爆破专栏丨Spring Security系列教之基于自定义数据库模型实现授权

原创 一一哥 Java架构栈 9月26日

收录于话题
#Spring Security爆破专栏
16个 >

前言

在上一个章节中，**一一哥** 给大家讲解了如何基于默认的数据库模型来实现认证授权，在这种模型里，用户的信息虽然是保存在数据库中的，但是有很多的限制！

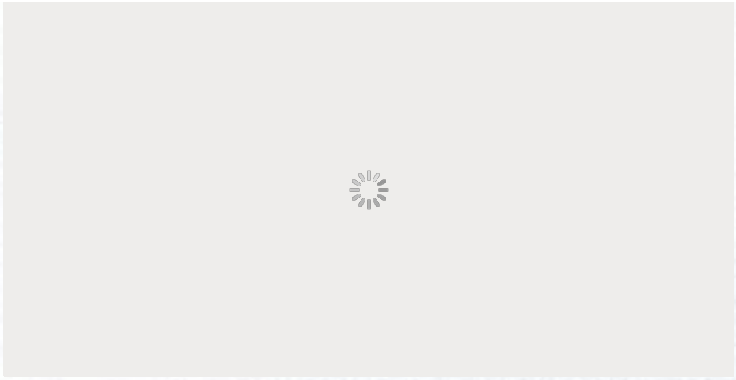
因为我们**必须按照源码规定的方式去建库建表，存在灵活性不足的问题。**而我们真正开发时，用户角色等表肯定要根据自己的项目需求来单独设计，所以我们有必要进行用户及角色表的自定义设计。

那么在本篇文章中，**壹哥**就带各位，结合自己的实际项目需求，进行数据库和表的自定义设计，然后在这个自定义的数据库中实现用户信息的认证与授权工作。

一. 核心API源码介绍

1. UserDetailsService源码

在上一章节中，我给大家介绍了一个JdbcUserDetailsManager类，其结构关系如下图所示：



JdbcUserDetailsManager类**可以实现基于默认数据库模型的授权认证**，但是如果我们想要采用一个更加灵活的方式--基于自定义数据库模型来实现认证授权，那么就需要利用UserDetailsService接口来实现了。

请先跟着**壹哥**来看看UserDetailsService接口的源码，简单了解一下该类的作用。

```
public interface UserDetailsService {

 /**
  * Locates the user based on the username.
  */
 UserDetails loadUserByUsername(String username) throws UsernameNotFoundExcept
}
```

从源码中可以看出，我们可以利用loadUserByUsername()方法，根据用户名查询出对应的UserDetails信息，那么UserDetails是什么呢?

2. UserDetails源码

```
public interface UserDetails extends Serializable {

 Collection<? extends GrantedAuthority> getAuthorities();

 String getPassword();

 String getUsername();

 boolean isAccountNonExpired();
```

```
    boolean isAccountNonLocked();

    boolean isCredentialsNonExpired();

    boolean isEnabled();
}
```

从UserDetails的源码中我们了解到，UserDetails其实就是一个包含了User信息的类，其中包含了用户名、密码、角色及账号状态等信息。

利用以上的两个核心API，我们就可以进行基于自定义数据库模型的认证授权工作了，因为不管我们项目中关于认证、授权的数据库结构如何变化，只要我们构造出一个UserDetails类，然后利用UserDetailsService进行用户信息的加载就可以了。

二. 基于自定义数据库模型实现授权

1. 创建数据库

在开始今天的代码之前，请先跟着壹哥来创建一个数据库，并在该库里创建用户角色表，建表脚本如下:

```
CREATE TABLE `users` (`id` bigint(20) NOT NULL AUTO_INCREMENT,
        `username` varchar(50) NOT NULL,
        `password` varchar(60) NOT NULL,
        `enable` tinyint(4) NOT NULL DEFAULT `1`,
        `roles` text character set utf8,
        PRIMARY KEY (`id`),
        KEY `username` (`username`)
        ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
```

该表中包含了用户名、密码、角色、可用状态等信息，然后我们在该表中添加一些测试数据，如下图所示:

2. 创建项目

准备好了数据库之后，**我们就继续在之前项目的基础之上，创建一个新的model模块**，其基本配置信息和之前一样，具体创建过程略，请参考之前的章节进行项目创建！

3. 创建User实体类

```
/**
 * 用户操作实体类
 */
public class User implements UserDetails {

    private Long id;

    private String username;

    private String password;

    private String roles;

    private boolean enable;

    private List<GrantedAuthority> authorities;

    public String getRoles() {
        return roles;
    }

    public void setRoles(String roles) {
        this.roles = roles;
    }

    public Long getId() {
        return id;
    }

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

    public boolean isEnable() {
```

```
        return enable;
    }

    public void setEnable(boolean enable) {
        this.enable = enable;
    }

    @Override
    public String getUsername() {
        return username;
    }

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

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return this.enable;
    }

    public void setAuthorities(List<GrantedAuthority> authorities) {
        this.authorities = authorities;
    }
```

```
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return this.authorities;
    }

    @Override
    public String getPassword() {
        return password;
    }

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

    @Override
    public boolean equals(Object obj) {

        return obj instanceof User?this.username.equals(((User)obj).username):
    }

    @Override
    public int hashCode() {
        return this.username.hashCode();
    }

}
```

在刚才的模块里，我们先创建一个User实体类，这个用户实体类需要实现 **UserDetails** 接口，并实现接口中的方法。

核心方法介绍：

- accountNonExpired、accountNonLocked、credentialsNonExpired、enabled 这四个属性分别用来描述用户的状态，表示账户是否没有过期、账户是否没有被锁定、密码是否没有过期、以及账户是否可用；
- roles 属性表示用户的角色；

- getAuthorities 方法返回用户的角色信息，一个用户可能会有多个角色，所以这里返回值是一个集合类型，我们在这个方法中把自己的 Role 角色稍微转化一下即可。

4. 定义Mapper接口

数据模型准备好之后，我们再来定义一个 UserMapper接口，这里我们利用Mybatis进行具体的数据库查询，直接利用注解的方式实现即可。

```
/**
 * @Mapper注解,可带可不带,因为有MapperScan扫描.
 */
@Mapper
public interface UserMapper {

    @Select("SELECT * FROM users WHERE username=#{username}")
    User findByUserName(@Param("username") String username);

}
```

5. 实现UserDetailsService接口

接在在service层，定义一个UserDetailsService子类，**实现UserDetailsService接口**，然后实现该接口中的loadUserByUsername()方法。这个方法的参数是用户在登录的时候传入的用户名，然后根据用户名去查询用户信息(查出来之后，系统会自动进行密码比对)。

```
@Service
public class MyUserDetailsService implements UserDetailsService {

    @Autowired
    private UserMapper userMapper;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotF
        // 从数据库尝试读取该用户
        User user = userMapper.findByUserName(username);
```

```
        // 用户不存在，抛出异常
        if (user == null) {
            throw new UsernameNotFoundException("用户不存在");
        }

        // 将数据库形式的roles解析为UserDetails的权限集
        // AuthorityUtils.commaSeparatedStringToAuthorityList是Spring Security
        //提供的用于将逗号隔开的权限集字符串切割成可用权限对象列表的方法
        // 当然也可以自己实现，如用分号来隔开等，参考generateAuthorities
        user.setAuthorities(AuthorityUtils.commaSeparatedStringToAuthorityList
        return user;
    }

}
```

主要要在实现方法中，利用user对象的setAuthorities()方法关联用户的所有角色信息，否则用户登录认证时就没有角色，认证也会失败。

6. 配置SecurityConfig类

编写完上面的UserDetailsService接口后，接下来我们创建SecurityConfig配置类，我们在configure方法中，关联配置自定义的UserService对象。

```
@EnableWebSecurity(debug = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .antMatchers("/admin/**")
                .hasRole("ADMIN")
                .antMatchers("/user/**")
                .hasRole("USER")
                .antMatchers("/visitor/**")
                .permitAll()
                .anyRequest()
                .authenticated()
```

```
                .and()
                .formLogin()
                .permitAll();
    }

    /**************************************在数据库中创建用户和角色**************

    @Autowired
    private MyUserDetailsService userDetailsService;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exceptio
        //关联UserDetailsService对象
        auth.userDetailsService(userDetailsService)
                //暂时不对密码进行加密配置
                .passwordEncoder(NoOpPasswordEncoder.getInstance());
    }

}
```

7. 配置入口类

我们这里因为是结合Mybatis来实现数据库操作的，所以一定要在入口类中，利用@MapperScan注解进行Mapper组件的扫描。

```
@SpringBootApplication
@MapperScan("com.yyg.security.mapper")
public class Demo03Application {

    public static void main(String[] args) {

        SpringApplication.run(Demo03Application.class, args);
    }

}
```

8. 添加数据库配置

接下来我们在application.yml文件中进行数据库连接的配置，请关联配置自己的数据库信

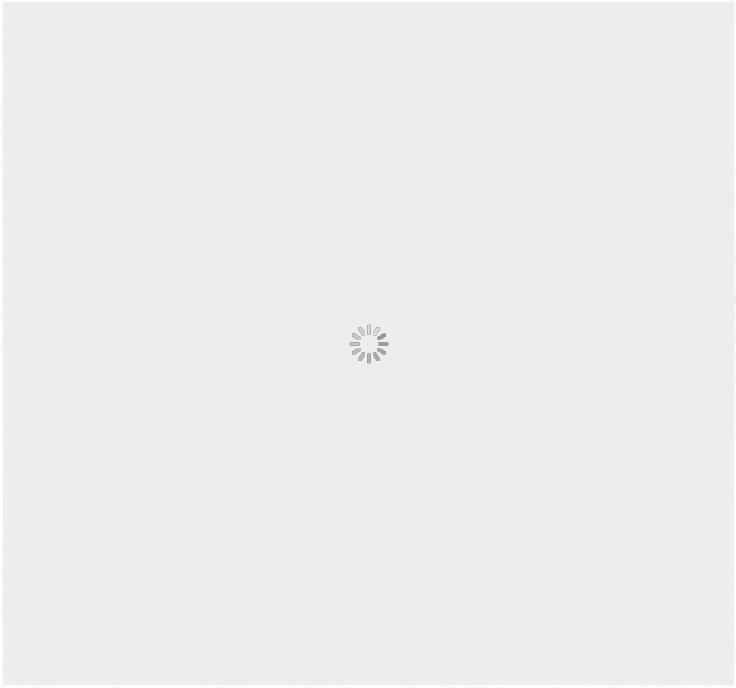
息。

```
spring:  datasource:    url: jdbc:mysql://localhost:3306/db-security?useUnicode
```

至此，本案例的核心代码我就给各位编写完毕啦！

9. 代码结构

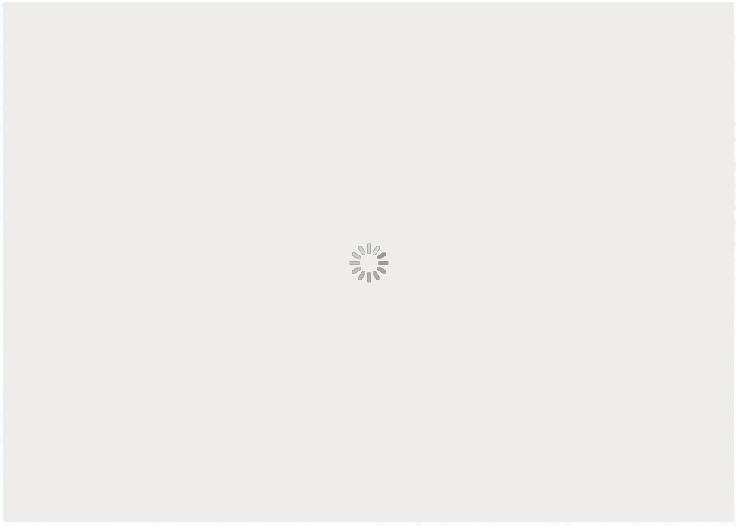
现在我们整个项目的代码结构如下，各位可以参考创建自己的项目：



三. 启动项目测试

编写完代码之后，接下来我们就把项目启动起来进行测试，这时候的效果跟基于内存模型的授权实现效果是一样的，具体界面我们不再展示。

在访问**/admin/**，**/user/**等接口时就会跳转到如下登录界面，登录认证成功后，会跳转到对应的接口访问页面。



好了，我们今天又学会了如何基于自定义的数据库模型进行认证授权了，是不是也很简单？如果你有不明白的地方，记得评论区留言哦！

往期精彩

Spring Security系列教程之创建项目(1)

Spring Security系列教程之实现HTTP基本认证(2)

Spring Security系列教程之实现Form表单认证(3)

Spring Security系列之实现HTTP摘要认证(4)

Spring Security系列之前后端分离时的安全处理方案(5)

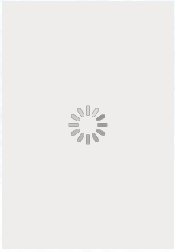
Spring Security系列教程之基于内存模型实现授权（6）

Spring Security系列教程之基于默认数据库模型实现授权（7）

文末福利

留言评论，今天的内容你学会了吗？

点赞最多的同学获得水杯一个！



收录于话题 #Spring Security爆破专栏·16个 >

上一篇

爆破专栏丨系列教程之Spring Security核心API讲解

下一篇

爆破专栏丨Spring Security系列教程之基于默认数据库模型实现授权

阅读原文