数据库表的设计：

投票表VOTE为中间表，它主要是关联谁关联了哪些问题哪个选项，最好在中间表也加上创建时间，可以记录关联的时间。

逻辑表user\_topic主要是关联了谁投了哪个话题，这样如果不在这边的用户，就表明他还没有投过票。

数据库SQL表的编写：

诀窍是从核心子查询语句开始编写，像剥洋葱一样进行sql剥离，最后再一层一层添加，例如查询投票结果集：

SELECT

d.ID as QUESTION\_ID,

d.`NAME` AS question\_name,

c.`NAME` AS `option\_name`,

b.vote\_num

FROM

`option` AS c

LEFT JOIN (

SELECT

a.OPTION\_ID,

SUM(

CASE a.WEIGHT

WHEN NULL THEN

0

ELSE

a.WEIGHT

END

) AS vote\_num

FROM

vote AS a

GROUP BY

a.OPTION\_ID

) AS b ON c.ID = b.OPTION\_ID

LEFT JOIN question AS d ON c.QUESTION\_ID = d.id

LEFT JOIN topic AS e ON e.id = d.TOPIC\_ID

WHERE

e.is\_valid = 1

ORDER BY d.ID desc,b.vote\_num desc

SQL查询结果的映射：

定义好VO，VO最好都继承自Domain，减少代码；

<resultMap id=*"VoteResultMap"* type=*"com.zhiye.bhmall.modules.vote.vo.VoteResultVo"*>

<result column=*"question\_name"* jdbcType=*"VARCHAR"* property=*"questionName"* />

<result column=*"option\_name"* jdbcType=*"VARCHAR"* property=*"optionName"* />

<result column=*"vote\_num"* jdbcType=*"INTEGER"* property=*"voteNum"* />

</resultMap>

<select id=*"selectVoteResult"* resultMap=*"VoteResultMap"*>

SELECT

d.ID as QUESTION\_ID,

d.`NAME` AS question\_name,

c.`NAME` AS `option\_name`,

b.vote\_num

FROM

`option` AS c

LEFT JOIN (

SELECT

a.OPTION\_ID,

SUM(

CASE a.WEIGHT

WHEN NULL THEN

0

ELSE

a.WEIGHT

END

) AS vote\_num

FROM

vote AS a

GROUP BY

a.OPTION\_ID

) AS b ON c.ID = b.OPTION\_ID

LEFT JOIN question AS d ON c.QUESTION\_ID = d.id

LEFT JOIN topic AS e ON e.id = d.TOPIC\_ID

WHERE

e.is\_valid = 1

ORDER BY d.ID desc,b.vote\_num desc

</select>

程序：

在属性文件加入下面语句可以使得改动程序以后自动重启：

spring.devtools.restart.enabled=true

spring.devtools.restart.additional-paths=src/main/java

pom.xml要加入依赖：

<dependency>

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

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

<scope>runtime</scope>

<optional>true</optional>

</dependency>

方法在对外暴露调用的时候，尤其是web app端要进行跨域配置，之前没有加入跨域，使得文杰调用服务端方法总是不成功，加入配置类是：

@Configuration

**public** **class** WebConfig **extends** WebMvcConfigurerAdapter {

@Override

**public** **void** addCorsMappings(CorsRegistry registry) {

// 允许跨域访问。

registry.addMapping("/\*\*")

.allowedOrigins("\*")

.allowedMethods("GET", "HEAD", "POST", "PUT", "PATCH", "DELETE", "OPTIONS", "TRACE")

.allowCredentials(**true**); // 加入这个使得可以传cookie保证session一致性。

}

}

加入日期转换配置文件，可以格式系统输出日期，如下所示：

@Configuration

**public** **class** DateConfig {

@Bean

**public** MappingJackson2HttpMessageConverter getMappingJackson2HttpMessageConverter() {

MappingJackson2HttpMessageConverter mappingJackson2HttpMessageConverter = **new** MappingJackson2HttpMessageConverter();

//设置日期格式

ObjectMapper objectMapper = **new** ObjectMapper();

SimpleDateFormat smt = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

objectMapper.setDateFormat(smt);

mappingJackson2HttpMessageConverter.setObjectMapper(objectMapper);

//设置中文编码格式

List<MediaType> list = **new** ArrayList<MediaType>();

list.add(MediaType.***APPLICATION\_JSON\_UTF8***);

mappingJackson2HttpMessageConverter.setSupportedMediaTypes(list);

**return** mappingJackson2HttpMessageConverter;

}

}

生成验证码：

1.在pom.xml文件当中要先加入依赖，如下

<!-- 生成验证码-->

<dependency>

<groupId>com.github.axet</groupId>

<artifactId>kaptcha</artifactId>

<version>0.0.9</version>

</dependency>

2.要加入配置文件类：

/\*\*

\* 生成验证码配置

\*/

@Configuration

**public** **class** KaptchaConfig {

@Bean

**public** DefaultKaptcha producer() {

Properties properties = **new** Properties();

properties.put("kaptcha.border", "no");

properties.put("kaptcha.textproducer.font.color", "black");

properties.put("kaptcha.textproducer.char.space", "5");

Config config = **new** Config(properties);

DefaultKaptcha defaultKaptcha = **new** DefaultKaptcha();

defaultKaptcha.setConfig(config);

**return** defaultKaptcha;

}

}

3.如何使用：

@Autowired

**private** Producer producer;

@ApiOperation(value = "createImg", notes = "")

@RequestMapping(value = "createImg", method = RequestMethod.***GET***)

**public** **void** captcha(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

response.setHeader("Cache-Control", "no-store, no-cache");

response.setContentType("image/jpeg");

// 生成文字验证码

String text = producer.createText();

// 将验证码保存至redis key为sessionId

// if (redisUtil != null) {

// String key = IMG\_REDIS\_KEY + request.getSession().getId();

// redisUtil.set(key, text, 120);

// }

// 生成图片验证码

BufferedImage image = producer.createImage(text);

ServletOutputStream out = response.getOutputStream();

ImageIO.*write*(image, "jpg", out);

IOUtils.*closeQuietly*(out);

}

缓存ehcache整合springboot使用：

1.添加依赖：

<dependency>

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

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

</dependency>

<dependency>

<groupId>net.sf.ehcache</groupId>

<artifactId>ehcache</artifactId>

</dependency>

2.加入配置文件

**import** org.springframework.cache.annotation.EnableCaching;

**import** org.springframework.cache.ehcache.EhCacheCacheManager;

**import** org.springframework.cache.ehcache.EhCacheManagerFactoryBean;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.core.io.ClassPathResource;

@Configuration

@EnableCaching

**public** **class** CacheConfiguration {

@Bean

**public** EhCacheCacheManager ehCacheCacheManager(EhCacheManagerFactoryBean bean) {

**return** **new** EhCacheCacheManager(bean.getObject());

}

@Bean

**public** EhCacheManagerFactoryBean ehCacheManagerFactoryBean() {

EhCacheManagerFactoryBean cacheManagerFactoryBean = **new** EhCacheManagerFactoryBean();

cacheManagerFactoryBean.setConfigLocation(**new** ClassPathResource("ehcache.xml"));

cacheManagerFactoryBean.setShared(**true**);

**return** cacheManagerFactoryBean;

}

}

3.在类路径加入配置文件：

ehcache.xml

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

<ehcache>

<cache name=*"verifyCodeCache"* maxElementsInMemory=*"1000"*/>

</ehcache>

4.使用：

**import** org.springframework.cache.Cache;

**import** org.springframework.cache.CacheManager;

@Autowired

CacheManager cacheManager;

Cache cache = cacheManager.getCache("verifyCodeCache");

String text = (String) httpSession.getAttribute(***IMG\_REDIS\_KEY***+mobil);

text = cache.get(***IMG\_REDIS\_KEY***+mobil,String.**class**);

异常处理：

[在serviceimpl服务实现类上面加上@Transactional(rollbackFor=Exception.**class**)](mailto:在serviceimpl服务实现类上面加上@Transactional(rollbackFor=Exception.class))使得不仅仅可以回滚RuntimeException。

日志打印，根路径加上logback-spring.xml(记得修改里面的扫描类)：

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

<configuration debug=*"false"*>

<!--定义日志文件的存储地址 勿在 LogBack 的配置中使用相对路径 -->

<property name=*"LOG\_HOME"* value=*"../logs/springBootLog-system"* />

<!-- <property name="LOG\_HOME" value="${LOG\_PATH}" /> -->

<!-- 控制台输出 -->

<appender name=*"STDOUT"* class=*"ch.qos.logback.core.ConsoleAppender"*>

<encoder class=*"ch.qos.logback.classic.encoder.PatternLayoutEncoder"*>

<!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度%msg：日志消息，%n是换行符 -->

<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n</pattern>

<charset>UTF-8</charset>

</encoder>

<param name=*"Encoding"* value=*"UTF-8"* />

</appender>

<!-- 按照每天生成日志文件 -->

<appender name=*"FILE"*

class=*"ch.qos.logback.core.rolling.RollingFileAppender"*>

<rollingPolicy class=*"ch.qos.logback.core.rolling.TimeBasedRollingPolicy"*>

<!--日志文件输出的文件名 -->

<FileNamePattern>${LOG\_HOME}/vote-%d{yyyy-MM-dd}.log

</FileNamePattern>

<!--日志文件保留天数 -->

<MaxHistory>30</MaxHistory>

</rollingPolicy>

<encoder class=*"ch.qos.logback.classic.encoder.PatternLayoutEncoder"*>

<!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度%msg：日志消息，%n是换行符 -->

<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} -

%msg%n</pattern>

<charset>UTF-8</charset>

</encoder>

<param name=*"Encoding"* value=*"UTF-8"* />

<!--日志文件最大的大小 -->

<triggeringPolicy

class=*"ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy"*>

<MaxFileSize>100MB</MaxFileSize>

</triggeringPolicy>

</appender>

<!-- 日志输出级别 -->

<root level=*"INFO"*>

<appender-ref ref=*"FILE"* />

<!-- 生产环境将请stdout,testfile去掉 -->

<appender-ref ref=*"STDOUT"* />

</root>

<!-- <root level="DEBUG">

<appender-ref ref="STDOUT" />

</root> -->

<logger name=*"com.zhiye.bhmall.modules.vote.mapper"* level=*"debug"* additivity=*"false"*>

<appender-ref ref=*"STDOUT"* />

</logger>

</configuration>

暴露给前端调用的方法的时候，如果用RequestBody的话，那都会当成字符串进行解析，这样其它的参数用拼写的模式，url要写成：http://xxx? userId=1& topicId=1，该例子同样也解决传递数组接收数组的问题，例如：

@ApiOperation(value = "投票", notes = "")

@PostMapping(value = "insert")

@ResponseBody

**public** Response insert(@RequestBody List<Vote> votes, @RequestParam(value = "userId") Integer userId,

@RequestParam(value = "topicId") Integer topicId) {

Response resp = **new** Response();

**boolean** isVoted = userService.checkVote(topicId, userId);

**if** (isVoted) {

resp.setCode("400");

resp.setMessage("本次投票只允许投一次");

**return** resp;

}

**int** res = **this**.voteService.insert(votes);

**if** (res > 0) {

UserTopic userTopic = **new** UserTopic();

userTopic.setTopicId(topicId);

userTopic.setUserId(userId);

userTopicService.insert(userTopic);

}

**return** resp;

}