producer" class="com.google.code.kaptcha.impl.DefaultKaptcha" scope="singleton">  <property name="config">  <bean class="com.google.code.kaptcha.util.Config">  <constructor-arg>  <props>  <prop key="kaptcha.border">no</prop>  <prop key="kaptcha.textproducer.font.color">black</prop>  <prop key="kaptcha.textproducer.char.length">5</prop>  </props>  </constructor-arg>  </bean>  </property>  </bean>  <!--配置拦截器-->  <mvc:interceptors>  <!--多个拦截器,顺序执行 -->  <mvc:interceptor>  <!-- /\*\*可以拦截路径不管多少层 -->  <mvc:mapping path="/\*\*"/>  <!--设置不拦截css目录下的所有文件-->  <mvc:exclude-mapping path="/css/\*\*"/>  <bean class="api.interceptor.LoginInterceptor"></bean>  </mvc:interceptor>  </mvc:interceptors>  <!--配置 MultipartResolver 处理文件上传-->  <bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">  <property name="defaultEncoding" value="UTF-8"/> <!-- 默认编码 (ISO-8859-1) -->  <property name="maxInMemorySize" value="10240"/> <!-- 最大内存大小 (10240)-->  <property name="uploadTempDir" value="/upload/"/> <!-- 上传后的目录名 -->  <property name="maxUploadSize" value="-1"/> <!-- 最大文件大小，单位是字节，-1为无限制 -->  </bean>  <!-- 保证实现了Shiro内部lifecycle函数的bean执行 -->  <bean id="lifecycleBeanPostProcessor" class="org.apache.shiro.spring.LifecycleBeanPostProcessor"/>  <!--&lt;!&ndash;用于启用shiro注解配置，这个必须要配置到springmvc-servlet.xml的中，要保证最后执行，否则会因为参数缺失而配置出错&ndash;&gt;-->  <!--&lt;!&ndash; 保证实现了Shiro内部lifecycle函数的bean执行 &ndash;&gt;-->  <!--<bean id="lifecycleBeanPostProcessor" class="org.apache.shiro.spring.LifecycleBeanPostProcessor"/>-->  <!--&lt;!&ndash; 开启Shiro的注解(如@RequiresRoles,@RequiresPermissions),需借助SpringAOP扫描使用Shiro注解的类,并在必要时进行安全逻辑验证 &ndash;&gt;-->  <!--&lt;!&ndash; 配置以下两个bean即可实现此功能 &ndash;&gt;-->  <!--&lt;!&ndash; Enable Shiro Annotations for Spring-configured beans. Only run after the lifecycleBeanProcessor has run &ndash;&gt;-->  <!--&lt;!&ndash; 由于本例中并未使用Shiro注解,故注释掉这两个bean(个人觉得将权限通过注解的方式硬编码在程序中,查看起来不是很方便,没必要使用) &ndash;&gt;-->  <!--&lt;!&ndash; &ndash;&gt;-->  <!--<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator" depends-on="lifecycleBeanPostProcessor"/>-->  <!--<bean class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor">-->  <!--<property name="securityManager" ref="securityManager"/>-->  <!--</bean>-->  </beans> |

### 2.2.2web.xml（web-int文件夹下）

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"  version="3.1">  <servlet>  <!--配置SpringMVC的 DispatcherServlet,相当于前置服务器baseServlet-->  <servlet-name>springmvc</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <!--配置 DispatcherServlet 的一个初始化参数: 配置 springmvc 配置文件的位置和名称 -->  <!--实际上也可以不通过 contextConfigLocation 来配置 springmvc 的配置文件, 而使用默认的.  默认读取的配置文件为: /WEB-INF/<servlet-name>（即上面设置的springmvc）-servlet.xml  如：springmvc-servlet.xml。所以只需将配置文件放到WEB-INF下就无需再写<init-param>配置了-->  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:springmvc-servlet.xml</param-value>  </init-param>  <!--设置一打开Tomcat就运行这个servlet-->  <load-on-startup>1</load-on-startup>  </servlet>    <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <!--/\*拦截的会走服务器并进行响应，不拦截的会直接放行直接走去页面显示\*/  /\* 拦截所有请求并响应  / 拦截所有请求并响应，不包括 jsp  \*.do 或 \*.action 只拦截 do 或 action 扩展名的请求并响应-->  <url-pattern>/</url-pattern>  </servlet-mapping>  <!--字符编码过滤器 : 放在其他过滤器的前面（解决post乱码问题）-->  <filter>  <filter-name>CharacterEncodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>utf-8</param-value>  </init-param>  <init-param>  <param-name>forceEncoding</param-name>  <param-value>true</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>CharacterEncodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping>  <!--ssm整合shiro中引入过滤器-->  <!-- 配置Shiro过滤器,先让Shiro过滤系统接收到的请求 -->  <!-- 这里filter-name必须对应applicationContext.xml或者 resource 文件夹下spring配置文件中配置过滤器相关的定义的<bean id="shiroFilter"/> -->  <!-- 使用[/\*]匹配所有请求,保证所有的可控请求都经过Shiro的过滤 -->  <!-- 通常会将此filter-mapping放置到最前面(即其他filter-mapping前面),以保证它是过滤器链中第一个起作用的 -->  <filter>  <filter-name>shiroFilter</filter-name> <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>  <init-param>  <param-name>targetFilterLifecycle</param-name>  <param-value>true</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>shiroFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping>  </web-app> |

### 2.2.3spring-shiro.xml

ssm整合中放在resources/spring文件夹下

|  |
| --- |
| <?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">  <!-- 缓存 -->  <bean id="cacheManager" class="org.apache.shiro.cache.ehcache.EhCacheManager">  <property name="cacheManagerConfigFile" value="classpath:ehcache.xml"/>  </bean>  <!-- 继承自AuthorizingRealm的自定义Realm,即指定Shiro验证用户登录的类为自定义的UserRealm.java -->  <bean id="userRealm" class="api.realm.UserRealm"></bean>  <!-- Shiro默认会使用Servlet容器的Session,可通过sessionMode属性来指定使用Shiro原生Session -->  <!-- 即<property name="sessionMode" value="native"/>,详细说明见官方文档 -->  <!-- 这里主要是设置自定义的单Realm应用,若有多个Realm,可使用'realms'属性代替 -->  <bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">  <property name="realm" ref="userRealm" />  <!--value是秒，cookie.maxAge红没事儿，是idea的问题-->  <property name="rememberMeManager.cookie.maxAge" value="60" />  <property name="cacheManager" ref="cacheManager"/>  </bean>  <!--这里的id要和web.xml中的filter的name要一致-->  <!-- Shiro主过滤器本身功能十分强大,其强大之处就在于它支持任何基于URL路径表达式的、自定义的过滤器的执行 -->  <!-- Web应用中,Shiro可控制的Web请求必须经过Shiro主过滤器的拦截,Shiro对基于Spring的Web应用提供了完美的支持 -->  <bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">  <!-- Shiro的核心安全接口,这个属性是必须的 -->  <property name="securityManager" ref="securityManager" />  <!--登录页面-->  <!-- 要求登录时的链接(可根据项目的URL进行替换),非必须的属性-->  <!--当认证失败时默认会自动寻找Web工程根目录下的"/login.html"页面-->  <property name="loginUrl" value="/login.html" />  <!--登录成功的首页面-->  <!-- 当认证成功后要跳转的连接 -->  <property name="successUrl" value="/" />  <!--授权失败的跳转链接-->  <!-- 用户访问未对其授权的资源时,所显示的连接 -->  <!-- 若想更明显的测试此属性可以修改它的值,如unauthorized.html -->  <property name="unauthorizedUrl" value="/unauthorized.html" />  <!-- Shiro连接约束配置,即过滤链的定义 -->  <!-- 此处可配合这篇文章来理解各个过滤连的作用http://blog.csdn.net/jadyer/article/details/12172839 -->  <!-- 下面value值的第一个'/'代表的路径是相对于HttpServletRequest.getContextPath()的值 -->  <!-- anon：spring预定义的空过滤器，定义等于这个过滤器就意味着等号前面对应的资源都将放行-->  <!-- authc：该过滤器下的页面必须验证后才能访问,它是Shiro内置的一个拦截器org.apache.shiro.web.filter.authc.FormAuthenticationFilter -->  <property name="filterChainDefinitions">  <value>  <!--/\*\*表示本文件夹及子文件全都过滤放行；/\*表示只放行本文件，但不放行子文件，不要和前置处理器的表示鬼，则混淆-->  /public/\*\*=anon  /login.html=anon  /api/sys/login=anon  /api/sys/captcha.jpg=anon  <!--/\*\*=authc-->  <!--角色授权-->  <!--当访问路径为/sys/user/list时，UserRealm会调用数据库user的角色信息，  只有当用户的角色是admin时才能够访问，否则直接跳转未授权页面。  roles是shiro的关键字-->  <!--/sys/user/list=roles[admin]-->  <!--/sys/\*\*=roles[admin]-->  <!--/file/\*\*=roles[editor]-->  <!--资源授权-->  <!--当访问路径为/sys/user/list时，UserRealm会调用数据库user的perms信息，  只有当用户的perms信息是sys:role:list时才能够访问，否则直接跳转未授权页面。  perms是shiro的关键字-->  <!--/sys/role/list=perms[sys:role:list]-->  <!--/sys/\*\*=perms[sys:\*]-->  <!--/schedule/\*\*=perms[schedule:\*]-->  <!--当有remenberme功能时，要使用user过滤器，同时关掉/\*\*=authc-->  /\*\*=user  </value>  </property>  </bean>  </beans> |

# 3.插件

（1）jdk插件：放在 <build><plugins> 节点下

|  |
| --- |
| <!-- java编译插件 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.6.0</version>  <configuration>  <source>1.8</source>  <target>1.8</target>  <encoding>${project.build.sourceEncoding}</encoding>  </configuration>  </plugin> |

（2）tomcat插件：放在 <build><plugins> 节点下

|  |
| --- |
| <!-- 配置Tomcat插件 -->  <plugin>  <groupId>org.apache.tomcat.maven</groupId>  <artifactId>tomcat7-maven-plugin</artifactId>  <version>2.2</version>  <configuration>  <port>8080</port>  <path>/test</path>  </configuration>  </plugin> |

1. 资源复制插件：放在<build>节点下

|  |
| --- |
| <!-- 拷贝资源文件 -->  <!-- 添加此节点将mybatis的XxxMapper.xml文件发布到target路径下 -->  <resources>  <resource>  <directory>src/main/java</directory>  <includes>  <!-- <include>\*\*/\*.properties</include> -->  <include>\*\*/\*.xml</include>  </includes>  <filtering>false</filtering>  </resource>  <!-- 上面的配置导致src/main/resources的所有文件都不能被扫描，因此还要配多一个 -->  <resource>  <directory>src/main/resources</directory>  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.xml</include>  </includes>  <filtering>false</filtering>  </resource>  </resources> |

（4）可以在每个项目中配置阿里云镜像：和<build>同一级别

|  |
| --- |
| <!-- 使用aliyun镜像 -->  <repositories>  <repository>  <id>aliyun</id>  <name>aliyun</name>  <url>http://maven.aliyun.com/nexus/content/groups/public</url>  </repository>  </repositories> |

（5）控制台乱码

|  |
| --- |
| <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <configuration>  <forkMode>once</forkMode>  <argLine>-Dfile.encoding=UTF-8</argLine>  <includes>  <include>\*\*/\*Test.java</include>  </includes>  <excludes>  </excludes>  </configuration>  </plugin>  </plugins>  </build> |

1. 逆向工程插件

|  |
| --- |
| <build>  <plugins>  <plugin>  <groupId>org.mybatis.generator</groupId>  <artifactId>mybatis-generator-maven-plugin</artifactId>  <version>1.3.2</version>  <dependencies>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.45</version>  </dependency>  </dependencies>  <configuration>  <!--配置文件的路径，注意修改-->  <configurationFile>${basedir}/src/main/resources/generatorConfig.xml</configurationFile>  <overwrite>true</overwrite>  </configuration>  </plugin>  </plugins>  </build> |

（7）资源上传私服插件

|  |
| --- |
| <build>  <plugins>  <!-- 要将源码放上去，需要加入这个插件 -->  <plugin>  <artifactId>maven-source-plugin</artifactId>  <version>2.1</version>  <configuration>  <attach>true</attach>  </configuration>  <executions>  <execution>  <phase>compile</phase>  <goals>  <goal>jar</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </build>  <!--发布到私服-->  <distributionManagement>  <repository>  <!-- 这里的ID要和setting的id一致 -->  <id>releases</id>  <url>http://localhost:8081/nexus/content/repositories/releases/</url>  </repository>  <!--这是打成快照版本的配置，如果不用这个snapshotRepository标签，打包失败，会报权限问题 -->  <snapshotRepository>  <id>snapshots</id>  <url>http://localhost:8081/nexus/content/repositories/snapshots/</url>  </snapshotRepository>  </distributionManagement> |

# 4.其他

## 4.1log4j

（1）log4j.properties(resources文件夹下；debug等级最低，可以显示INFO)

|  |
| --- |
| log4j.rootLogger=INFO, stdout  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m %n    # Spring  log4j.logger.org.springframework=WARN  log4j.logger.org.hibernate=INFO  #打印sql语句  log4j.logger.api.dao = DEBUG  # General Apache libraries  log4j.logger.org.apache=INFO  # Default Shiro logging  log4j.logger.org.apache.shiro=TRACE  #mybatis  log4j.logger.org.mybatis = ERROR  # Disable verbose logging  log4j.logger.org.apache.shiro.util.ThreadContext=INFO  log4j.logger.org.apache.shiro.cache.ehcache.EhCache=INFO |

## 4.2shiro

### 4.2.1shiro缓存

Ehcache.xml文件 放在resources下

Ehcache配置文件的详细说明

http://blog.csdn.net/mlitsn/article/details/1909192

缓存存活时间和失效时间：

http://www.cnblogs.com/sprinkle/p/6539086.html

设置缓存的大小

http://elim.iteye.com/blog/2116749

|  |
| --- |
| <ehcache>  <diskStore path="java.io.tmpdir"/>  <defaultCache  maxEntriesLocalHeap="2000"  maxElementsInMemory="10000"  eternal="false"  timeToIdleSeconds="120"  timeToLiveSeconds="120"  overflowToDisk="true" />  </ehcache> |

# Mybatis

## 5.1mybatis-config.xml

Resources下，该文件最好建立模板，到时候直接右键模板添加

|  |
| --- |
| <?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>  <settings>  <!--全局设置数据库列表名下划线与实体类属性名大小写问题-->  <setting name="mapUnderscoreToCamelCase" value="true"/>  </settings>    <!--用于解决resultType等类型赋值时不需要输全路径问题-->  <!--<typeAliases>-->  <!--&lt;!&ndash;方法一：一次定义一个对象别名，每个对象需要单独设置&ndash;&gt;-->  <!--<typeAlias type="pojo.TbUser" alias="TbUser"/>-->  <!--&lt;!&ndash;方法二：设置包扫描&ndash;&gt;-->  <!--<package name="pojo"/>-->  <!--</typeAliases>-->  <environments default="development">  <environment id="development">  <transactionManager type="JDBC"/>  <dataSource type="POOLED">  <property name="driver" value="${driver}"/>  <property name="url" value="${url}"/>  <property name="username" value="${username}"/>  <property name="password" value="${password}"/>  </dataSource>  </environment>  </environments>  <mappers>  ##<mapper resource="org/mybatis/example/BlogMapper.xml"/>  <package name="com.qf.dao"/>  </mappers>  </configuration> |

## 5.2逆向工程generatorConfig.xml

Resources文件夹下

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE generatorConfiguration  PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"  "http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">  <generatorConfiguration>  <!--targetRuntime有MyBatis3，MyBatis3Simple，ibatis2,其中MyBatis3会自动生成example文件-->  <!--defaultModelType值默认为conditional，另外还有flat，hierarchical两个值。-->  <context id="Mysql" targetRuntime="MyBatis3Simple" defaultModelType="flat">  <!--分隔符-->  <!--打开关键字设置-->  <property name="autoDelimitKeywords" value="true"/>  <!--关键字前加单引号-->  <property name="beginningDelimiter" value="`"/>  <!--关键字后加单引号-->  <property name="endingDelimiter" value="`"/>  <!-- 一些工具 -->  <plugin type="org.mybatis.generator.plugins.EqualsHashCodePlugin"></plugin>  <plugin type="org.mybatis.generator.plugins.SerializablePlugin"></plugin>  <plugin type="org.mybatis.generator.plugins.ToStringPlugin"></plugin>  <commentGenerator>  <!-- 取消时间戳 -->  <property name="suppressDate" value="true" />  <!-- 是否去除自动生成的注释 true：是 ： false:否 -->  <!-- 1）当某个注释被你删掉之后，你再次点击自动生成，没有注释的内容可能会仍然存在，那么有可能会生成重复内容。  2）当设置为完全没有注释时，那么每次重新生成逆向工程之前，一定要把所有自动生成的文件删掉，否则xml文件会在已有的内容之后添加自动生成的内容，造成同样内容累加 3）综上最好不要删除注释，也最好不要设置关闭注释。4）suppress 抑制-->  <property name="suppressAllComments" value="false" />  </commentGenerator>  <!--数据库链接 参数 -->  <jdbcConnection driverClass="com.mysql.jdbc.Driver"  connectionURL="jdbc:mysql://localhost:3306/oa\_helen" userId="root"  password="123456">  </jdbcConnection>  <javaTypeResolver>  <!-- This property is used to specify whether MyBatis Generator should  force the use of java.math.BigDecimal for DECIMAL and NUMERIC fields, -->  <!-- 默认false，把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer，为 true时把JDBC DECIMAL 和  NUMERIC 类型解析为java.math.BigDecimal -->  <property name="forceBigDecimals" value="true" />  </javaTypeResolver>  <!-- 生成模型的包名和位置 -->  <javaModelGenerator targetPackage="com.qfedu.oa.pojo"  targetProject="src/main/java">  <!-- 是否对数据库查询结果进行trim操作：将接收到的字符串空格去掉，比如数据库定义varchar（10），但填入的内容只占2个字符，则剩余的用空格占位，但pojo中总不能出现空格吧 -->  <property name="trimStrings" value="true" />  </javaModelGenerator>  <!-- 生成映射文件的包名和位置 -->  <sqlMapGenerator targetPackage="com.qfedu.oa.dao"  targetProject="src/main/java">  </sqlMapGenerator>  <!-- 生成DAO的包名和位置 XMLMAPPER | ANNOTATEDMAPPER-->  <javaClientGenerator type="XMLMAPPER"  targetPackage="com.qfedu.oa.dao" targetProject="src/main/java">  </javaClientGenerator>  <!-- 要生成哪些表 -->  <table tableName="sys\_menu" domainObjectName="SysMenu">  <generatedKey column="id" sqlStatement="Mysql"/>  </table>  <table tableName="sys\_role" domainObjectName="SysRole">  <generatedKey column="id" sqlStatement="Mysql"/>  </table>  <table tableName="sys\_user" domainObjectName="SysUser">  <generatedKey column="id" sqlStatement="Mysql"/>  </table>  <table tableName="sys\_role\_menu" domainObjectName="SysRoleMenu">  <generatedKey column="id" sqlStatement="Mysql"/>  </table>  <table tableName="sys\_user\_role" domainObjectName="SysUserRole">  <generatedKey column="id" sqlStatement="Mysql"/>  </table>  <table tableName="sys\_config" domainObjectName="SysConfig">  <generatedKey column="id" sqlStatement="Mysql"/>  </table>  <table tableName="sys\_log" domainObjectName="SysLog">  <generatedKey column="id" sqlStatement="Mysql"/>  </table>  </context>  </generatorConfiguration> |

# 工具类

## 6.1 R对象

|  |
| --- |
| package com.qfedu.springmvc.dto;  import java.util.HashMap;  import java.util.Map;  public class R extends HashMap<String, Object> {  private static final long serialVersionUID = 1L;    public R() {  put("code", 0);  }    public static R error() {  return error(500, "未知异常，请联系管理员");  }    public static R error(String msg) {  return error(500, msg);  }    public static R error(int code, String msg) {  R r = new R();  r.put("code", code);  r.put("msg", msg);  return r;  }  public static R ok(String msg) {  R r = new R();  r.put("msg", msg);  return r;  }    public static R ok(Map<String, Object> map) {  R r = new R();  r.putAll(map);  return r;  }    public static R ok() {  return new R();  }  public R put(String key, Object value) {  super.put(key, value);  return this;  }  } |

# 第五章 项目整合

# 1.通用设置

## 1.1解决idea1.5警告

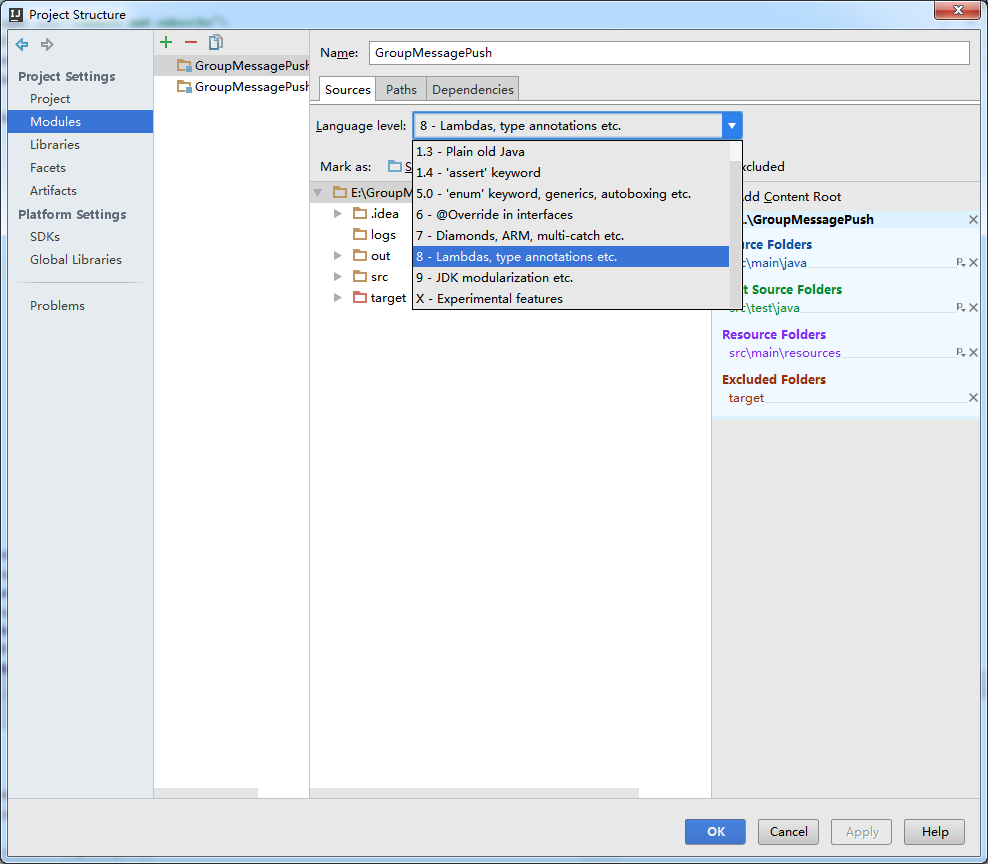
（1）修改Maven的Settings.xml文件添加如下内容

|  |
| --- |
| <profile>  <id>jdk-1.8</id>  <activation>  <activeByDefault>true</activeByDefault>  <jdk>1.8</jdk>  </activation>  <properties>  <maven.compiler.source>1.8</maven.compiler.source>  <maven.compiler.target>1.8</maven.compiler.target>  <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>  </properties>  </profile> |

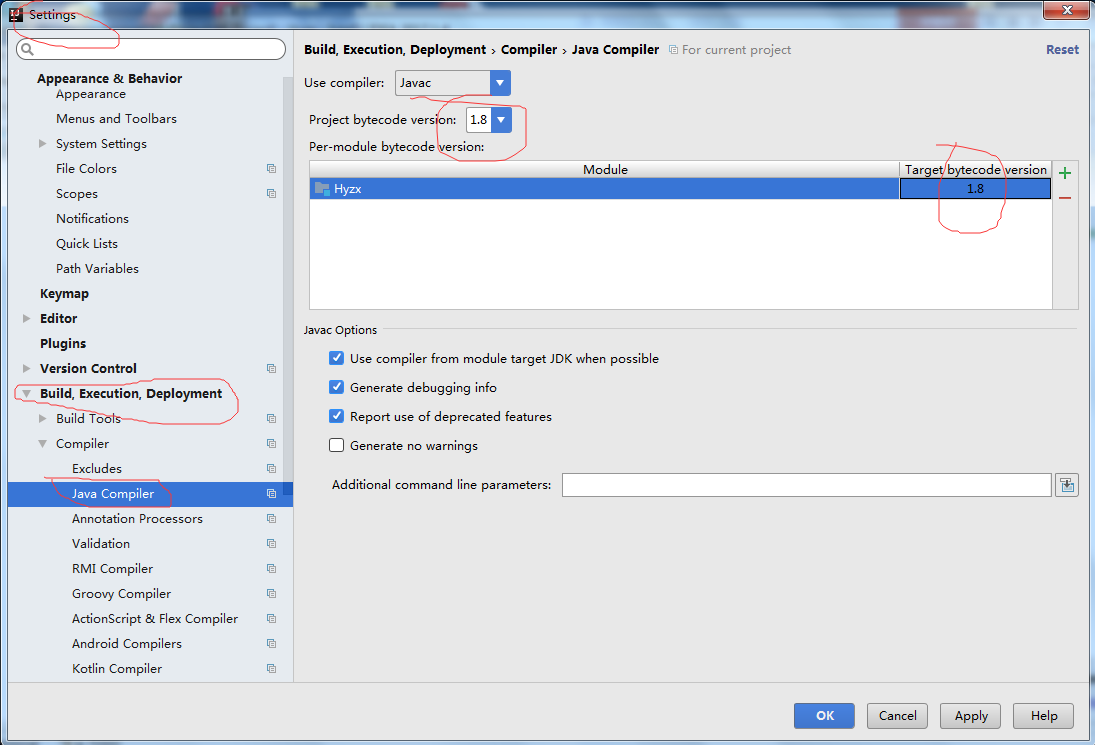
（2）在项目的pom.xml文件中添加：

|  |
| --- |
| <properties>  <maven.compiler.source>1.8</maven.compiler.source>  <maven.compiler.target>1.8</maven.compiler.target>  </properties> |

（3）打开项目配置，设置Modules的Language Level为”8”



（4）设置setting--Java Compiler，将默认jdk和当前modual的jdk版本切换为1.8



## 1.2解决乱码问题

1. 解决post乱码问题

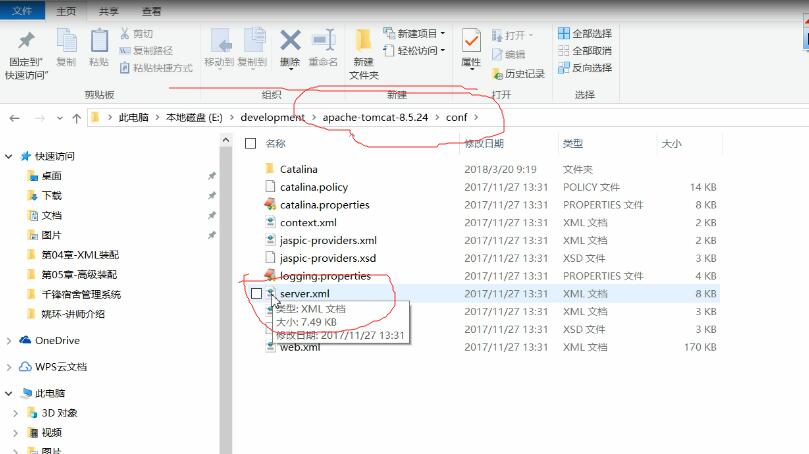
将以下代码插入到web.xml文件

|  |
| --- |
| <!-- 字符编码过滤器 : 放在其他过滤器的前面-->  <filter>  <filter-name>CharacterEncodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>utf-8</param-value>  </init-param>  <init-param>  <param-name>forceEncoding</param-name>  <param-value>true</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>CharacterEncodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

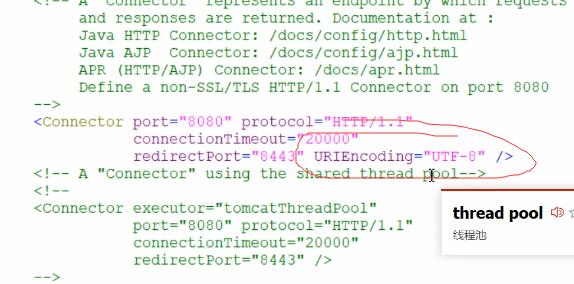
1. get乱码问题

get 乱码 和 tomcat 版本有关，tomcat7和7以下需要解决乱码问题，tomcat8及以上不用。

解决方法如下图：首先在tomcat文件中找到server.xml



然后在文件中该位置插入圈中的设置。URIEncoding="UTF-8"

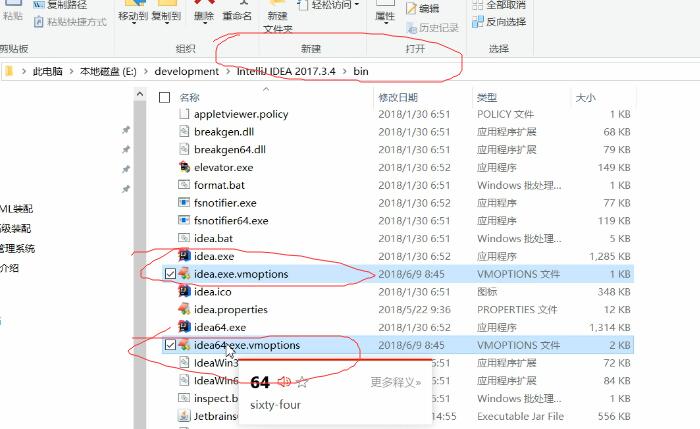


1. idea乱码

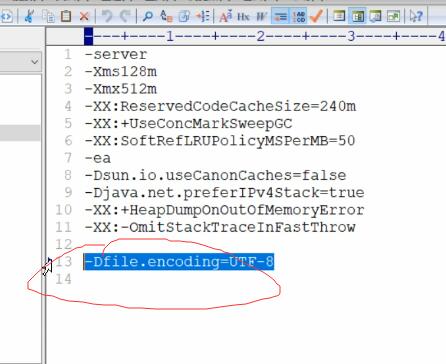
<https://www.cnblogs.com/liaoyanglong/p/6639039.html>

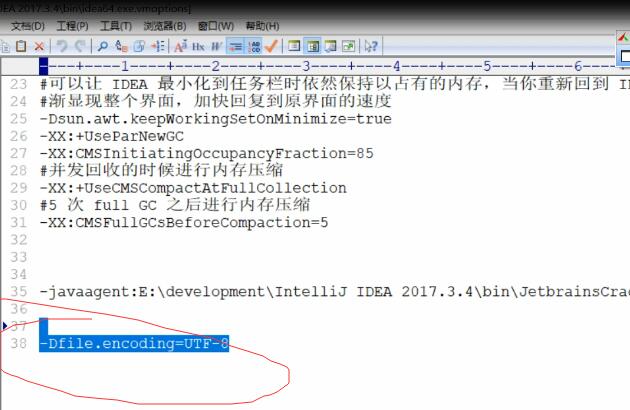
以下方法仅供参考，不全，一切以上边的链接内容为准。

（3.1）找到下图所示两个文件

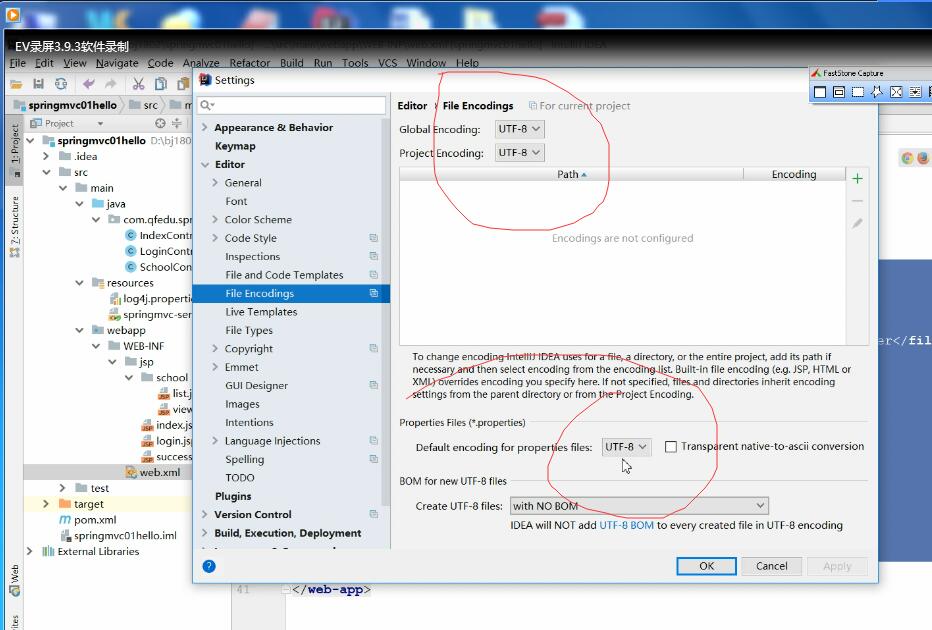


在文件最下面插入圈中所示代码:-Dfile.encoding=UTF-8





（3.2）如下图设置字符码



（3.3）在pom.xml中插入以下插件

|  |
| --- |
| <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <configuration>  <forkMode>once</forkMode>  <argLine>-Dfile.encoding=UTF-8</argLine>  <includes>  <include>\*\*/\*Test.java</include>  </includes>  <excludes>  </excludes>  </configuration>  </plugin>  </plugins>  </build> |

去掉

|  |
| --- |
| <includes>  <include>\*\*/\*Test.java</include>  </includes>  <excludes>  </excludes> |

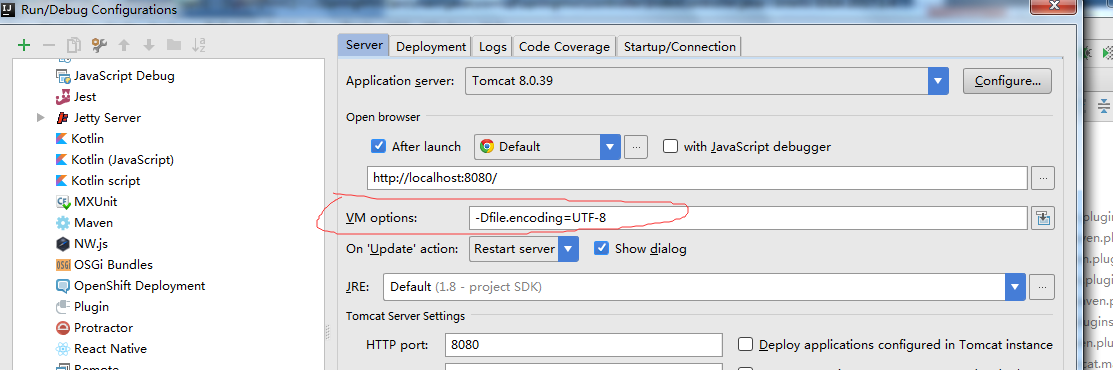
部分后，表示拦截所有。



1. 解决新版Idea乱码问题

一般来说，idea默认会以UTF-8模式启动tomcat，但Idea版本过新，idea启动Tomcat可能不会以UTF-8模式启动，因此造成控制台输出有时候会出现乱码，在前面几个解决乱码设置设置完成的前提下，解决方法如下：

（4.1）不再使用插件tomcat，使用idea自带的独立tomcat，具体的设置流程见2.6节。在设置中有个如下图所示页面，在VM option出加上如下代码：-Dfile.encoding=UTF-8



## 1.3pom中常见设置

### 1.3.1约定版本

|  |
| --- |
| <!--规定相关文件版本-->  <properties>  <spring.version>4.3.13.RELEASE</spring.version>  <hibernate.version>5.2.10.Final</hibernate.version>  <!--日志文件管理包版本-->  <slf4j.version>1.7.21</slf4j.version>  <log4j.version>1.2.17</log4j.version>  </properties> |

# 2.整合ssh

## 2.1准备项目

（1）创建新maven项目：

|  |
| --- |
| <packaging>war</packaging> |

1. 加入jar包

|  |
| --- |
| <dependencies>  <!-- 日志处理 -->  <!-- log start -->  <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>${log4j.version}</version>  </dependency>  <!-- 接口：整合了log4j之外的其他格式的日志显示，需要实现-->  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>${slf4j.version}</version>  </dependency>  <!-- slf4j接口的实现-->  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>${slf4j.version}</version>  </dependency>  <!-- log end -->    <!-- Spring -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aop</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- spring 整合 hibernate -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-orm</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- spring 测试-->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-test</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- hibernate -->  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-core</artifactId>  <version>${hibernate.version}</version>  </dependency>  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-ehcache</artifactId>  <version>${hibernate.version}</version>  </dependency>  <!-- MySql -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.40</version>  </dependency>  <!-- 连接池 -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>1.0.28</version>  </dependency>  <!-- JSP、servlet相关 -->  <dependency>  <groupId>jstl</groupId>  <artifactId>jstl</artifactId>  <version>1.2</version>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  <version>3.1.0</version>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jsp-api</artifactId>  <version>2.0</version>  <scope>provided</scope>  </dependency>  <!-- Jackson Json处理工具包 -->  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  <version>2.8.10</version>  </dependency>  <!-- 单元测试 -->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  </dependency>  </dependencies> |

补充：为什么使用SLF4J而不是Log4J

<https://blog.csdn.net/minicto/article/details/52672472>

（3）基本配置

（3.1）db.properties（resources文件夹下）

|  |
| --- |
| jdbc.jdbcUrl=jdbc:mysql://localhost:3306/hyzx  jdbc.driverClass=com.mysql.jdbc.Driver  jdbc.user=root  jdbc.password=123 |

（3.2）log4j.properties（resources文件夹下）

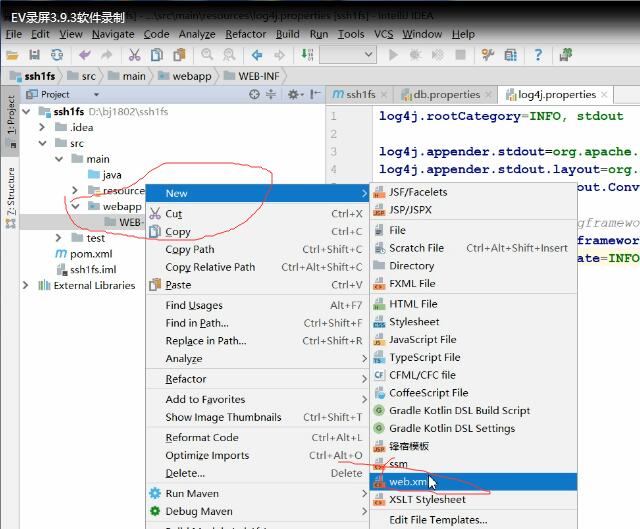
|  |
| --- |
| log4j.rootCategory=INFO, stdout  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %t %c{2}:%L - %m%n  #log4j.category.org.springframework.beans.factory=INFO  log4j.category.org.springframework=INFO  log4j.category.org.hibernate=INFO |

（4）创建数据库

hyzx

（5）配置web.xml

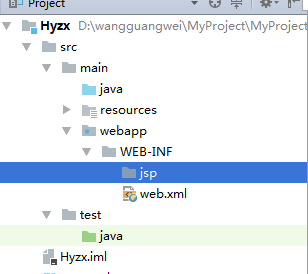
配置在WEB-INF下



web.xml中引入spring和springmvc，欢迎页，配置spring容器监听器，前端控制器，post乱码处理

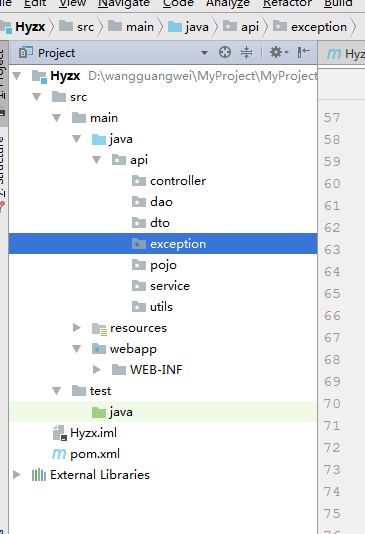
|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"  version="3.1">  <!-- spring：配置spring容器监听器 -->  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring/spring-\*.xml</param-value>  </context-param>  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener>  <!-- 配置SpringMVC的 DispatcherServlet-->  <servlet>  <servlet-name>springmvc</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <!-- 配置 DispatcherServlet 的一个初始化参数: 配置 springmvc 配置文件的位置和名称 -->  <!--  实际上也可以不通过 contextConfigLocation 来配置 springmvc 的配置文件, 而使用默认的.  默认的配置文件为: /WEB-INF/<servlet-name>-servlet.xml  如：springmvc-servlet.xml  -->  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring/springmvc-servlet.xml</param-value>  <!-- <param-value>/WEB-INF/springmvc-servlet.xml</param-value> -->  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <!-- 所有的客户端请求由 DispatcherServlet处理：前端控制器模式-->  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping>  <!-- 字符编码过滤器 : 放在其他过滤器的前面-->  <filter>  <filter-name>CharacterEncodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>utf-8</param-value>  </init-param>  <init-param>  <param-name>forceEncoding</param-name>  <param-value>true</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>CharacterEncodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping>  <!--  配置 org.springframework.web.filter.HiddenHttpMethodFilter: 可以把 POST 请求转为 DELETE 或 PUT 请求  -->  <filter>  <filter-name>HiddenHttpMethodFilter</filter-name>  <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>HiddenHttpMethodFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping>  </web-app> |

1. jsp文件夹



1. 包

controller、service、dao、pojo、utils、dto、exception



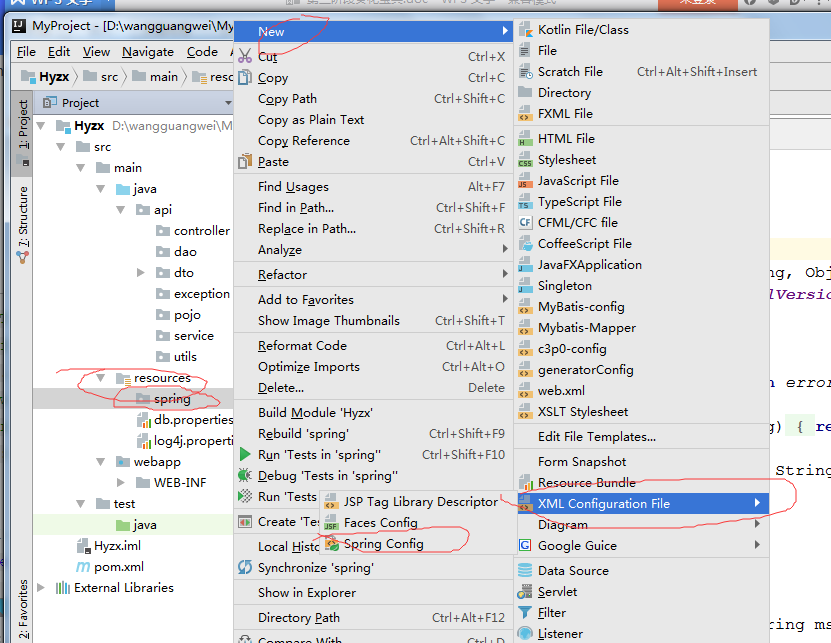
（8）R.class

|  |
| --- |
| package api.dto;  import java.util.HashMap;  import java.util.Map;  public class R extends HashMap<String, Object> {  private static final long serialVersionUID = 1L;  public R() {  }  public static R create() {  return new R();  }  public R put(Object value) {  R r = new R();  r.put("value",value);  return r;  }  public R put(String key, Object value) {  //引入super调用HashMap的put方法是为了和本方法名put区分开，防止出错。  super.put(key, value);  return this;  }  // public static R error() {  // return error(500, "未知异常，请联系管理员");  // }  //  // public static R error(String msg) {  // return error(500, msg);  // }  //  // public static R error(int code, String msg) {  // R r = new R();  // r.put("code", code);  // r.put("msg", msg);  // return r;  // }  //  // public static R ok(int code, String msg) {  // R r = new R();  // r.put("code", code);  // r.put("msg", msg);  // return r;  // }  //  // public static R ok(String msg) {  // R r = new R();  // r.put("msg", msg);  // return r;  // }  //  // public static R ok(Map<String, Object> map) {  // R r = new R();  // r.putAll(map);  // return r;  // }  } |

（9）配置spring文件

（9.1）spring/springmvc-servlet.xml

在resources文件夹下建立spring文件夹，然后建立springmvc-servlet.xml，并复制其内容。



spring/springmvc-servlet.xml：包扫描、视图解析器、注解驱动、静态资源

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context" xmlns:p="http://www.springframework.org/schema/p"  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.xsd http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd">  <!-- 扫描包下所有标注@Controller的组件 -->  <context:component-scan base-package="api.controller"/>  <!--扫描切面-->  <context:component-scan base-package="api.exception"/>  <!-- 视图解析器 -->  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"  p:prefix="/WEB-INF/jsp/"  p:suffix=".jsp" />  <!-- 配置注解驱动 -->  <mvc:annotation-driven />  <!-- 处理静态资源 -->  <mvc:default-servlet-handler/>  </beans> |

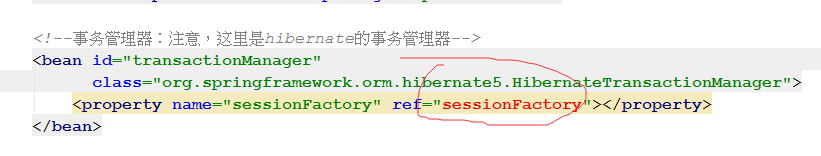
（9.2）spring/spring-dao.xml

|  |
| --- |
| <?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:context="http://www.springframework.org/schema/context"  xmlns:util="http://www.springframework.org/schema/util" xmlns:p="http://www.springframework.org/schema/p"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd">  <!-- 扫描dao -->  <context:component-scan base-package="api.dao" />  <!-- 加载数据库配置文件和其他配置文件 -->  <context:property-placeholder location="classpath:db.properties" />  <!-- 数据库连接池 -->  <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"  destroy-method="close">  <property name="url" value="${jdbc.jdbcUrl}" />  <property name="username" value="${jdbc.user}" />  <property name="password" value="${jdbc.password}" />  <property name="driverClassName" value="${jdbc.driverClass}" />  <!-- 初始化连接大小 -->  <property name="initialSize" value="0" />  <!-- 连接池最大使用连接数量 -->  <property name="maxActive" value="20" />  <!-- 连接池最小空闲 -->  <property name="minIdle" value="0" />  <!-- 获取连接最大等待时间 -->  <property name="maxWait" value="60000" />  </bean>  <util:properties id="hibernateProperties">  <prop key="hibernate.dialect">org.hibernate.dialect.MySQL57Dialect</prop>  <prop key="hibernate.show\_sql">true</prop>  <prop key="hibernate.format\_sql">true</prop>  <prop key="hibernate.hbm2ddl.auto">update</prop>  <!--单独使用hibernate时才需要的配置-->  <!--<prop key="current\_session\_context\_class">thread</prop>-->  </util:properties>  <!-- 创建session工厂-->  <!--用spring的session，整合hibernate的时候要注意版本号，版本号不同配置不同，本程序用的是hibernate5-->  <!--所以自己在网上搜配置文件时，要规定好spring和hibernate版本号前两位；例如搜：spring4.3与hibernate5.2整合-->  <bean id="sessionFactory"  class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">  <!-- 将连接池注入到sessionFactory -->  <property name="dataSource" ref="dataSource" />  <!-- 配置hibernate基本信息 -->  <property name="hibernateProperties" ref="hibernateProperties" />  <!-- 扫描实体类的位置（jpa注解使用该语句配置） -->  <property name="packagesToScan" value="api.pojo" />  <!-- 指定映射文件所在的目录（使用xml注解时使用该句配置）-->  <!-- <property name="mappingDirectoryLocations" value="classpath:api/pojo" /> -->  </bean>  </beans> |

（9.3）spring/spring-service.xml

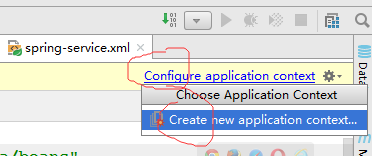
|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:aop="http://www.springframework.org/schema/aop" xmlns:p="http://www.springframework.org/schema/p"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">  <!-- 扫描service -->  <context:component-scan base-package="api.service" />  <!--事务管理器：注意，这里是hibernate的事务管理器-->  <bean id="transactionManager"  class="org.springframework.orm.hibernate5.HibernateTransactionManager">  <property name="sessionFactory" ref="sessionFactory"></property>  </bean>  <!-- 通知 -->  <tx:advice id="txAdvice" transaction-manager="transactionManager">  <tx:attributes>  <!-- 传播行为 -->  <!-- REQUIRED：如果有事务，则在事务中执行；如果没有事务，则开启一个新的事物 -->  <tx:method name="save\*" propagation="REQUIRED" />  <tx:method name="insert\*" propagation="REQUIRED" />  <tx:method name="add\*" propagation="REQUIRED" />  <tx:method name="create\*" propagation="REQUIRED" />  <tx:method name="delete\*" propagation="REQUIRED" />  <tx:method name="update\*" propagation="REQUIRED" />  <!-- SUPPORTS：如果有事务，则在事务中执行；如果没有事务，则不会开启事物 -->  <tx:method name="find\*" propagation="SUPPORTS" read-only="true" />  <tx:method name="select\*" propagation="SUPPORTS" read-only="true" />  <tx:method name="get\*" propagation="SUPPORTS" read-only="true" />  <tx:method name="query\*" propagation="SUPPORTS" read-only="true" />  <!--当以上事务都不符合业务的name时全都按照下面的设定进行事务配置-->  <tx:method name="\*" propagation="SUPPORTS" read-only="true" />  </tx:attributes>  </tx:advice>  <!-- 切面 -->  <aop:config>  <!-- 定义在service包或者子包里的任意方法的执行 -->  <aop:pointcut id="txPointCut"  expression="execution(\* api.service..\*.\*(..))" />  <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut" />  </aop:config>  </beans> |

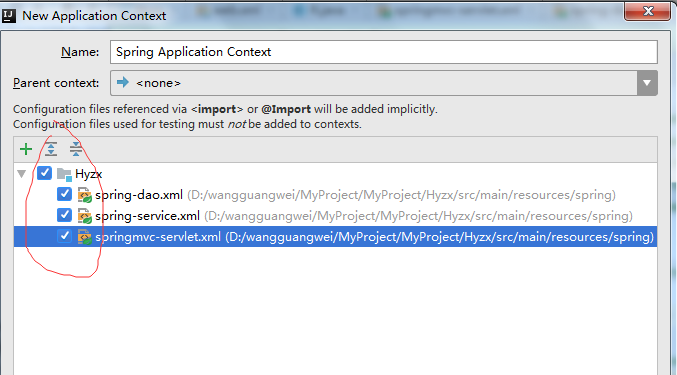
（9.5）三个文件配置完后，会发现下图所示区域爆红，这说明三个文件之间还没有进行关联，有些id获取不到。



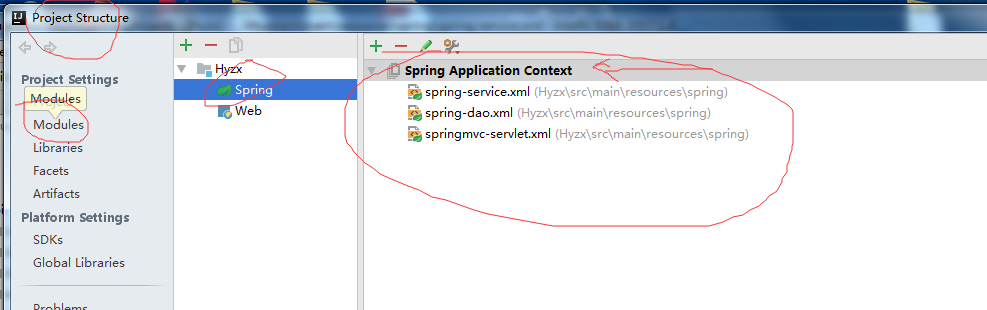
所以接下来需要给他们创建关联：

随便找三个文件中任意一个，然后按下面图配置：

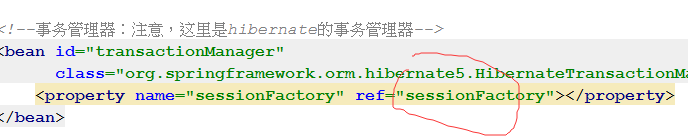




点击ok，然后检查是否在一块儿了：



如上图所示，三个文件都在箭头所指区域下，锁以配置完成，如下图所示，ref不报红了。



## 2.2第一套的基本代码

（1）创建实体类

|  |
| --- |
| package api.pojo;  import javax.persistence.\*;  //标明它是实体类，用于pojo扫描  @Entity  //标明对应数据库表格  @Table(name="t\_user")  public class User {  //表明主键  @Id  //设定自增长策略  @GeneratedValue(strategy = GenerationType.IDENTITY)  //设置对应数据库表的列,列名最好加前缀，防止遇到列名为关键字，影响sql语句执行  @Column(name="u\_id")  private Integer id;  @Column(name = "u\_username")  private String username;  @Column(name = "u\_password")  private String password;  @Column(name = "u\_email")  private String email;  @Column(name = "u\_flag")  private Integer flag;  @Column(name = "u\_role")  private Integer role;  @Column(name = "u\_phone")  private String phone;  @Column(name = "u\_code")  private String code;  public User() {  }  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 getEmail() {  return email;  }  public void setEmail(String email) {  this.email = email;  }  public Integer getFlag() {  return flag;  }  public void setFlag(Integer flag) {  this.flag = flag;  }  public Integer getRole() {  return role;  }  public void setRole(Integer role) {  this.role = role;  }  public String getPhone() {  return phone;  }  public void setPhone(String phone) {  this.phone = phone;  }  public String getCode() {  return code;  }  public void setCode(String code) {  this.code = code;  }  @Override  public String toString() {  return "User{" +  "id=" + id +  ", username='" + username + '\'' +  ", password='" + password + '\'' +  ", email='" + email + '\'' +  ", flag=" + flag +  ", role=" + role +  ", phone='" + phone + '\'' +  ", code='" + code + '\'' +  '}';  }  } |

（2）创建user对应的dao接口和实现

|  |
| --- |
| package api.dao;  import api.pojo.User;  import java.util.List;  public interface UserDao {  List<User> queryList();  } |
| package api.dao.impl;  import api.dao.UserDao;  import api.pojo.User;  import org.hibernate.Session;  import org.hibernate.SessionFactory;  import org.hibernate.query.Query;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Repository;  import java.util.List;  @Repository  public class UserDaoImpl implements UserDao{  @Autowired  private SessionFactory sessionFactory;  public Session getCurrentSession(){  return sessionFactory.getCurrentSession();  }  public List<User> queryList() {  Query<User> query = getCurrentSession().createQuery("from User", User.class);  return query.list();  }  } |

1. srvice层接口和实现

|  |
| --- |
| package api.service;  import api.pojo.User;  import java.util.List;  public interface UserService {  //当业务层添加方法后，要去spring-service.xml中看一下事务设置是否合适，进行相应的添加删除事务的配置  List<User> queryList();  } |
| package api.service.impl;  import api.dao.UserDao;  import api.pojo.User;  import api.service.UserService;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import java.util.List;  @Service  public class UserServiceImpl implements UserService {  @Autowired  private UserDao userDao;  public List<User> queryList() {  List<User> userList = userDao.queryList();  return userList;  }  } |

1. Controller

|  |
| --- |
| package api.controller;  import api.dto.R;  import api.pojo.User;  import api.service.UserService;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Controller;  import org.springframework.web.bind.annotation.\*;  import java.util.List;  @RequestMapping("/api")  @Controller  //@RestController可以替代@Controller，此时下面的@ResponseBody可以省略  //@RestController  public class UserController {  @Autowired  private UserService userService;  //@GetMapping("\*"),@postMapping("\*")可以替代@RequestMapping("\*")从而实现restful风格，明确请求方式  @ResponseBody  @RequestMapping(value = "/queryUserList",method= RequestMethod.GET)  public R queryUserList(){  List<User> userList = userService.queryUserList();  return R.create().put(userList);  }  } |

（5）配置TomCat并用postMan测试

-Dfile.encoding=UTF-8

# 3 SSM整合

## 3.1准备项目

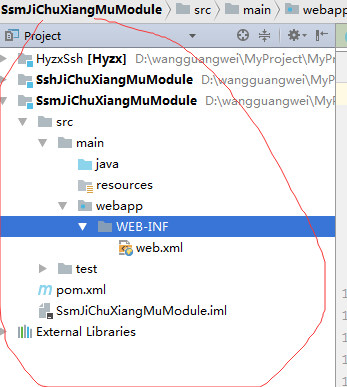
### 3.1.1 基本架构准备

#### 3.1.1.1项目基础创建

（1）新建项目，打包，引入pom，创建数据库 oa\_ssm，执行脚本：权限管理基础表.sql。

|  |
| --- |
| <packaging>war</packaging> |
| <properties>  <maven.compiler.source>1.8</maven.compiler.source>  <maven.compiler.target>1.8</maven.compiler.target>  <spring.version>4.3.13.RELEASE</spring.version>  <mybatis.version>3.4.5</mybatis.version>  <!-- log4j日志文件管理包版本 -->  <slf4j.version>1.7.21</slf4j.version>  <log4j.version>1.2.17</log4j.version>  </properties>  <dependencies>  <!-- 日志处理 -->  <!-- log start -->  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>${slf4j.version}</version>  </dependency>  <!-- log end -->  <!-- Spring -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aop</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-tx</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- spring 整合 mybatis包 -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis-spring</artifactId>  <version>1.3.2</version>  </dependency>  <!-- spring 测试-->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-test</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- mybatis -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>${mybatis.version}</version>  </dependency>  <!-- MySql -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.40</version>  </dependency>  <!-- 连接池 -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>1.0.28</version>  </dependency>  <!-- JSP、servlet相关 -->  <dependency>  <groupId>jstl</groupId>  <artifactId>jstl</artifactId>  <version>1.2</version>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  <version>3.1.0</version>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jsp-api</artifactId>  <version>2.0</version>  <scope>provided</scope>  </dependency>  <!-- Jackson Json处理工具包 -->  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  <version>2.8.10</version>  </dependency>  <!-- 单元测试 -->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <scope>test</scope>  </dependency>  </dependencies>  <build>  <!-- 拷贝资源文件 -->  <!-- 添加此节点将mybatis的XxxMapper.xml文件发布到target路径下 -->  <resources>  <resource>  <directory>src/main/java</directory>  <includes>  <!-- <include>\*\*/\*.properties</include> -->  <include>\*\*/\*.xml</include>  </includes>  <filtering>false</filtering>  </resource>  <!-- 上面的配置导致src/main/resources的所有文件都不能被扫描，因此还要配多一个 -->  <resource>  <directory>src/main/resources</directory>  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.xml</include>  </includes>  <filtering>false</filtering>  </resource>  </resources>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <configuration>  <forkMode>once</forkMode>  <argLine>-Dfile.encoding=UTF-8</argLine>  </configuration>  </plugin>  </plugins>  </build> |

（2）完善文件夹



Web.xml代码如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"  version="3.1">  <!-- spring：配置spring容器监听器 -->  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring/spring-\*.xml</param-value>  </context-param>  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener>  <!-- 配置SpringMVC的 DispatcherServlet-->  <servlet>  <servlet-name>springmvc</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <!-- 配置 DispatcherServlet 的一个初始化参数: 配置 springmvc 配置文件的位置和名称 -->  <!--  实际上也可以不通过 contextConfigLocation 来配置 springmvc 的配置文件, 而使用默认的.  默认的配置文件为: /WEB-INF/<servlet-name>-servlet.xml  如：springmvc-servlet.xml  -->  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring/springmvc-servlet.xml</param-value>  <!-- <param-value>/WEB-INF/springmvc-servlet.xml</param-value> -->  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <!-- 所有的客户端请求由 DispatcherServlet处理：前端控制器模式-->  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping>  <!-- 字符编码过滤器 : 放在其他过滤器的前面-->  <filter>  <filter-name>CharacterEncodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>utf-8</param-value>  </init-param>  <init-param>  <param-name>forceEncoding</param-name>  <param-value>true</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>CharacterEncodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping>  <!--  配置 org.springframework.web.filter.HiddenHttpMethodFilter: 可以把 POST 请求转为 DELETE 或 PUT 请求  -->  <!--<filter>-->  <!--<filter-name>HiddenHttpMethodFilter</filter-name>-->  <!--<filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>-->  <!--</filter>-->  <!--<filter-mapping>-->  <!--<filter-name>HiddenHttpMethodFilter</filter-name>-->  <!--<url-pattern>/\*</url-pattern>-->  <!--</filter-mapping>-->  </web-app> |

#### 3.1.1.2逆向工程引入dao，pojo

（1）另建临时项目，生成逆向工程自动文件，最好不要在项目中使用逆向工程，避免误操作造成所有文件都删除重新生成的问题。

另建项目时，文件夹架构要和项目一致，方便复制。

（2）导入逆向工程pom

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.mybatis.generator</groupId>  <artifactId>mybatis-generator-core</artifactId>  <version>1.3.6</version>  </dependency>  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>3.4.5</version>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.mybatis.generator</groupId>  <artifactId>mybatis-generator-maven-plugin</artifactId>  <version>1.3.6</version>  <dependencies>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.45</version>  </dependency>  </dependencies>  <configuration>  <!--配置文件的路径-->  <configurationFile>${basedir}/src/main/resources/generatorConfig.xml</configurationFile>  <overwrite>true</overwrite>  </configuration>  </plugin>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <configuration>  <forkMode>once</forkMode>  <argLine>-Dfile.encoding=UTF-8</argLine>  <excludes>  </excludes>  </configuration>  </plugin>  </plugins>  </build> |

（3）resources下创建 generatorConfig.xml

配置参考：<https://blog.csdn.net/isea533/article/details/42102297>

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE generatorConfiguration  PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"  "http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">  <generatorConfiguration>  <!--targetRuntime有MyBatis3，MyBatis3Simple，ibatis2,其中MyBatis3会自动生成example文件-->  <!--defaultModelType值默认为conditional，另外还有flat，hierarchical两个值。-->  <context id="Mysql" targetRuntime="MyBatis3" defaultModelType="flat">  <!--分隔符-->  <!--打开关键字设置-->  <property name="autoDelimitKeywords" value="true"/>  <!--关键字前加单引号-->  <property name="beginningDelimiter" value="`"/>  <!--关键字后加单引号-->  <property name="endingDelimiter" value="`"/>  <!-- 一些工具 -->  <plugin type="org.mybatis.generator.plugins.EqualsHashCodePlugin"></plugin>  <plugin type="org.mybatis.generator.plugins.SerializablePlugin"></plugin>  <plugin type="org.mybatis.generator.plugins.ToStringPlugin"></plugin>  <commentGenerator>  <!-- 取消时间戳 -->  <property name="suppressDate" value="true"/>  <!-- 是否去除自动生成的注释 true：是 ： false:否 -->  <!-- 1）当某个注释被你删掉之后，你再次点击自动生成，没有注释的内容可能会仍然存在，那么有可能会生成重复内容。  2）当设置为完全没有注释时，那么每次重新生成逆向工程之前，一定要把所有自动生成的文件删掉，否则xml文件会在已有的内容之后添加自动生成的内容，造成同样内容累加 3）综上最好不要删除注释，也最好不要设置关闭注释。4）suppress 抑制-->  <property name="suppressAllComments" value="false"/>  </commentGenerator>    <!--数据库链接 参数 -->  <jdbcConnection driverClass="com.mysql.jdbc.Driver"  connectionURL="jdbc:mysql://localhost:3306/oa\_ssm" userId="root"  password="123">  </jdbcConnection>  <javaTypeResolver>  <!-- This property is used to specify whether MyBatis Generator should  force the use of java.math.BigDecimal for DECIMAL and NUMERIC fields, -->  <!-- 默认false，把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer，为 true时把JDBC DECIMAL 和  NUMERIC 类型解析为java.math.BigDecimal -->  <property name="forceBigDecimals" value="true"/>  </javaTypeResolver>    <!-- 生成模型的包名和位置 -->  <javaModelGenerator targetPackage="api.sys.pojo"  targetProject="src/main/java">  <!-- 是否对数据库查询结果进行trim操作：将接收到的字符串空格去掉，比如数据库定义varchar（10），但填入的内容只占2个字符，则剩余的用空格占位，但pojo中总不能出现空格吧 -->  <property name="trimStrings" value="true"/>  </javaModelGenerator>    <!-- 生成映射文件的包名和位置 -->  <sqlMapGenerator targetPackage="api.sys.dao"  targetProject="src/main/java">  </sqlMapGenerator>    <!-- 生成DAO的包名和位置 XMLMAPPER | ANNOTATEDMAPPER-->  <!-- XMLMAPPER 使用XML方式| ANNOTATEDMAPPER 使用注解方式-->  <javaClientGenerator type="XMLMAPPER"  targetPackage="api.sys.dao" targetProject="src/main/java">  </javaClientGenerator>    <!-- 要生成哪些表 -->  <table tableName="sys\_menu" domainObjectName="SysMenu">  <generatedKey column="id" sqlStatement="Mysql"/>  </table>  <table tableName="sys\_role" domainObjectName="SysRole">  <generatedKey column="id" sqlStatement="Mysql"/>  </table>  <table tableName="sys\_user" domainObjectName="SysUser">  <generatedKey column="id" sqlStatement="Mysql"/>  </table>  <table tableName="sys\_role\_menu" domainObjectName="SysRoleMenu">  <generatedKey column="id" sqlStatement="Mysql"/>  </table>  <table tableName="sys\_user\_role" domainObjectName="SysUserRole">  <generatedKey column="id" sqlStatement="Mysql"/>  </table>  <table tableName="sys\_config" domainObjectName="SysConfig">  <generatedKey column="id" sqlStatement="Mysql"/>  </table>  <table tableName="sys\_log" domainObjectName="SysLog">  <generatedKey column="id" sqlStatement="Mysql"/>  </table>  </context>  </generatorConfiguration> |

（3.4）运行插件

mybatis-generator:generate

（3.5）将生成的java文件夹下的文件平行复制到项目中

### 3.1.2配置文件

1. log4j.properties（resources下）

|  |
| --- |
| log4j.rootLogger=INFO, stdout  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %t %c{2}:%L - %m%n  #log4j.category.org.springframework.beans.factory=INFO  log4j.logger.org.springframework=INFO  #mybatis  log4j.logger.org.mybatis = ERROR  #打印sql语句  log4j.logger.api.dao = DEBUG |

1. db.properties（resources下）

|  |
| --- |
| jdbc.jdbcUrl=jdbc:mysql://localhost:3306/oa\_ssm  jdbc.driverClass=com.mysql.jdbc.Driver  jdbc.user=root  jdbc.password=123 |

1. mybatis-config.xml（idea有模板）（resources/mybatis文件夹下）

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?>  <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>  <!--<typeAliases>-->  <!--&lt;!&ndash;方法一：一次定义一个对象别名，每个对象需要单独设置&ndash;&gt;-->  <!--<typeAlias type="pojo.TbUser" alias="TbUser"/>-->  <!--&lt;!&ndash;方法二：设置包扫描&ndash;&gt;-->  <!--<package name="pojo"/>-->  <!--</typeAliases>-->  <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/oa\_ssm?characterEncoding=UTF-8"/>  <property name="username" value="root"/>  <property name="password" value="123"/>  </dataSource>  </environment>  </environments>  <mappers>  <package name="api.dao"/>  </mappers>  </configuration> |

（4）spring-dao.xml（resources/spring文件夹下）

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:util="http://www.springframework.org/schema/util"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd">  <!-- 扫描dao -->  <context:component-scan base-package="api.dao" />  <!-- 加载数据库配置文件和其他配置文件 -->  <context:property-placeholder location="classpath:db.properties" />  <!-- 数据库连接池 -->  <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"  destroy-method="close">  <property name="url" value="${jdbc.jdbcUrl}" />  <property name="username" value="${jdbc.user}" />  <property name="password" value="${jdbc.password}" />  <property name="driverClassName" value="${jdbc.driverClass}" />  <!-- 初始化连接大小 -->  <property name="initialSize" value="0" />  <!-- 连接池最大使用连接数量 -->  <property name="maxActive" value="20" />  <!-- 连接池最小空闲 -->  <property name="minIdle" value="0" />  <!-- 获取连接最大等待时间 -->  <property name="maxWait" value="60000" />  </bean>  <!--整合mybatis：sqlSessionFactory-->  <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  <property name="dataSource" ref="dataSource" />  <property name="configLocation" value="classpath:mybatis/mybatis-config.xml" />  </bean>  <!--配置dao层：扫描Mapper文件，用于生成Mapper的代理对象-->  <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  <property name="basePackage" value="api.dao" />  <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />  </bean>  </beans> |

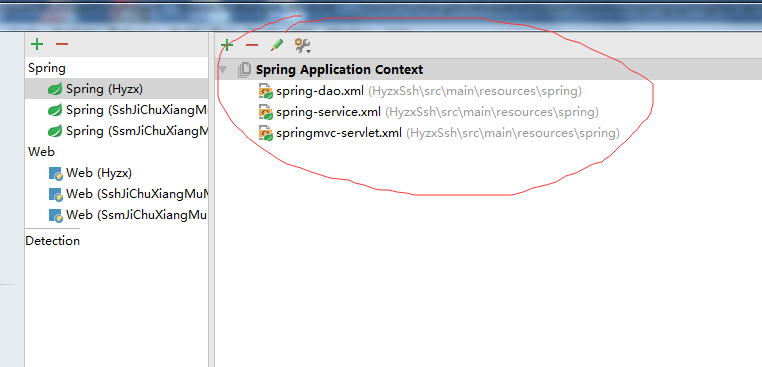
（5）spring-service.xml（resources/spring文件夹下）

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">  <!-- 扫描service -->  <context:component-scan base-package="api.service" />  <!--事务管理器-->  <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource" />  </bean>  <!-- 通知 -->  <tx:advice id="txAdvice" transaction-manager="transactionManager">  <tx:attributes>  <!-- 传播行为 -->  <!-- REQUIRED：如果有事务，则在事务中执行；如果没有事务，则开启一个新的事物 -->  <tx:method name="save\*" propagation="REQUIRED" />  <tx:method name="insert\*" propagation="REQUIRED" />  <tx:method name="add\*" propagation="REQUIRED" />  <tx:method name="create\*" propagation="REQUIRED" />  <tx:method name="delete\*" propagation="REQUIRED" />  <tx:method name="update\*" propagation="REQUIRED" />  <!-- SUPPORTS：如果有事务，则在事务中执行；如果没有事务，则不会开启事物 -->  <tx:method name="find\*" propagation="SUPPORTS" read-only="true" />  <tx:method name="select\*" propagation="SUPPORTS" read-only="true" />  <tx:method name="get\*" propagation="SUPPORTS" read-only="true" />  <tx:method name="query\*" propagation="SUPPORTS" read-only="true" />  <tx:method name="\*" propagation="SUPPORTS" read-only="true" />  </tx:attributes>  </tx:advice>  <!-- 切面 -->  <aop:config>  <!-- 定义在service包或者子包里的任意方法的执行 -->  <aop:pointcut id="txPointCut"  expression="execution(\* api.service..\*.\*(..))" />  <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut" />  </aop:config>  </beans> |

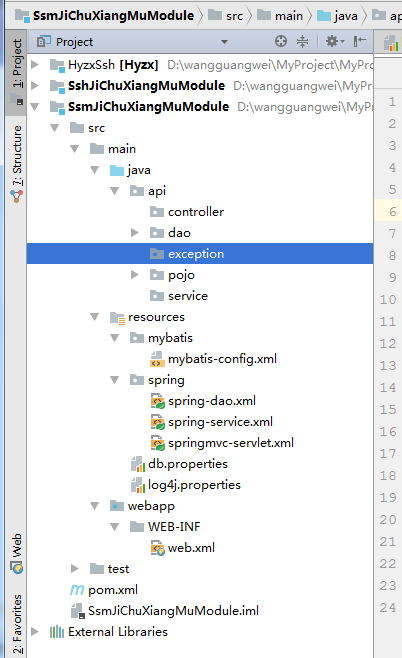
（6）springmvc-servlet.xml（resources/spring文件夹下）

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context" xmlns:p="http://www.springframework.org/schema/p"  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.xsd http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd">  <!-- 扫描包下所有标注@Controller的组件 -->  <context:component-scan base-package="api.controller"/>  <!--扫描切面-->  <context:component-scan base-package="api.exception"/>  <!-- 视图解析器 -->  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"  p:prefix="/WEB-INF/jsp/"  p:suffix=".jsp" />  <!-- 配置注解驱动 -->  <mvc:annotation-driven />  <!-- 处理静态资源 -->  <mvc:default-servlet-handler/>  </beans> |

（7）将三个spring文件加载到一个，目录下



（7）根据xml文件增加相应的包路径



## 3.2第一套基本代码

### 3.2.1开发业务层

|  |
| --- |
| package api.service.sys;  import api.pojo.sys.SysMenu;  import java.util.List;  /\*\*  \* Created by Administrator on 2018/6/25.  \*/  public interface SysMenuService {  List<SysMenu> selectAllSysMenu();  } |
| package api.service.impl.sys;  import api.dao.sys.SysMenuMapper;  import api.pojo.sys.SysMenu;  import api.service.sys.SysMenuService;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import java.util.List;  /\*\*  \* Created by Administrator on 2018/6/25.  \*/  @Service  public class SysMenuServiceImpl implements SysMenuService {  //这个地方提示无法自动注入，不用管它  @Autowired  private SysMenuMapper sysMenuMapper;  public List<SysMenu> selectAllSysMenu() {  List<SysMenu> sysMenus = sysMenuMapper.selectByExample(null);  return sysMenus;  }  } |

### 3.2.2开发控制层

|  |
| --- |
| package api.controller;  import api.pojo.sys.SysMenu;  import api.service.sys.SysMenuService;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Controller;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.ResponseBody;  import java.util.List;  /\*\*  \* Created by Administrator on 2018/6/25.  \*/  @RequestMapping("api")  @Controller  public class SysController {  @Autowired  private SysMenuService sysMenuService;  @ResponseBody  @RequestMapping("sys/selectAllSysMenu")  public List<SysMenu> selectAllSysMenu(){  return sysMenuService.selectAllSysMenu();  }  } |

启动服务，测试

## 补充：权限管理基础表.sql

|  |
| --- |
| /\*  Navicat MySQL Data Transfer  Source Server : localhost  Source Server Version : 50721  Source Host : localhost:3306  Source Database : oa\_bj1802  Target Server Type : MYSQL  Target Server Version : 50721  File Encoding : 65001  Date: 2018-06-23 16:17:18  \*/  SET FOREIGN\_KEY\_CHECKS=0;  -- ----------------------------  -- Table structure for sys\_config  -- ----------------------------  DROP TABLE IF EXISTS `sys\_config`;  CREATE TABLE `sys\_config` (  `id` bigint(20) NOT NULL AUTO\_INCREMENT,  `key` varchar(50) DEFAULT NULL COMMENT 'key',  `value` varchar(2000) DEFAULT NULL COMMENT 'value',  `status` tinyint(4) DEFAULT '1' COMMENT '状态 0：隐藏 1：显示',  `remark` varchar(500) DEFAULT NULL COMMENT '备注',  PRIMARY KEY (`id`),  UNIQUE KEY `key` (`key`)  ) ENGINE=InnoDB AUTO\_INCREMENT=11 DEFAULT CHARSET=utf8 COMMENT='系统配置信息表';  -- ----------------------------  -- Records of sys\_config  -- ----------------------------  INSERT INTO `sys\_config` VALUES ('1', 'CLOUD\_STORAGE\_CONFIG\_KEY', '{\"aliyunAccessKeyId\":\"\",\"aliyunAccessKeySecret\":\"\",\"aliyunBucketName\":\"\",\"aliyunDomain\":\"\",\"aliyunEndPoint\":\"\",\"aliyunPrefix\":\"\",\"qcloudBucketName\":\"\",\"qcloudDomain\":\"\",\"qcloudPrefix\":\"\",\"qcloudSecretId\":\"\",\"qcloudSecretKey\":\"\",\"qiniuAccessKey\":\"NrgMfABZxWLo5B-YYSjoE8-AZ1EISdi1Z3ubLOeZ\",\"qiniuBucketName\":\"ios-app\",\"qiniuDomain\":\"http://7xqbwh.dl1.z0.glb.clouddn.com\",\"qiniuPrefix\":\"upload\",\"qiniuSecretKey\":\"uIwJHevMRWU0VLxFvgy0tAcOdGqasdtVlJkdy6vV\",\"type\":1}', '0', '云存储配置信息');  INSERT INTO `sys\_config` VALUES ('2', 'SHIRO\_FILTER\_CONFIG\_01', '/images/\*\*=anon', '1', 'shiro过滤器');  INSERT INTO `sys\_config` VALUES ('5', 'SHIRO\_FILTER\_CONFIG\_02', '/css/\*\*=anon', '1', 'shiro过滤器');  INSERT INTO `sys\_config` VALUES ('6', 'SHIRO\_FILTER\_CONFIG\_03', '/libs/\*\*=anon', '1', 'shiro过滤器');  INSERT INTO `sys\_config` VALUES ('7', 'SHIRO\_FILTER\_CONFIG\_04', '/login.html=anon', '1', 'shiro过滤器');  INSERT INTO `sys\_config` VALUES ('8', 'SHIRO\_FILTER\_CONFIG\_05', '/sys/login=anon', '1', 'shiro过滤器');  INSERT INTO `sys\_config` VALUES ('9', 'SHIRO\_FILTER\_CONFIG\_06', '/captcha.jpg=anon', '1', 'shiro过滤器');  INSERT INTO `sys\_config` VALUES ('10', 'SHIRO\_FILTER\_CONFIG\_99', '/\*\*=user', '1', 'shiro过滤器');  -- ----------------------------  -- Table structure for sys\_log  -- ----------------------------  DROP TABLE IF EXISTS `sys\_log`;  CREATE TABLE `sys\_log` (  `id` bigint(20) NOT NULL AUTO\_INCREMENT,  `username` varchar(50) DEFAULT NULL COMMENT '用户名',  `operation` varchar(50) DEFAULT NULL COMMENT '用户操作',  `method` varchar(200) DEFAULT NULL COMMENT '请求方法',  `params` varchar(5000) DEFAULT NULL COMMENT '请求参数',  `ip` varchar(64) DEFAULT NULL COMMENT 'IP地址',  `create\_date` datetime DEFAULT NULL COMMENT '创建时间',  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=14 DEFAULT CHARSET=utf8 COMMENT='系统日志';  -- ----------------------------  -- Records of sys\_log  -- ----------------------------  INSERT INTO `sys\_log` VALUES ('11', 'admin', '修改用户', 'name.yaohuan.dt.service.impl.SysUserServiceImpl.update()', '{\"createTime\":1508459924000,\"createUserId\":1,\"email\":\"2\",\"mobile\":\"2\",\"password\":\"\",\"roleIdList\":[5],\"status\":1,\"userId\":6,\"username\":\"2\"}', '0:0:0:0:0:0:0:1', '2017-10-21 05:54:57');  INSERT INTO `sys\_log` VALUES ('12', 'admin', '修改用户', 'name.yaohuan.dt.service.impl.SysUserServiceImpl.update()', '{\"createTime\":1508459924000,\"createUserId\":1,\"email\":\"2\",\"mobile\":\"2\",\"password\":\"\",\"roleIdList\":[5],\"status\":1,\"userId\":6,\"username\":\"2\"}', '0:0:0:0:0:0:0:1', '2017-10-21 05:56:09');  INSERT INTO `sys\_log` VALUES ('13', 'admin', '修改用户', 'name.yaohuan.dt.service.impl.SysUserServiceImpl.update()', '{\"createTime\":1508459924000,\"createUserId\":1,\"email\":\"2\",\"mobile\":\"2\",\"password\":\"\",\"roleIdList\":[5],\"status\":1,\"userId\":6,\"username\":\"2\"}', '0:0:0:0:0:0:0:1', '2017-10-21 10:30:56');  -- ----------------------------  -- Table structure for sys\_menu  -- ----------------------------  DROP TABLE IF EXISTS `sys\_menu`;  CREATE TABLE `sys\_menu` (  `id` bigint(20) NOT NULL AUTO\_INCREMENT,  `parent\_id` bigint(20) DEFAULT NULL COMMENT '父菜单ID，一级菜单为0',  `name` varchar(50) DEFAULT NULL COMMENT '菜单名称',  `url` varchar(200) DEFAULT NULL COMMENT '菜单URL',  `perms` varchar(500) DEFAULT NULL COMMENT '授权(多个用逗号分隔，如：user:list,user:create)',  `type` int(11) DEFAULT NULL COMMENT '类型 0：目录 1：菜单 2：按钮',  `icon` varchar(50) DEFAULT NULL COMMENT '菜单图标',  `order\_num` int(11) DEFAULT NULL COMMENT '排序',  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=42 DEFAULT CHARSET=utf8 COMMENT='菜单管理';  -- ----------------------------  -- Records of sys\_menu  -- ----------------------------  INSERT INTO `sys\_menu` VALUES ('1', '0', '系统管理', '', '', '0', 'fa fa-cog', '300');  INSERT INTO `sys\_menu` VALUES ('2', '1', '用户管理', 'sys/user/index', '', '1', 'fa fa-user', '310');  INSERT INTO `sys\_menu` VALUES ('3', '1', '角色管理', 'sys/role/index', '', '1', 'fa fa-user-secret', '320');  INSERT INTO `sys\_menu` VALUES ('4', '1', '菜单管理', 'sys/menu/index', '', '1', 'fa fa-th-list', '330');  INSERT INTO `sys\_menu` VALUES ('5', '31', 'SQL监控', 'druid/index.html', '', '1', 'fa fa-bug', '210');  INSERT INTO `sys\_menu` VALUES ('6', '30', '任务管理', 'schedule/job/index', '', '1', '', '110');  INSERT INTO `sys\_menu` VALUES ('7', '6', '查看', '', 'schedule:job:list,schedule:job:info', '2', '', '111');  INSERT INTO `sys\_menu` VALUES ('8', '6', '新增', '', 'schedule:job:save', '2', '', '112');  INSERT INTO `sys\_menu` VALUES ('9', '6', '修改', '', 'schedule:job:update', '2', '', '113');  INSERT INTO `sys\_menu` VALUES ('10', '6', '删除', '', 'schedule:job:delete', '2', '', '114');  INSERT INTO `sys\_menu` VALUES ('11', '6', '暂停', '', 'schedule:job:pause', '2', '', '115');  INSERT INTO `sys\_menu` VALUES ('12', '6', '恢复', '', 'schedule:job:resume', '2', '', '116');  INSERT INTO `sys\_menu` VALUES ('13', '6', '立即执行', '', 'schedule:job:run', '2', '', '117');  INSERT INTO `sys\_menu` VALUES ('14', '30', '任务日志', 'schedule/joblog/index', 'schedule:joblog:list', '1', '', '120');  INSERT INTO `sys\_menu` VALUES ('15', '2', '查看', '', 'sys:user:list,sys:user:info,sys:role:select', '2', '', '311');  INSERT INTO `sys\_menu` VALUES ('16', '2', '新增', '', 'sys:user:save,sys:role:select', '2', '', '312');  INSERT INTO `sys\_menu` VALUES ('17', '2', '修改', '', 'sys:user:update,sys:role:select', '2', '', '313');  INSERT INTO `sys\_menu` VALUES ('18', '2', '删除', '', 'sys:user:delete', '2', '', '314');  INSERT INTO `sys\_menu` VALUES ('19', '3', '查看', '', 'sys:role:list,sys:role:info,sys:menu:perms', '2', '', '321');  INSERT INTO `sys\_menu` VALUES ('20', '3', '新增', '', 'sys:role:save,sys:menu:perms', '2', '', '322');  INSERT INTO `sys\_menu` VALUES ('21', '3', '修改', '', 'sys:role:update,sys:menu:perms', '2', '', '323');  INSERT INTO `sys\_menu` VALUES ('22', '3', '删除', '', 'sys:role:delete', '2', '', '324');  INSERT INTO `sys\_menu` VALUES ('23', '4', '查看', '', 'sys:menu:list,sys:menu:info', '2', '', '331');  INSERT INTO `sys\_menu` VALUES ('24', '4', '新增', '', 'sys:menu:save,sys:menu:select', '2', '', '332');  INSERT INTO `sys\_menu` VALUES ('25', '4', '修改', '', 'sys:menu:update,sys:menu:select', '2', '', '333');  INSERT INTO `sys\_menu` VALUES ('26', '4', '删除', '', 'sys:menu:delete', '2', '', '334');  INSERT INTO `sys\_menu` VALUES ('27', '1', '参数管理', 'sys/config/index', 'sys:config:list,sys:config:info,sys:config:save,sys:config:update,sys:config:delete', '1', 'fa fa-sun-o', '340');  INSERT INTO `sys\_menu` VALUES ('28', '1', '代码生成器', 'sys/generator.html', 'sys:generator:list,sys:generator:code', '1', 'fa fa-rocket', '349');  INSERT INTO `sys\_menu` VALUES ('29', '31', '系统日志', 'sys/log/index', 'sys:log:list', '1', 'fa fa-file-text-o', '220');  INSERT INTO `sys\_menu` VALUES ('30', '0', '定时任务', '', '', '0', 'fa fa-clock-o', '100');  INSERT INTO `sys\_menu` VALUES ('31', '0', '系统监控', '', '', '0', 'fa fa-binoculars', '200');  INSERT INTO `sys\_menu` VALUES ('36', '0', 'test1', 'url', '123', '1', 'fa fa-add', '1');  INSERT INTO `sys\_menu` VALUES ('37', '6', '22', null, '22', '2', null, '0');  INSERT INTO `sys\_menu` VALUES ('38', '0', '33', null, null, '0', '333', '3');  INSERT INTO `sys\_menu` VALUES ('39', '36', '55', '55', '55', '1', '5', '5');  INSERT INTO `sys\_menu` VALUES ('40', '36', '66', '66`', '66', '1', '66', '66');  INSERT INTO `sys\_menu` VALUES ('41', '31', '7788', '7788', '7788', '0', '7788', '7788');  -- ----------------------------  -- Table structure for sys\_role  -- ----------------------------  DROP TABLE IF EXISTS `sys\_role`;  CREATE TABLE `sys\_role` (  `id` bigint(20) NOT NULL AUTO\_INCREMENT,  `role\_name` varchar(100) DEFAULT NULL COMMENT '角色名称',  `remark` varchar(100) DEFAULT NULL COMMENT '备注',  `create\_user\_id` bigint(20) DEFAULT NULL COMMENT '创建者ID',  `create\_time` datetime DEFAULT NULL COMMENT '创建时间',  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=7 DEFAULT CHARSET=utf8 COMMENT='角色';  -- ----------------------------  -- Records of sys\_role  -- ----------------------------  INSERT INTO `sys\_role` VALUES ('5', '系统管理员', '', '1', '2017-10-15 18:05:57');  INSERT INTO `sys\_role` VALUES ('6', '测试角色', '', '1', '2017-10-17 05:32:10');  -- ----------------------------  -- Table structure for sys\_role\_menu  -- ----------------------------  DROP TABLE IF EXISTS `sys\_role\_menu`;  CREATE TABLE `sys\_role\_menu` (  `id` bigint(20) NOT NULL AUTO\_INCREMENT,  `role\_id` bigint(20) DEFAULT NULL COMMENT '角色ID',  `menu\_id` bigint(20) DEFAULT NULL COMMENT '菜单ID',  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=230 DEFAULT CHARSET=utf8 COMMENT='角色与菜单对应关系';  -- ----------------------------  -- Records of sys\_role\_menu  -- ----------------------------  INSERT INTO `sys\_role\_menu` VALUES ('190', '5', '1');  INSERT INTO `sys\_role\_menu` VALUES ('191', '5', '2');  INSERT INTO `sys\_role\_menu` VALUES ('192', '5', '15');  INSERT INTO `sys\_role\_menu` VALUES ('193', '5', '16');  INSERT INTO `sys\_role\_menu` VALUES ('194', '5', '17');  INSERT INTO `sys\_role\_menu` VALUES ('195', '5', '18');  INSERT INTO `sys\_role\_menu` VALUES ('196', '5', '3');  INSERT INTO `sys\_role\_menu` VALUES ('197', '5', '19');  INSERT INTO `sys\_role\_menu` VALUES ('198', '5', '20');  INSERT INTO `sys\_role\_menu` VALUES ('199', '5', '21');  INSERT INTO `sys\_role\_menu` VALUES ('200', '5', '22');  INSERT INTO `sys\_role\_menu` VALUES ('201', '5', '4');  INSERT INTO `sys\_role\_menu` VALUES ('202', '5', '23');  INSERT INTO `sys\_role\_menu` VALUES ('203', '5', '24');  INSERT INTO `sys\_role\_menu` VALUES ('204', '5', '25');  INSERT INTO `sys\_role\_menu` VALUES ('205', '5', '26');  INSERT INTO `sys\_role\_menu` VALUES ('206', '5', '5');  INSERT INTO `sys\_role\_menu` VALUES ('207', '5', '6');  INSERT INTO `sys\_role\_menu` VALUES ('208', '5', '7');  INSERT INTO `sys\_role\_menu` VALUES ('209', '5', '8');  INSERT INTO `sys\_role\_menu` VALUES ('210', '5', '9');  INSERT INTO `sys\_role\_menu` VALUES ('211', '5', '10');  INSERT INTO `sys\_role\_menu` VALUES ('212', '5', '11');  INSERT INTO `sys\_role\_menu` VALUES ('213', '5', '12');  INSERT INTO `sys\_role\_menu` VALUES ('214', '5', '13');  INSERT INTO `sys\_role\_menu` VALUES ('215', '5', '14');  INSERT INTO `sys\_role\_menu` VALUES ('216', '5', '27');  INSERT INTO `sys\_role\_menu` VALUES ('217', '5', '29');  INSERT INTO `sys\_role\_menu` VALUES ('218', '5', '28');  INSERT INTO `sys\_role\_menu` VALUES ('225', '6', '1');  INSERT INTO `sys\_role\_menu` VALUES ('226', '6', '2');  INSERT INTO `sys\_role\_menu` VALUES ('227', '6', '15');  INSERT INTO `sys\_role\_menu` VALUES ('228', '6', '17');  INSERT INTO `sys\_role\_menu` VALUES ('229', '6', '18');  -- ----------------------------  -- Table structure for sys\_user  -- ----------------------------  DROP TABLE IF EXISTS `sys\_user`;  CREATE TABLE `sys\_user` (  `id` bigint(20) NOT NULL AUTO\_INCREMENT,  `username` varchar(50) NOT NULL COMMENT '用户名',  `password` varchar(100) DEFAULT NULL COMMENT '密码',  `email` varchar(100) DEFAULT NULL COMMENT '邮箱',  `mobile` varchar(100) DEFAULT NULL COMMENT '手机号',  `status` tinyint(4) DEFAULT NULL COMMENT '状态 0：禁用 1：正常',  `create\_user\_id` bigint(20) DEFAULT NULL COMMENT '创建者ID',  `create\_time` datetime DEFAULT NULL COMMENT '创建时间',  PRIMARY KEY (`id`),  UNIQUE KEY `username` (`username`)  ) ENGINE=InnoDB AUTO\_INCREMENT=5 DEFAULT CHARSET=utf8 COMMENT='系统用户';  -- ----------------------------  -- Records of sys\_user  -- ----------------------------  INSERT INTO `sys\_user` VALUES ('1', 'admin', '8b64db1b8cb9f9c2b2ae41c65b7f2c4b1456f68dd1235b8527234fae5e40bce5', '100@qq.com', '13666666666', '1', null, '2017-06-01 15:33:20');  INSERT INTO `sys\_user` VALUES ('4', 'helen', '5aa0f441849a94bd1f18852df531919d0024f78affbf37c229d78820bbbae226', '55317332@qq.com', '13766816630', '1', '1', '2017-10-15 14:54:49');  -- ----------------------------  -- Table structure for sys\_user\_role  -- ----------------------------  DROP TABLE IF EXISTS `sys\_user\_role`;  CREATE TABLE `sys\_user\_role` (  `id` bigint(20) NOT NULL AUTO\_INCREMENT,  `user\_id` bigint(20) DEFAULT NULL COMMENT '用户ID',  `role\_id` bigint(20) DEFAULT NULL COMMENT '角色ID',  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=65 DEFAULT CHARSET=utf8 COMMENT='用户与角色对应关系';  -- ----------------------------  -- Records of sys\_user\_role  -- ----------------------------  INSERT INTO `sys\_user\_role` VALUES ('36', '1', '5');  INSERT INTO `sys\_user\_role` VALUES ('37', '1', '6');  INSERT INTO `sys\_user\_role` VALUES ('38', '4', '6');  INSERT INTO `sys\_user\_role` VALUES ('39', '5', '5');  INSERT INTO `sys\_user\_role` VALUES ('64', '6', '5'); |

# 第六章 思想问题

# 1电商项目

## 1.1购物车思路

## 1.2 freemarker思路

1.为什么使用网页静态化技术：

对于电商网站的商品详细页来说，至少几百万个商品，每个商品又有大量的信息，这样的情况同样也适用于使用网页静态化来解决。

网页静态化技术和缓存技术的共同点都是为了减轻数据库的访问压力，但是具体的应用场景不同，缓存比较适合小规模的数据，而网页静态化比较适合大规模且相对变化不太频繁的数据。另外网页静态化还有利于SEO。

另外我们如果将网页以纯静态化的形式展现，就可以使用Nginx这样的高性能的web服务器来部署。Nginx可以承载5万的并发，而Tomcat只有几百。

2.freemarker的使用：

两个必须的条件：ftl模板+java数据的注入，ftl模板中可能会使用${message}，这样你的java数据需要注入message的值

3.生成文件使用步骤：

第一步：创建一个 Configuration 对象，直接 new 一个对象。构造方法的参数就是 freemarker的版本号。

第二步：设置模板文件所在的路径。

第三步：设置模板文件使用的字符集。一般就是 utf-8.

第四步：加载一个模板，创建一个模板对象。

第五步：创建一个模板使用的数据集，可以是 pojo 也可以是 map。一般是 Map。

第六步：创建一个 Writer 对象，一般创建一 FileWriter 对象，指定生成的文件名。

第七步：调用模板对象的 process 方法输出文件。

第八步：关闭流。

4.代码案例：

|  |
| --- |
| public void test1() throws IOException, TemplateException {  //1创建对象  Configuration configuration=new Configuration(Configuration.getVersion());  //指定模板所在的目录,这个目录下面可能会有很多模板  configuration.setDirectoryForTemplateLoading(new File("C:\\Users\\Administrator.PC-20180226XFFD\\Desktop\\TitanicWang\\BaseProject\\FreeMarkerProject\\freemarkerproject\\src\\main\\resources"));  configuration.setDefaultEncoding("UTF-8");  //获取具体要操作的模板  Template template = configuration.getTemplate("hello.ftl");  //设置数据  Map<String, Object> model = new HashMap<>();  model.put("hello","知名企业家冤狱11年,现像国家索赔20亿人民币");  User user=new User();  user.setName("婷姐");  user.setPassword("0");  model.put("user", user);  List<User> users = new ArrayList<>();  for (int i = 1; i < 11; i++) {  user = new User();  user.setPassword("密码:--->" + i);  user.setName("婷姐:---->"+i);  users.add(user);  }  model.put("users", users);  Date date = new Date();  model.put("fadsfadsadfs", date);  model.put("nulldata", null);//此处null代表我们实际获取到的数据是空的  //限高//黑名单  //写成文件  //Writer writer = new FileWriter("C:\\Users\\Administrator.PC-20180226XFFD\\Desktop\\wgw\\123.html");  String path = "C:\\Users\\Administrator.PC-20180226XFFD\\Desktop\\wgw\\123.html";  Writer writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(path), "UTF-8"));  template.process(model,writer);  writer.close();  } |

## 1.3 solr

### 1.3.1为什么要用solr，IK Analyzer，配置域

（1）为什么要用solr？

大多数搜索引擎应用都必须具有某种搜索功能，问题是搜索功能往往是巨大的资源消耗并且它们由于沉重的数据库加载而拖垮你的应用的性能。

这就是为什么转移负载到一个外部的搜索服务器是一个不错的主意，Apache Solr是一个流行的开源搜索服务器，它通过使用类似REST的HTTP API，这就确保你能从几乎任何编程语言来使用solr。

Solr是一个开源搜索平台，用于构建搜索应用程序。 它建立在Lucene(全文搜索引擎)之上。 Solr是企业级的，快速的和高度可扩展的。 使用Solr构建的应用程序非常复杂，可提供高性能。

Solr可以和Hadoop一起使用。由于Hadoop处理大量数据，Solr帮助我们从这么大的源中找到所需的信息。不仅限于搜索，Solr也可以用于存储目的。像其他NoSQL数据库一样，它是一种非关系数据存储和处理技术。

总之，Solr是一个可扩展的，可部署，搜索/存储引擎，优化搜索大量以文本为中心的数据。

个人分析总结：solr是跨平台的，因为任何应用环境只要通过http就可以访问solr，实现结果的显示。

（2） IK Analyzer

IK Analyzer的配置：在 solr 工程的 WEB-INF/classes 目录下找到schema.xml 文件，然后修改 Solrhome 的 schema.xml 文件，配置一个 FieldType，使用 IKAnalyzer

|  |
| --- |
| <fieldType name="text\_ik" class="solr.TextField">  <analyzer class="org.wltea.analyzer.lucene.IKAnalyzer"/>  </fieldType> |

（3）配置域（静态）

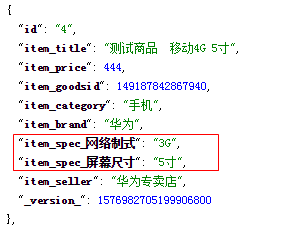
域相当于数据库的表字段，用户存放数据，因此用户根据业务需要去定义相关的Field（域），一般来说，每一种对应着一种数据，用户对同一种数据进行相同的操作。域的配置是在schema.xml 文件中配置，例如：

|  |
| --- |
| <field name="item\_goodsid" type="long" indexed="true" stored="true"/>  <field name="item\_title" type="text\_ik" indexed="true" stored="true"/> |

域的常用属性：name：指定域的名称type：指定域的类型 indexed：是否索引stored：是否存储 required：是否必须 multiValued：是否多值

（4）动态配置域

当我们需要动态扩充字段时，我们需要使用动态域。对于品优购，规格的值是不确定的，所以我们需要使用动态域来实现。需要实现的效果如下：



配置：

|  |
| --- |
| <dynamicField name="item\_spec\_\*" type="string" indexed="true" stored="true" /> |

### 1.3.2使用solr的步骤

（1）在src/main/resources下创建 applicationContext-solr.xml

|  |
| --- |
| <!-- solr服务器地址 -->  <solr:solr-server id="solrServer" url="http://192.168.200.128:8080/solr" />  <!-- solr模板，使用solr模板可对索引库进行CRUD的操作 -->  <bean id="solrTemplate" class="org.springframework.data.solr.core.SolrTemplate">  <constructor-arg ref="solrServer" />  </bean> |

（2）创建你需要存到solr的实体类

创建 cn.itcast.pojo 包，将品优购的TbItem实体类拷入本工程 ,属性使用@Field注解标识 。如果属性与配置文件定义的域名称不一致，需要在注解中指定域名称。

在你操作这个类的时候，属性被赋的值就会传递到solr的对应的域的属性里

|  |
| --- |
| public class TbItem implements Serializable{  @Field  private Long id;  @Field("item\_title")  private String title;    @Field("item\_price")  private BigDecimal price;  @Field("item\_image")  private String image;  @Field("item\_goodsid")  private Long goodsId;  @Field("item\_category")  private String category;  @Field("item\_brand")  private String brand;  @Field("item\_seller")  private String seller;  .......  } |

（3）然后按照平时操作mysql数据库的套路操作solr即可

只是多了创建solr模板自动注入的语句。

增加（修改）

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.class)  @ContextConfiguration(locations="classpath:applicationContext-solr.xml")  public class TestTemplate {  @Autowired  private SolrTemplate solrTemplate;    @Test  public void testAdd(){  TbItem item=new TbItem();  item.setId(1L);  item.setBrand("华为");  item.setCategory("手机");  item.setGoodsId(1L);  item.setSeller("华为2号专卖店");  item.setTitle("华为Mate9");  item.setPrice(new BigDecimal(2000));  solrTemplate.saveBean(item);  solrTemplate.commit();  }  } |

查询

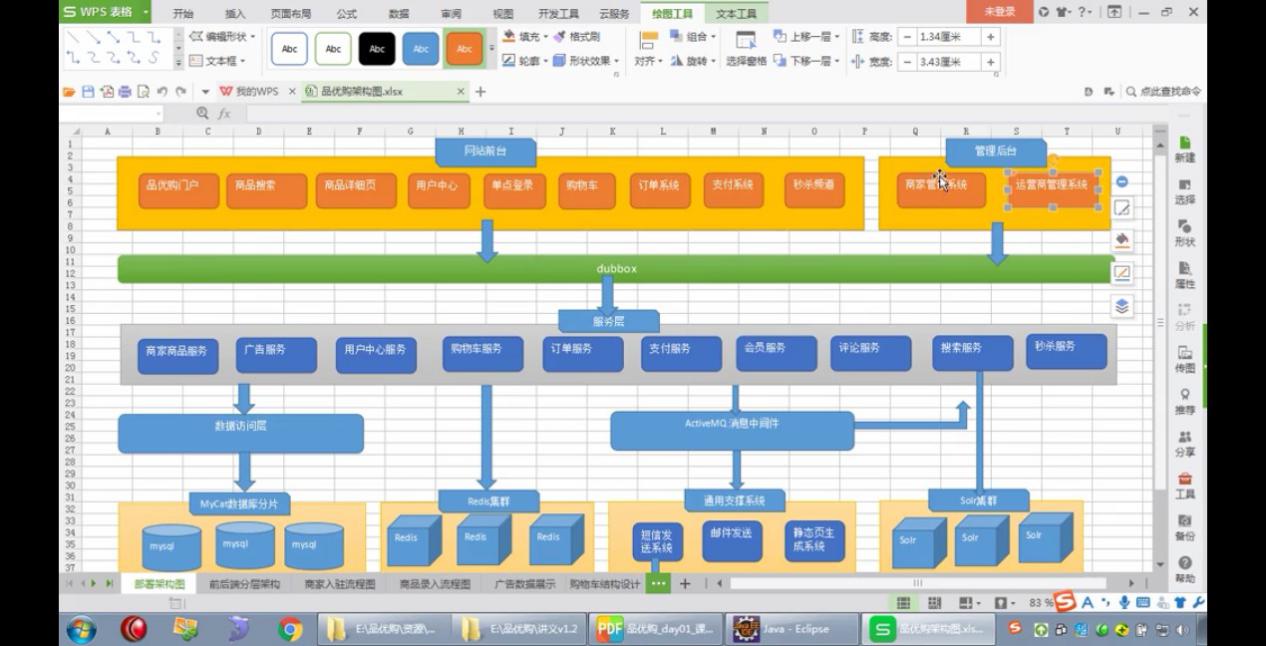
|  |
| --- |
| @Test  public void testFindOne(){  TbItem item = solrTemplate.getById(1, TbItem.class);  System.out.println(item.getTitle());  } |

按主键删除

|  |
| --- |
| public void testDelete(){  solrTemplate.deleteById("1");  solrTemplate.commit();  } |

## 1.4整体分析

### 1.4.1电商系统总体架构



# 2杂类

## 2.1线程和锁

### 2.1.1线程

#### 2.1.1.1线程的基本实现

线程实现共有三种方式：继承Thread，实现接口Runnable，实现接口Callable;

如下面的程序，继承Thread时，new的是线程子类的对象（ MyThread）并赋值名字，此时new一次对象走一次线程子类，所以每个线程各不相关

|  |
| --- |
| public class MyThread extends Thread{//继承Thread类  public MyThread(String name) {//若不通过构造函数赋名字可以通过在主函数中t1.setName("");赋名  super(name);  }  public void run() {//重写run方法  for(int i=0;i<50;i++){//Thread.currentThread().getName()功能是获取当前线程名称  System.out.println("子线程在执行..."+i+"....."+Thread.currentThread().getName());  }  }  }  /\*测试函数\*/  public class Demo1 {  public static void main(String[] args) {  MyThread t1=new MyThread("线程1");//创建线程对象并通过构造方法给线程附名字  MyThread t2=new MyThread("线程2");  //3启动了一个子线程  t1.start();//start（）方法会自动执行run方法的方法体  t2.start();//线程的执行是抢占式，所以t2并不一定非要等t1执行完之后才执行  for(int i=0;i<100;i++){//执行main方法的线程叫主线程  System.out.println("主线程执行了\*\*\*\*\*\*"+i);  }  }  } |

如下面的程序，实现接口Runnable，new的是Thread对象，同时可以赋值线程名字和共享资源，此时线程子类TicketRes只new了一次，所以所有线程会共享TicketRes类中的资源。

|  |
| --- |
| public class TicketRes implements Runnable{//实现Runnable接口  private int ticket=100;//票资源共享  public void run() {  while(true){//问题在于，线程是抢占式的，当t1抢到cpu.开始卖票，打印第100张票，没有执行ticket--，时间到了，释放cpu，那么t2抢到cpu开始卖票，此时ticket仍然是100，那么t2也就会打印第100张票，这样程序就出错了，所以以此类推此程序会出现四个窗口同时卖100张票或者有的窗口卖的还是负数票，解决方法：见多线程同步章节  if(ticket<1){  break;  }  System.out.println(Thread.currentThread().getName()+"卖了第"+ticket+"张票");  ticket--;  }  }  }  public class Demo2 {  public static void main(String[] args) {  TicketRes res=new TicketRes();//1创建TicketRes  Thread w1=new Thread(res);//2创建线程  Thread w2=new Thread(res);  Thread w3=new Thread(res);  Thread w4=new Thread(res);  //3启动  w1.start();  w2.start();  w3.start();  w4.start();  }  } |

#### 4.4.1.2 线程的运行和小配置

（1）线程运行：主线程中调用start函数开启线程，进入线程类执行run方法

中间可能遇到stop函数终止线程，可能遇见sleep休眠

（2）线程可以配置优先级，可以配置后台线程，可以并入线程

### 2.1.2锁

#### 2.1.2.1锁的实现和死锁

（1）锁实现的四种基本方法：同步代码块synchronized（lock）（lock是一个定义的私有的Object对象），静态方法同步代码块，非静态方法同步代码块，可重入锁（显式）

后面两者区别在于静态方法使用类.class当锁，而非静态方法使用this当锁。

同步代码块可以直接写在Run函数中，而其他两个可以在Run函数中调用该方法，线程执行体必须写在锁所在的方法下面，要不然就没意义了。

（2）显式锁

这里介绍的显式锁即可重入锁ReentrantLock，由下面的程序可以看出，打开锁的时候是调用lock.unlock();，而之前的几个锁是线程执行完自动释放锁。

|  |
| --- |
| public class TicketRes implements Runnable{  private int ticket=100;  //可重入锁  private ReentrantLock lock=new ReentrantLock();  public void run() {  while(true){  lock.lock();//上锁  try {  if (ticket < 1) {  break;  }  System.out.println(Thread.currentThread().getName() + "卖第" + ticket + "张票");  ticket--;  } finally {  lock.unlock();  }  }  }  } |

（3）死锁条件：（1）两个以上的线程 （2）至少两个锁以上 （3）同步中嵌套同步

#### 2.1.2.2线程、锁在单例中的使用

懒汉式的单例是线程不安全的，所以在多线程中使用懒汉式单例时需要在单例对象类中判断对象是否存在之前添加锁。

## 2.2shiro



# 3 spring家族

## 3.1spring

### 3.1.1spring的原理

Spring运作的核心依靠Ioc和Aop

Ioc控制反转通过java反射机制将真实类对象的创建权转交给spring容器，spring容器会根据xml文件bean的定义创建对应类的对象并建立相应的实例依赖关系。

Aop面向切面编程则会实现spring在指定切入点执行需要补充的业务和方法

### 3.1.2aop

#### 3.1.2.1aop的执行流程

定义好通知类和目标类，通知类中定义好before，around等五个通知时间的需要执行的方法，然后在配置类中定义好通知对象和目标对象，还要定义好通知的切入点（全限定路径名称），然后将通知类的定义的方法配置到xml文件中对应的通知时间属性位置，然后执行方法就可以了。

# 9999下拉

Cloud boot 工具集不是框架