## 1.跨域问题

跨域问题实在前后端分离开发时常见问题，出于安全原因，浏览器禁止ajax调用驻留在当前网点外的资源。

### [1.1@CrossOrigin](mailto:1.1@CrossOrigin)

该注解是一种相对于简单的实现方式，应用在方法或控制类的注解，使用路径注解的方法。

@CrossOrion有两个参数，origins：允许可访问的域列表，maxAge准备响应前的缓存持续最大时间。

### 1.2使用WebMvcConfigurer进行配置

@Configuration  
public class MvcConfig {  
 @Bean  
 WebMvcConfigurer configurer() {  
 return new WebMvcConfigurer() {  
 @Override  
 public void addCorsMappings(CorsRegistry registry) {  
 registry  
 .addMapping("/")  
 .allowedOrigins("\*")  
 .allowedHeaders("\*")  
 .allowCredentials(false)  
 .allowedMethods("\*");  
 }  
 }  
 }  
}

### 1.3通过CorsFilter进行配置

@Configuration  
public class CorsConfig {  
  
 private CorsConfiguration buildConfig() {  
 CorsConfiguration corsConfiguration = new CorsConfiguration();  
 corsConfiguration.addAllowedOrigin(CorsConfiguration.*ALL*);  
 corsConfiguration.addAllowedHeader(CorsConfiguration.*ALL*);  
 corsConfiguration.addAllowedMethod(CorsConfiguration.*ALL*);  
 corsConfiguration.setAllowCredentials(true);  
 return corsConfiguration;  
 }  
  
 @Bean  
 public CorsFilter corsFilter() {  
 UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();  
 source.registerCorsConfiguration("/\*\*", buildConfig());  
 return new CorsFilter(source);  
 }  
}

### 1.4通过SpringSecurity进行

适用于多个SpringSecurity配置并行使用

@Configuration  
@ConditionalOnProperty  
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {  
 @Autowired private UrlBasedCorsConfigurationSource corsConfigurationSource;  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http  
 .cors()  
 .configurationSource(corsConfigurationSource);  
 }  
  
}

@Configuration  
public class CorsConfig {  
  
 private CorsConfiguration buildConfig() {  
 CorsConfiguration corsConfiguration = new CorsConfiguration();  
 corsConfiguration.addAllowedOrigin(CorsConfiguration.*ALL*);  
 corsConfiguration.addAllowedHeader(CorsConfiguration.*ALL*);  
 corsConfiguration.addAllowedMethod(CorsConfiguration.*ALL*);  
 corsConfiguration.setAllowCredentials(true);  
 return corsConfiguration;  
 }  
  
 @Bean  
 public UrlBasedCorsConfigurationSource corsFilter() {  
 UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();  
 source.registerCorsConfiguration("/\*\*", buildConfig());  
 return source;  
 }  
}

## 2.JPA查询

Jpa查询主要存在两大类七小类。

两大类分为：依托于repository的查询实现，依托于EntityManager的查询实现

七小类分别为

1. 依托于repository通过方法名进行查询

②依托于repository通过@Query注解实现原生sql查询

③依托于repository通过@Query注解实现jpql查询

④依托于repository通过查询拼接控件拼接查询条件进行查询

⑤依托于EntityManager通过jpql进行查询

⑥依托于EntityManager通过原生sql进行查询

⑦依托于EntityManager通过查询拼接控件拼接查询条件进行查询

### 2.1依托于repository通过方法名进行查询

依托于repository通过方法名进行查询就是根据方法名来自动生成SQL，主要的语法是findXXBy,readAXXBy,queryXXBy,countXXBy, getXXBy后面跟属性名称，也使用一些加一些关键字And、 Or，修改、删除、统计也是类似语法，基本上SQL体系中的关键词都可以使用，例如：LIKE、 IgnoreCase、 OrderBy。

具体的关键字，使用方法和生产成SQL如下表所示

|  |  |  |
| --- | --- | --- |
| Keyword | Sample | JPQL snippet |
| And | findByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |
| Is,Equals | findByFirstnameIs,findByFirstnameEquals | … where x.firstname = ?1 |
| Between | findByStartDateBetween | … where x.startDate between ?1 and ?2 |
| LessThan | findByAgeLessThan | … where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | … where x.age ⇐ ?1 |
| GreaterThan | findByAgeGreaterThan | … where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | … where x.age >= ?1 |
| After | findByStartDateAfter | … where x.startDate > ?1 |
| Before | findByStartDateBefore | … where x.startDate < ?1 |
| IsNull | findByAgeIsNull | … where x.age is null |
| IsNotNull,NotNull | findByAge(Is)NotNull | … where x.age not null |
| Like | findByFirstnameLike | … where x.firstname like ?1 |
| NotLike | findByFirstnameNotLike | … where x.firstname not like ?1 |
| StartingWith | findByFirstnameStartingWith | … where x.firstname like ?1 (parameter bound with appended %) |
| EndingWith | findByFirstnameEndingWith | … where x.firstname like ?1 (parameter bound with prepended %) |
| Containing | findByFirstnameContaining | … where x.firstname like ?1 (parameter bound wrapped in %) |
| OrderBy | findByAgeOrderByLastnameDesc | … where x.age = ?1 order by x.lastname desc |
| Not | findByLastnameNot | … where x.lastname <> ?1 |
| In | findByAgeIn(Collection ages) | … where x.age in ?1 |
| NotIn | findByAgeNotIn(Collectionage) | … where x.age not in ?1 |
| TRUE | findByActiveTrue() | … where x.active = true |
| FALSE | findByActiveFalse() | … where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | … where UPPER(x.firstame) = UPPER(?1) |

### 2.2依托于repository通过@Query注解实现原生sql查询

## 3.bean的创建依赖

在开发环境中，我们会使用大量的注解创建bean，但是bean之间会有相互依赖或依赖环境中的某些类或变量，在这种情况下，bean的创建顺序颠倒或类和变量的缺失会造成异常。在此情况下，需要我们对依赖于其他bean的bean进行依赖控制。

Springboot提供了以下注解

@ConditionalOnBean在上下文中存在某个bean对象时才会实例化一个bean

@ConditionalOnClass只有存在某个类时才实例化一个bean对象

@ConditionalOnExpression当表达式为true时实例化一个bean对象

@ConditionalOnMissingBean在上下文中不存在某个Bean对象时创建一个bean对象

@ConditionalOnMissingClass在缺乏某个类时才实例化一个对象

## 4.在tomcat上部署文件生成（war包）

public class ServletInitializer extends SpringBootServletInitializer {  
  
 @Override  
 protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {  
 return application.sources(ManagerApplication.class);  
 }  
}

再将内置tomcat包移除

compile('org.springframework.boot:spring-boot-starter-web')  
exclude group: 'org.springframework.boot',name: 'spring-boot-starter-tomcat'

## 5.JPA审计功能

操作数据库映射实体类时，通常需要记录createBy、createAt、updateBy、updateAt，如果每个新增或修改的对象都要手动添加会比较麻烦。

SpringBoot jpa提供个简易方法，四个注解@CreateDate、@LastModifyDate、@CreateBy、@LastModifiedBy。使用这些注解时需要再之程序中添加注解@EnableJpaAuditing

通过实现AuditorAware接口实现createBy和LastModifiedBy的自动写入。

## 6.权限控制思路

### 6.1不使用SpringSecurity，通过springSession进行控制和@ModelAttribute/Filter

本方法适用于权限较少，权限验证逻辑简单的系统。

第一步：

设置一个Session域的Bean的类

@SessionScope  
@Slf4j  
@Getter  
public class ActorResolver {  
 private boolean isAnonymous;  
 private String id;  
 private Role role;  
 private String tenantId;  
  
 @PostConstruct  
 public void init() {  
 this.isAnonymous = true;  
 }  
  
 public void bind(@NonNull User user) throws IllegalArgumentException {  
 this.isAnonymous = false;  
 this.id = user.getId();  
 this.role = user.getRole();  
 if (!this.role.isSystemAdminOrOperator()) {  
 this.tenantId = user.getTenant().getId();  
 }  
 }  
}

第二步：

在配置类中将其注册成Bean

@Bean  
public ActorResolver actorResolver() {  
 return new ActorResolver();  
}

第三步：

通过@ModelAttribute注释/Filter进行登录验证或权限验证

第四步（扩展）：

配置Redis

@Configuration  
@EnableCaching  
public class RedisConfig extends CachingConfigurerSupport {  
 @Value("${spring.redis.host}")  
 private String host;  
  
 @Value("${spring.redis.port}")  
 private int port;  
  
 @Bean  
 public CacheManager cacheManager(@SuppressWarnings("rawtypes") RedisConnectionFactory factory) {  
 RedisCacheManager cacheManager = RedisCacheManager.*create*(factory);  
  
 return cacheManager;  
 }  
  
 @Bean  
 public RedisTemplate<String, String> redisTemplate(RedisConnectionFactory factory) {  
  
 StringRedisTemplate template = new StringRedisTemplate(factory);  
 setSerializer(template); // 设置序列化工具  
 template.afterPropertiesSet();  
 return template;  
 }  
  
 private void setSerializer(StringRedisTemplate template) {  
 @SuppressWarnings({"rawtypes", "unchecked"})  
 Jackson2JsonRedisSerializer jackson2JsonRedisSerializer =  
 new Jackson2JsonRedisSerializer(Object.class);  
 ObjectMapper om = new ObjectMapper();  
 om.setVisibility(PropertyAccessor.*ALL*, JsonAutoDetect.Visibility.*ANY*);  
 om.enableDefaultTyping(ObjectMapper.DefaultTyping.*NON\_FINAL*);  
 jackson2JsonRedisSerializer.setObjectMapper(om);  
 template.setValueSerializer(jackson2JsonRedisSerializer);  
 }  
}

@Configuration  
@EnableRedisHttpSession  
public class HttpSessionConfig {  
  
 @Bean  
 public LettuceConnectionFactory connectionFactory() {  
 return new LettuceConnectionFactory();  
 }  
  
 @Bean  
 public HttpSessionIdResolver httpSessionIdResolver() {  
 return HeaderHttpSessionIdResolver.*xAuthToken*();  
 }  
}

### [6.2使用SpringSecurity的@EnableGlobalMethodScurity](mailto:6.2使用SpringSecurity的@EnableGlobalMethodScurity)注解组进行。

详解：

1、@EnableGlobalMethodSecurity(securedEnabled=true)开启@Secured 注解过滤权限

2、@EnableGlobalMethodSecurity(jsr250Enabled=true)开启@RolesAllowed 注解过滤权限

3、@EnableGlobalMethodSecurity(prePostEnabled=true)使用表达式时间方法级别的安全性 4个注解可用

@PreAuthorize 在方法调用之前,基于表达式的计算结果来限制对方法的访问

@PostAuthorize 允许方法调用,但是如果表达式计算结果为false,将抛出一个安全性异常

@PostFilter 允许方法调用,但必须按照表达式来过滤方法的结果

@PreFilter 允许方法调用,但必须在进入方法之前过滤输入值

@PreAuthorize("hasAuthority('ADMIN')")和@PreAuthorize("hasAuthority('ROLE\_ADMIN')") // 都可以，只要和实现的getAuthoritie里面的role对上就可以

@Secured({"ADMIN"})和@Secured({"ROLE\_ADMIN"}) // 实现的getAuthoritie里面的role都必须要有ROLE\_前缀

第一步：定义配置类继承WebSecurityConfigurerAdapter的类，并添加@EnableGlobalMethodScurity注解

第二步：添加preFilter，设置权限

第三步：在具体方法上添加注解

## 7.多SpringSecurity配置

系统中经常会出现后端服务的不同接口被不同的前端调用，需要不同的权限控制和Spring安全控制。此时就需要进行多SpringSecurity配置。区分不同SpringSecurity原理是每个SpringSecurity都会调用\*Matcher方法进行区分。

## 8.SpringBoot---JPA配置多个数据库连接

在工作中可能涉及到连接多个数据库进行数据之间的验证，在某些场景下我们可能需要在一个项目中配置多个数据，本篇文章简要介绍在Spring Boot中配置JPA的多数据库连接，以配置两个数据库为例。

本例中使用gradle进行版本管理。

1. 添加数据库配置

**manager**:  
 **spring**:  
 **datasource**:  
 **driver-class-name**: org.mariadb.jdbc.Driver  
 **jdbcUrl**: jdbc:mysql://${**MANAGER\_DB\_HOSTNAME:localhost**}:3306/${**MANAGER\_DB\_NAME:manager**}?useSSL=false  
 **username**: ${**MANAGER\_DB\_USER\_NAME:root**}  
 **password**: ${**MANAGER\_DB\_PASSWORD:admin**}  
**slave**:  
 **spring**:  
 **datasource**:  
 **driver-class-name**: org.mariadb.jdbc.Driver  
 **jdbcUrl**: jdbc:mysql://${**SLAVE\_DB\_HOSTNAME:localhost**}:3306/${**SLAVE\_DB\_NAME:slave**}?useSSL=false  
 **username**: ${**SLAVE\_DB\_USER\_NAME:root**}  
 **password**: ${**SLAVE\_DB\_PASSWORD:admin**}

2.创建数据源配置类

@Configuration  
public class DataSourceConfig {  
 */\*\*  
 \* create DataSource instance of manager  
 \*  
 \** ***@return*** *\*/* @Primary  
 @Bean(name = "managerDataSource")  
 @ConfigurationProperties(prefix = "manager.spring.datasource")  
 public DataSource v1DataSource() {  
 return DataSourceBuilder.*create*().build();  
 }  
  
 */\*\*  
 \* create DataSource instance of slave  
 \*  
 \** ***@return*** *\*/* @Bean(name = "slaveDataSource")  
 @ConfigurationProperties(prefix = "slave.spring.datasource")  
 public DataSource v2DataSource() {  
 return DataSourceBuilder.*create*().build();  
 }  
}

3,manager数据库配置

@Configuration  
@EnableTransactionManagement  
@EnableJpaRepositories(  
 entityManagerFactoryRef = "managerEntityManagerFactory",  
 transactionManagerRef = "managerTransactionManager",  
 basePackageClasses = {ManagerRepository.class}  
)  
public class ManagerConfig {  
 @Autowired  
 @Qualifier(value = "managerDataSource")  
 private DataSource managerDataSource;  
  
 @Primary  
 @Bean(name = "managerEntityManager")  
 public EntityManager entityManager(EntityManagerFactoryBuilder builder) {  
 return managerEntityManagerFactory(builder).getObject().createEntityManager();  
 }  
  
 @Primary  
 @Bean(name = "managerEntityManagerFactory")  
 public LocalContainerEntityManagerFactoryBean managerEntityManagerFactory(  
 EntityManagerFactoryBuilder builder) {  
 return builder  
 .dataSource(managerDataSource)  
 .packages(ManagerEntity.class)  
 .persistenceUnit("managerPersistenceUnit")  
 .build();  
 }  
  
 @Primary  
 @Bean(name = "managerTransactionManager")  
 PlatformTransactionManager managerTransactionManager(EntityManagerFactoryBuilder builder) {  
 return new JpaTransactionManager(managerEntityManagerFactory(builder).getObject());  
 }  
}

public interface ManagerEntity {  
}

4.slave配置

@Configuration  
@EnableTransactionManagement  
@EnableJpaRepositories(  
 entityManagerFactoryRef = "slaveEntityManagerFactory",  
 transactionManagerRef = "slaveTransactionManager",  
 basePackageClasses = {SlaveRepository.class}  
)  
public class SlaveConfig {  
 @Autowired  
 @Qualifier(value = "slaveDataSource")  
 private DataSource slaveDataSource;  
  
 @Bean(name = "slaveEntityManager")  
 public EntityManager entityManager(EntityManagerFactoryBuilder builder) {  
 return slaveEntityManagerFactory(builder).getObject().createEntityManager();  
 }  
  
 @Bean(name = "slaveEntityManagerFactory")  
 public LocalContainerEntityManagerFactoryBean slaveEntityManagerFactory(  
 EntityManagerFactoryBuilder builder) {  
 return builder  
 .dataSource(slaveDataSource)  
 .packages(ManagerEntity.class)  
 .persistenceUnit("managerPersistenceUnit")  
 .build();  
 }  
  
 @Bean(name = "slaveTransactionManager")  
 PlatformTransactionManager slaveTransactionManager(EntityManagerFactoryBuilder builder) {  
 return new JpaTransactionManager(slaveEntityManagerFactory(builder).getObject());  
 }  
}

public interface SlaveEntity {  
}