前期内容

- 基于内存实现用户的认证和授权
- SpringSecurity支持两种权限认证
  - 基于请求【需要配置类中配置】
  - 基于方法【需要开启注解，在类或者方法上使用响应的注解】
- 实现自定义的UserDetailsService，实现自定义登录。基于数据库实现

动态权限控制

SpringSecurity中我们可以使用基于注解的方式实现，可以控制方法级别的权限认证，有一个问题在于项目中的接口权限如果发生变化，此时代码就要修改，修改之后要重新上线。基于此我们希望可以通过修改配置的方式，去控制接口的权限，实现动态控制。此处我们是通过数据库配置的方式修改【此处也可以将数据缓存到Redis中，因为接口权限不会经常变化】，在SpringSecurity6中实现动态权限控制：

- SpringSecurity配置文件中再其它请求的配置后边，添加access配置，传入一个【AuthorizationManager】
- 自定义类实现 AuthorizationManager 接口

实现思路

在实现类中查询当前访问的接口所需要的权限，并且根据当前认证的用户所拥有的权限，判断是否包含接口所需权限，所有数据库中应该有数据记录接口的访问权限

访问路径：其实是一个url，可以对url进行权限设置。比如sys/employee/list他的作用是访问员工列表，他需要sys:employee:list【这个权限也是存到数据库中】

修改数据表

```
CREATE TABLE `ums_menu` (
  `id` bigint NOT NULL AUTO_INCREMENT COMMENT '主键',
  `parent_id` bigint NOT NULL DEFAULT '0' COMMENT '父id',
  `menu_name` varchar(255) DEFAULT NULL COMMENT '菜单名',
  `path` varchar(255) DEFAULT NULL COMMENT '访问路径',
  `sort` int DEFAULT '0' COMMENT '排序',
  `perms` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT
NULL COMMENT '权限标识',
  `menu_type` int DEFAULT NULL COMMENT '类型：0，目录，1菜单，2：按钮',
  `icon` varchar(255) DEFAULT NULL COMMENT '图标',
  `deleted` int DEFAULT NULL COMMENT '是否删除',
  `create_time` datetime DEFAULT NULL COMMENT '创建时间',
  `update_time` datetime DEFAULT NULL COMMENT '修改时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=11 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci;
```

实现AuthorizationManager 接口

在该接口中，获取访问路径【URI】,再获取用户的权限列表，做判断就行了，写完之后需要配置到配置类中

```
package com.stt.springsecuritydemo7.web.manager;

import com.baomidou.mybatisplus.core.conditions.query.LambdaQueryWrapper;
import com.stt.springsecuritydemo7.domain.entity.UmsMenu;
import com.stt.springsecuritydemo7.mapper.UmsMenuMapper;
import jakarta.servlet.http.HttpServletRequest;
import lombok.extern.slf4j.Slf4j;
import org.springframework.security.authorization.AuthorizationDecision;
import org.springframework.security.authorization.AuthorizationManager;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.GrantedAuthority;
import
org.springframework.security.web.access.intercept.RequestAuthorizationContext;
import org.springframework.stereotype.Component;
import java.util.Collection;
import java.util.function.Supplier;

/**
 * 判断请求路径是否有权限访问
 */
@Component
@Slf4j
public class SttAuthorizationManager implements
AuthorizationManager<RequestAuthorizationContext> {

    private final UmsMenuMapper menuMapper;

    public SttAuthorizationManager(UmsMenuMapper menuMapper) {
        this.menuMapper = menuMapper;
    }

    @Override
    public AuthorizationDecision check(Supplier<Authentication> authentication,
RequestAuthorizationContext requestAuthorizationContext) {
        // 获取请求路径,获取HttpServletRequest
        HttpServletRequest request = requestAuthorizationContext.getRequest();
        String uri = request.getRequestURI();
        String url = request.getRequestURL().toString();
        // 有些请求不需要认证
        if("/auth/login".equals(uri) || "/logout".equals(uri) ||
"/error".equals(uri)) {
            return new AuthorizationDecision(true);
        }
        // 根据uri获取路径的权限
        UmsMenu umsMenu = menuMapper.selectOne(new LambdaQueryWrapper<UmsMenu>
().eq(UmsMenu::getPath, uri.replaceFirst("/","")));
        if(umsMenu == null) {
            return new AuthorizationDecision(false);
        }
        // 获取路径访问权限
```

```
        String menuPerm = umsMenu.getPerms();
        if(menuPerm == null || menuPerm.trim().equals("")) {
            return new AuthorizationDecision(true);
        }

        // 与用户权限集合做判断
        Collection<? extends GrantedAuthority> authorities =
authentication.get().getAuthorities();
        for (GrantedAuthority authority : authorities) {
            String userPerm = authority.getAuthority();
            if(userPerm.equals(menuPerm)) {
                return new AuthorizationDecision(true);
            }
        }
        return new AuthorizationDecision(false);
    }
}
```

动态权限问题

1. 请求时每次都需要查询数据库菜单的权限，会对数据库造成压力。可以通过redis来缓解数据库压力，引入redis之后需要解决redis与mysql的数据一致性问题
2. 用户的权限是在登陆时就获取到的，后续的操作并不会获取最新的权限，此时也是可以通过redis来存储用户信息【包含权限信息】，修改权限之后需要修改：
    1. redis中缓存的权限数据
    2. 用户的权限数据
        1. 用户很多，修改繁琐
        2. 用户量不大，改了也就改了

> 权限这种数据一般很少修改

rememberMe功能

rememberMe也就是记住我功能，在登陆时可以勾选此选项，客户端会记住我们的登录信息，下次访问时可以根据记住的登录信息，实现自动登录。

QQ邮箱

163邮箱

gitee

记住我或持久登录身份验证是指网站能够在会话之间记住主体的身份。这通常是通过向浏览器发送cookie来实现的，cookie在后续的会话中被检测到，并导致自动登录。Spring Security为这些操作提供了两个具体的实现。

- 使用 哈希编码 方式保持基于cookie的令牌的安全性
- 使用 数据库 或其他持久存储机制来存储生成的令牌。

请注意，这两个实现都需要UserDetailsService。

简单哈希方法

这种方法使用散列来实现有用的记住我的策略。本质上，一个cookie在成功的交互验证后被发送到浏览器，cookie的组成如下：

```
base64(username + ":" + expirationTime + ":" + algorithmName + ":"
algorithmHex(username + ":" + expirationTime + ":" password + ":" + key))

username: 可由UserDetailsService识别
password: 与检索到的UserDetails中的匹配
expirationTime: “记住我”令牌过期的日期和时间，以毫秒表示
key: 防止修改记住我令牌的私钥
algorithmName:用于生成和验证记住我的令牌签名的算法
```

记住我令牌仅在指定的时间段内有效，并且仅在用户名、密码和密钥未更改的情况下有效。值得注意的是，这有一个潜在的安全问题，因为捕获的记住我令牌在令牌到期之前可以从任何用户代理使用。如果主体知道令牌被捕获，他们可以很容易地更改密码，并立即使所有有问题的记住我的令牌无效。如果需要更重要的安全性，应该使用持久化令牌。**或者，根本不应该使用“记住我”服务。**

简单的哈希方式只需要在配置类中开启rememberMe功能即可

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    // 关闭csrf
    http.csrf(csrf -> csrf.disable());
    // 配置请求拦截策略
    http.authorizeHttpRequests(auth -> auth.anyRequest().authenticated());
    // 使用SpringSecurity默认的登录页面
    http.formLogin(Customizer.withDefaults());
    // 开启默认的记住我功能,rememberMeCookieName:设置浏览器断存储的cookeie名称，默认为
remember-me
    http.rememberMe(remember -> remember.rememberMeCookieName("rememberMe"));
    return http.build();
}
```

此方式可以将cookie获取到之后放到另一个客户端照样可以访问，而且不会修改cookie值，很不安全，如果需要使用记住我功能可以通过`持久化令牌`方式`会安全一点`，但不能保障完全安全哦！

持久化令牌

持久化令牌就是将令牌存储在数据库中，数据库应包含一个`persistent_logins`表，该表通过使用以下SQL（或等效SQL）创建：

```
create table persistent_logins (
    series varchar(64) primary key,
    username varchar(64) not null,
    token varchar(64) not null,
    last_used timestamp not null
)
```

实体类

```
@Data
@TableName("persistent_logins")
public class PersistentLogins {
    private String series;
    private String username;
    private String token;
    private Date lastUsed;
}
```

实现类

自定义实现类对数据表进行操作，其实就是在前端登陆时如果选择了记住我功能，则生成token，存到数据表中，退出登录需要将token删除掉。

退出客户端之后再来访问时，如果上一次选择了记住我则`自动登录`。还会更新token。如果cookie被盗取，则会修改cookie的值，大家可以自行测试一下哦，**印象会更深刻**！

其实就是在实现类中实现token的，增删改查方法。

```
package com.stt.springsecuritydemo8.token;

import com.baomidou.mybatisplus.core.conditions.query.LambdaQueryWrapper;
import com.baomidou.mybatisplus.core.conditions.update.LambdaUpdateWrapper;
import com.stt.springsecuritydemo8.domain.emtity.PersistentLogins;
import com.stt.springsecuritydemo8.mapper.PersistentLoginsMapper;
import lombok.extern.slf4j.Slf4j;
import
org.springframework.security.web.authentication.rememberme.PersistentRememberMeT
oken;
import
org.springframework.security.web.authentication.rememberme.PersistentTokenReposi
tory;
import org.springframework.stereotype.Component;
import java.util.Date;

@Component
@Slf4j
public class DaoCaoPersistentTokenRepositoryImpl implements
PersistentTokenRepository {

    private final PersistentLoginsMapper persistentLoginsMapper;

    public DaoCaoPersistentTokenRepositoryImpl(PersistentLoginsMapper
persistentLoginsMapper) {
        this.persistentLoginsMapper = persistentLoginsMapper;
    }

    /**
     * 创建token
     * @param token
     */
    @Override
    public void createNewToken(PersistentRememberMeToken token) {
        PersistentLogins persistentLogins = new PersistentLogins();
        persistentLogins.setSeries(token.getSeries());
        persistentLogins.setUsername(token.getUsername());
        persistentLogins.setToken(token.getTokenValue());
        persistentLogins.setLastUsed(token.getDate());
        // 存储到数据库中
        persistentLoginsMapper.insert(persistentLogins);
    }

    @Override
    public void updateToken(String series, String tokenValue, Date lastUsed) {
        LambdaUpdateWrapper<PersistentLogins> updateWrapper = new
LambdaUpdateWrapper<>();

 updateWrapper.set(PersistentLogins::getToken,tokenValue).set(PersistentLogins::
getLastUsed,lastUsed)
                .eq(PersistentLogins::getSeries,series);
        // 调用修改方法
        persistentLoginsMapper.update(null,updateWrapper);
```

```
    }

    /**
     * 获取token
     * @param seriesId
     * @return
     */
    @Override
    public PersistentRememberMeToken getTokenForSeries(String seriesId) {
        LambdaQueryWrapper<PersistentLogins> queryWrapper = new
LambdaQueryWrapper<>();
        queryWrapper.eq(PersistentLogins::getSeries,seriesId);
        // 调用查询方法
        PersistentLogins result =
persistentLoginsMapper.selectOne(queryWrapper);

        PersistentRememberMeToken retrunResult = new PersistentRememberMeToken(

 result.getUsername(),result.getSeries(),result.getToken(),result.getLastUsed()
        );
        return retrunResult;
    }

    @Override
    public void removeUserTokens(String username) {
        LambdaUpdateWrapper<PersistentLogins> updateWrapper = new
LambdaUpdateWrapper<>();
        updateWrapper.eq(PersistentLogins::getUsername,username);
        persistentLoginsMapper.delete(updateWrapper);
    }
}
```

配置

将实现类配置到rememberMe功能上

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private DaoCaoPersistentTokenRepositoryImpl persistentTokenRepository;
    // 配置，启用rememberMe功能
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        // 关闭csrf
        http.csrf(csrf -> csrf.disable());
        // 配置请求拦截策略
        http.authorizeHttpRequests(auth -> auth.anyRequest().authenticated());
        // 使用SpringSecurity默认的登录页面
        http.formLogin(Customizer.withDefaults());
        // 开启默认的记住我功能
        http.rememberMe(remember -> remember.rememberMeCookieName("rememberMe")
                        .tokenRepository(persistentTokenRepository));
        return http.build();
```

```
    } );
    }
}
```

实现原理

RememberMeAuthenticationFilter

当启用了记住我功能之后，会自动启用`RememberMeAuthenticationFilter`这个过滤器，该过滤器的doFilter方法如下

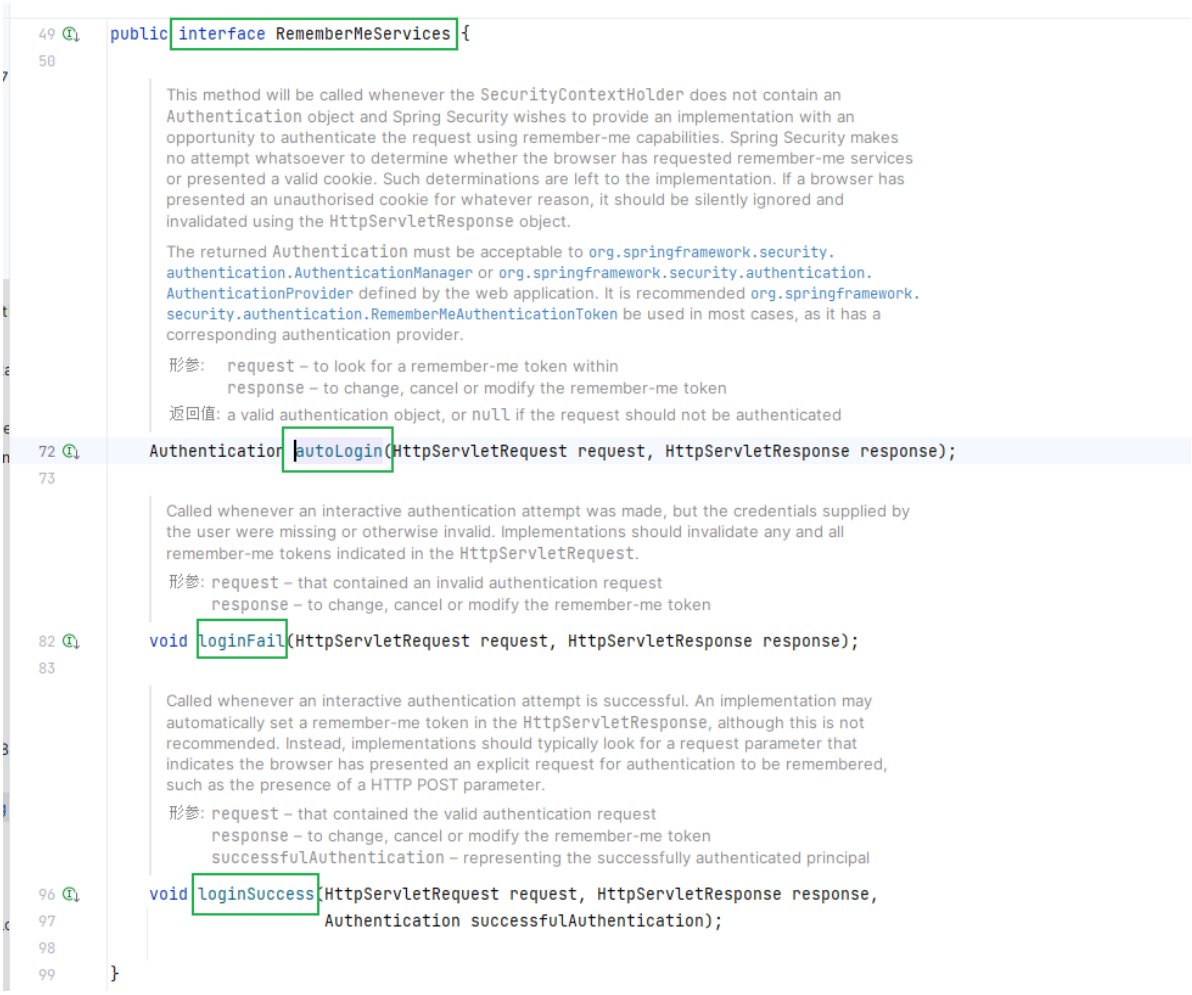
```
private void doFilter(HttpServletRequest request, HttpServletResponse response,
FilterChain chain)
    throws IOException, ServletException {
    // 从SecurityContext中获取Authentication，如果有则认为已经认证过了，直接放行
    // 如果没有 Authentication 则认为没有认证，会去做自动登录【认证就是登录】
    if (this.securityContextHolderStrategy.getContext().getAuthentication() !=
null) {
        this.logger.debug(LogMessage
                          .of(() -> "SecurityContextHolder not populated with
remember-me token, as it already contained: '"
                                  +
this.securityContextHolderStrategy.getContext().getAuthentication() + "'"));
        chain.doFilter(request, response);
        return;
    }
    // 此处调用 rememberMeServices 的autoLogin方法做自动登录，登录成功则返回一个
Authentication对象
    // 对象名为 rememberMeAuth，此处其实就是根据request中的cookie做解码操作判断cookie值是
否正确
    // 后边会详细介绍该方法，该方法在 RememberMeServices类中
    Authentication rememberMeAuth = this.rememberMeServices.autoLogin(request,
response);
    // 判断不是null，则认为该cookie存在
    if (rememberMeAuth != null) {
        // Attempt authentication via AuthenticationManager
        try {
            // 此处调用 authenticate方法认证用户
            rememberMeAuth =
this.authenticationManager.authenticate(rememberMeAuth);
            // 获取 SecurityContext 对象
            SecurityContext context =
this.securityContextHolderStrategy.createEmptyContext();
            // 将 Authentication 放入 SecurityContext 中，大家应该还记得两者的关系吧
            context.setAuthentication(rememberMeAuth);
            this.securityContextHolderStrategy.setContext(context);
            onSuccessfulAuthentication(request, response, rememberMeAuth);
            this.logger.debug(LogMessage.of(() -> "SecurityContextHolder
populated with remember-me token: '"
                                    +
this.securityContextHolderStrategy.getContext().getAuthentication() + "'"));
            this.securityContextRepository.saveContext(context, request,
response);
```

```
            if (this.eventPublisher != null) {
                this.eventPublisher.publishEvent(new
InteractiveAuthenticationSuccessEvent(

 this.securityContextHolderStrategy.getContext().getAuthentication(),
this.getClass()));
            }
            if (this.successHandler != null) {
                this.successHandler.onAuthenticationSuccess(request, response,
rememberMeAuth);
                return;
            }
        }
        // 如果认证过程中出现异常
        catch (AuthenticationException ex) {
            this.logger.debug(LogMessage
                                .format("SecurityContextHolder not populated with
remember-me token, as AuthenticationManager "
                                        + "rejected Authentication returned by
RememberMeServices: '%s'; "
                                        + "invalidating remember-me token",
rememberMeAuth),
                                ex);
            // 调用 rememberMeServices中的 登陆失败方法
            this.rememberMeServices.loginFail(request, response);
            onUnsuccessfulAuthentication(request, response, ex);
        }
    }
    chain.doFilter(request, response);
}
```

RememberMeServices

通过上边的代码看出，获取cookie是通过RememberMeServices的autoLogin方法实现，登录失败的话要调用loginFail方法，其实它是一个接口，里边主要包含三个方法，这三个方法都有对应的类做了一些实现，如果想自定义的话，就是实现这个接口或者继承内部默认的实现类`AbstractRememberMeServices`就可以了



```
49  public interface RememberMeServices {
50

        This method will be called whenever the SecurityContextHolder does not contain an
        Authentication object and Spring Security wishes to provide an implementation with an
        opportunity to authenticate the request using remember-me capabilities. Spring Security makes
        no attempt whatsoever to determine whether the browser has requested remember-me services
        or presented a valid cookie. Such determinations are left to the implementation. If a browser has
        presented an unauthorised cookie for whatever reason, it should be silently ignored and
        invalidated using the HttpServletResponse object.

        The returned Authentication must be acceptable to org.springframework.security.
        authentication.AuthenticationManager or org.springframework.security.authentication.
        AuthenticationProvider defined by the web application. It is recommended org.springframework.
        security.authentication.RememberMeAuthenticationToken be used in most cases, as it has a
        corresponding authentication provider.

        形参:  request – to look for a remember-me token within
               response – to change, cancel or modify the remember-me token
        返回值: a valid authentication object, or null if the request should not be authenticated
72      Authentication autoLogin(HttpServletRequest request, HttpServletResponse response);
73

        Called whenever an interactive authentication attempt was made, but the credentials supplied by
        the user were missing or otherwise invalid. Implementations should invalidate any and all
        remember-me tokens indicated in the HttpServletRequest.

        形参: request – that contained an invalid authentication request
              response – to change, cancel or modify the remember-me token
82      void loginFail(HttpServletRequest request, HttpServletResponse response);
83

        Called whenever an interactive authentication attempt is successful. An implementation may
        automatically set a remember-me token in the HttpServletResponse, although this is not
        recommended. Instead, implementations should typically look for a request parameter that
        indicates the browser has presented an explicit request for authentication to be remembered,
        such as the presence of a HTTP POST parameter.

        形参: request – that contained the valid authentication request
              response – to change, cancel or modify the remember-me token
              successfulAuthentication – representing the successfully authenticated principal
96      void loginSuccess(HttpServletRequest request, HttpServletResponse response,
97                        Authentication successfulAuthentication);
98
99  }
```

autoLogin

此处的autoLogin代码就是`AbstractRememberMeServices`这个抽象类的默认实现：

```
public Authentication autoLogin(HttpServletRequest request, HttpServletResponse
response) {
    // 调用获取cookie的方法，该方法的源码放到下边了，返回的值就是rememberMe的值
    String rememberMeCookie = extractRememberMeCookie(request);
    if (rememberMeCookie == null) {
        return null;
    }
    this.logger.debug("Remember-me cookie detected");
    if (rememberMeCookie.length() == 0) {
        this.logger.debug("Cookie was empty");
        cancelCookie(request, response);
        return null;
    }
    // 如果不是null，长度业大于0
    try {
        // 解析Cookie，该方法实现也放在下边了，
        String[] cookieTokens = decodeCookie(rememberMeCookie);
        // 此方法就会去调用loadUserByUsername方法实现认证了。
        // 此处会根据你是基于哈希，还是持久化令牌方法去掉用不同的 RememberMeServices的方法
        // 1、TokenBasedRememberMeServices：哈希方式
        // 2、PersistentTokenBasedRememberMeServices：持久化令牌方式
        UserDetails user = processAutoLoginCookie(cookieTokens, request,
response);
        this.userDetailsChecker.check(user);
```

```
            this.logger.debug("Remember-me cookie accepted");
            return createSuccessfulAuthentication(request, user);
        }
        catch (CookieTheftException ex) {
            cancelCookie(request, response);
            throw ex;
        }
        catch (UsernameNotFoundException ex) {
            this.logger.debug("Remember-me login was valid but corresponding user
not found.", ex);
        }
        catch (InvalidCookieException ex) {
            this.logger.debug("Invalid remember-me cookie: " + ex.getMessage());
        }
        catch (AccountStatusException ex) {
            this.logger.debug("Invalid UserDetails: " + ex.getMessage());
        }
        catch (RememberMeAuthenticationException ex) {
            this.logger.debug(ex.getMessage());
        }
        // 清除cookie
        cancelCookie(request, response);
        return null;
    }

    // 获取cookie值
    protected String extractRememberMeCookie(HttpServletRequest request) {
        Cookie[] cookies = request.getCookies();
        if ((cookies == null) || (cookies.length == 0)) {
            return null;
        }
        // 循环遍历cookie，获取名称符合条件的，也就是rememberMe
        for (Cookie cookie : cookies) {
            if (this.cookieName.equals(cookie.getName())) {
                return cookie.getValue();
            }
        }
        return null;
    }
    // 解析cookie，返回的是String[]数组，里边包含了cookie的值，编码方式，过期时间等四个信息
    protected String[] decodeCookie(String cookieValue) throws
InvalidCookieException {
        for (int j = 0; j < cookieValue.length() % 4; j++) {
            cookieValue = cookieValue + "=";
        }
        String cookieAsPlainText;
        try {
            cookieAsPlainText = new
String(Base64.getDecoder().decode(cookieValue.getBytes()));
        }
        catch (IllegalArgumentException ex) {
            throw new InvalidCookieException("Cookie token was not Base64 encoded;
value was '" + cookieValue + "'");
        }
```

```
    String[] tokens = StringUtils.delimitedListToStringArray(cookieAsPlainText,
DELIMITER);
    for (int i = 0; i < tokens.length; i++) {
        try {
            tokens[i] = URLDecoder.decode(tokens[i],
StandardCharsets.UTF_8.toString());
        }
        catch (UnsupportedEncodingException ex) {
            this.logger.error(ex.getMessage(), ex);
        }
    }
    return tokens;
}
```

TokenBasedRememberMeServices

如果是传统哈希方式实现的rememberMe就会调用`TokenBasedRememberMeServices`的`processAutoLoginCookie`方法实现自动登录

```
protected UserDetails processAutoLoginCookie(String[] cookieTokens,
HttpServletRequest request,
            HttpServletResponse response) {
    if (!isValidCookieTokensLength(cookieTokens)) {
        throw new InvalidCookieException(
            "Cookie token did not contain 3 or 4 tokens, but contained '" +
Arrays.asList(cookieTokens) + "'");
    }
    // 获取过期时间
    long tokenExpiryTime = getTokenExpiryTime(cookieTokens);
    if (isTokenExpired(tokenExpiryTime)) {
        throw new InvalidCookieException("Cookie token[1] has expired (expired
on '" + new Date(tokenExpiryTime)
                                        + "'; current time is '" + new Date() +
"')");
    }
    // 调用 UserDetailsService的 loadUserByUsername方法
    UserDetails userDetails =
getUserDetailsService().loadUserByUsername(cookieTokens[0]);
    Assert.notNull(userDetails, () -> "UserDetailsService " +
getUserDetailsService()
                    + " returned null for username " + cookieTokens[0] + ". " +
"This is an interface contract violation");
    // 获取签名方式
    String actualTokenSignature = cookieTokens[2];
    RememberMeTokenAlgorithm actualAlgorithm = this.matchingAlgorithm;
    // If the cookie value contains the algorithm, we use that algorithm to check
the
    // signature
    if (cookieTokens.length == 4) {
        actualTokenSignature = cookieTokens[3];
        actualAlgorithm = RememberMeTokenAlgorithm.valueOf(cookieTokens[2]);
    }
    // 根据数字签名计算cookie
    String expectedTokenSignature = makeTokenSignature(tokenExpiryTime,
userDetails.getUsername(),
```

```
userDetails.getPassword(), actualAlgorithm);
    if (!equals(expectedTokenSignature, actualTokenSignature)) {
        throw new InvalidCookieException("Cookie contained signature '" +
actualTokenSignature + "' but expected '"
                                         + expectedTokenSignature + "'");
    }
    return userDetails;
}
```

总的来说，RememberMe功能实现是通过上述三个类，可以自己debug一下，印象更深哦！

总结

1. RememberMe 实质就是将令牌以 Cookie 的形式响应给浏览器，然后浏览器进行本地存储，等下次未登录时，再次再访问系统时，即会拿该令牌连请求一起到服务器端，令牌会使得后台进行自动登录（即用户认证）。
2. 持久化token方式在关闭浏览器再次访问时存储的cookie值会发生变化，而哈希方式并不会。
3. 当我们自定义 SecurityFilterChain 安全过滤器链的时候，如果配置了 rememberMe() 的话，那 RememberMeAuthenticationFilter 即会是过滤器链中的一员。
4. 阐述一下流程（实现该功能最主要的类是 RememberMeServices，首次认证的响应和自动登录的认证它都是主角）：在没有自定义过滤器的情况下，一般都是使用 UsernamePasswordAuthenticationFilter 进行用户认证的，当点击了 checkbox 按钮连同用户信息一起向服务器端请求时，首先是 UsernamePasswordAuthenticationFilter 去完成认证，如果认证成功会调用 RememberMeServices 中的 onLoginSuccess 方法（这个看具体实现），此时响应中就已存在其令牌信息了（即 Cookie）。当关闭浏览器或Session过期，访问服务器端需要认证的资源时，请求报文中会带着这个Cookie一起到服务器端，会进入到 RememberMeAuthenticationFilter 过滤器中，它会调用 RememberMeServices 中的 autoLogin 方法进行自动登录，在 autoLogin 自动登录过程中，会调用 processAutoLoginCookie 方法进行令牌认证（该方法取决于 RememberMeServices 的实现类），autoLogin 方法执行完成认证成功后，返回了用户数据源信息，接下来就是一些封装数据信息到 SecurityContextHolder 中等等一些操作。
5. RememberMeServices 是核心类，Spring Security 中提供了两种实现类，一种是 TokenBasedRememberMeServices，这种方式就是将用户名、超时时间、密码等信息进行 Base64编码即一些操作组成的令牌给服务器，验证令牌就是对浏览器中发来的进行反编码看看是否一致，这种方式安全度很低；另一种实现是 PersistentTokenBasedRememberMeServices，它内部依赖于 PersistentTokenRepository 仓库，提供了基于内存和基于 Jdbc 的实现，它比前一种安全，原因是它每次自动认证后会更新令牌（即Cookie），如果在自动认证过程中发现令牌不一致会及时剔除（即从PersistentTokenRepository仓库中删除），报Cookie被盗窃异常。
6. PersistentTokenBasedRememberMeServices 中同 TokenBasedRememberMeServices 一样的是它也使用的是 Base64 编码后进行令牌设置，自动登录认证令牌的时候也需要解码；不同的是它是由两数据组成的（序列号（固定的），token（会变化的）），序列号在 onLoginSuccess 方法中生成的，然后存在浏览器上，在 processAutoLoginCookie 方法中不会改变这个序列号，只会变token，可以说浏览器Cookie解码后的序列号是固定死的（没有重新登录验证的情况下）。
7. 使用起来很简单（一般使用的是PersistentTokenBasedRememberMeServices，且使用的仓库实现是 JdbcTokenRepositoryImpl），在配置 rememberMeConfigurer 时，配置一个 tokenRepository(PersistentTokenRepository) 即可，它会自动为我们配置 rememberMeServices 的，因为Spring Security就俩那实现，如果你配置了 PersistentTokenRepository，实质就默认你是使用了 PersistentTokenBasedRememberMeServices

OAuth2.0

OAuth全称【Open Authorization】是一个开放的授权标准，允许用户让第三方应用访问该用户在本应用中的数据，而无需将账号密码提供给第三方应用，OAuth通过颁发令牌的方式进行授权。所以OAuth是安全的。

每一个令牌授权一个特定网站在特定时间访问特定资源，OAuth 让用户可以授权第三方网站访问他们存储在另外服务提供者的某些特定信息，而非所有内容。为简化客户端开发提供了特定的授权流，包括 Web 应用、桌面应用、移动端应用等。

其中OAuth2.0是OAuth的延续版本，并不向前兼容，完全废弃了OAuth1.0

OAuth2.0中的角色

角色	作用
Authorization Server	认证服务器。用于认证用户，颁发token。如果客户端认证通过，则发放访问资源服务器的令牌
Resource Server	资源服务器。拥有受保护资源，对非法请求拦截，对请求token解析。如果请求包含正确的访问令牌，则可以访问资源
Client	客户端。它请求资源服务器时，会带上访问令牌，从而成功访问资源
Resource Owner	资源拥有者。最终用户，他有访问资源的账号与密码（指的是用户）

实现流程

- 申请对应认证服务器的OAuth权限
- 引入 OAuth客户端依赖
- 实现OAuth2.0授权
    - 获取第三方应用的授权码
    - 根据授权码，换取用户的信息

申请接入OAuth2.0

QQ：QQ互联，创建应用，需要有备案域名和网站或移动应用

微信：在微信开放平台上创建应用，需要有备案域名

还有钉钉、微博、github等各种各样的第三方平台提供了OAuth2.0授权认证，此处我们以gitee为例，因为它可以输入本地url路径，无需审核方便调用和测试。其他平台道理相同，无非换一套接口就可以了。

gitee上实现OAuth

首先看一下gitee的三方授权登录流程，此流程在giteeAPI手册中有体现，提供了`授权码模式`和`密码模式`，此处我们使用的是授权码模式

- 首先访问gitee的认证服务器，获取授权码
- 再通过授权码访问应用服务器，获取用户的首先信息
- 获取用户授权信息需要调用应用服务器，此时在我们的回调中实现

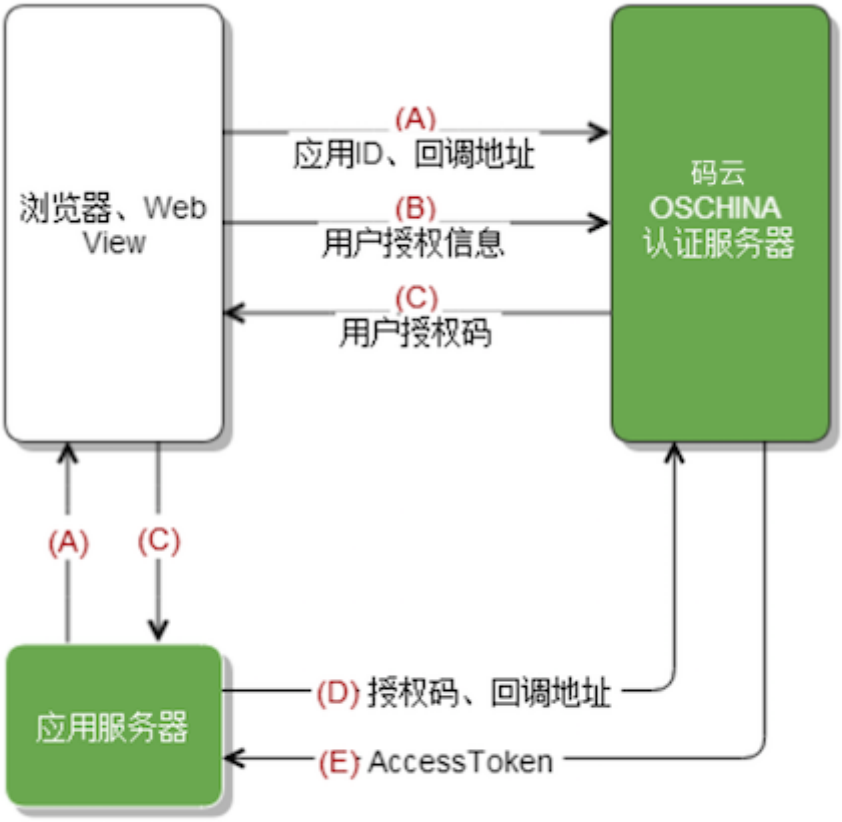


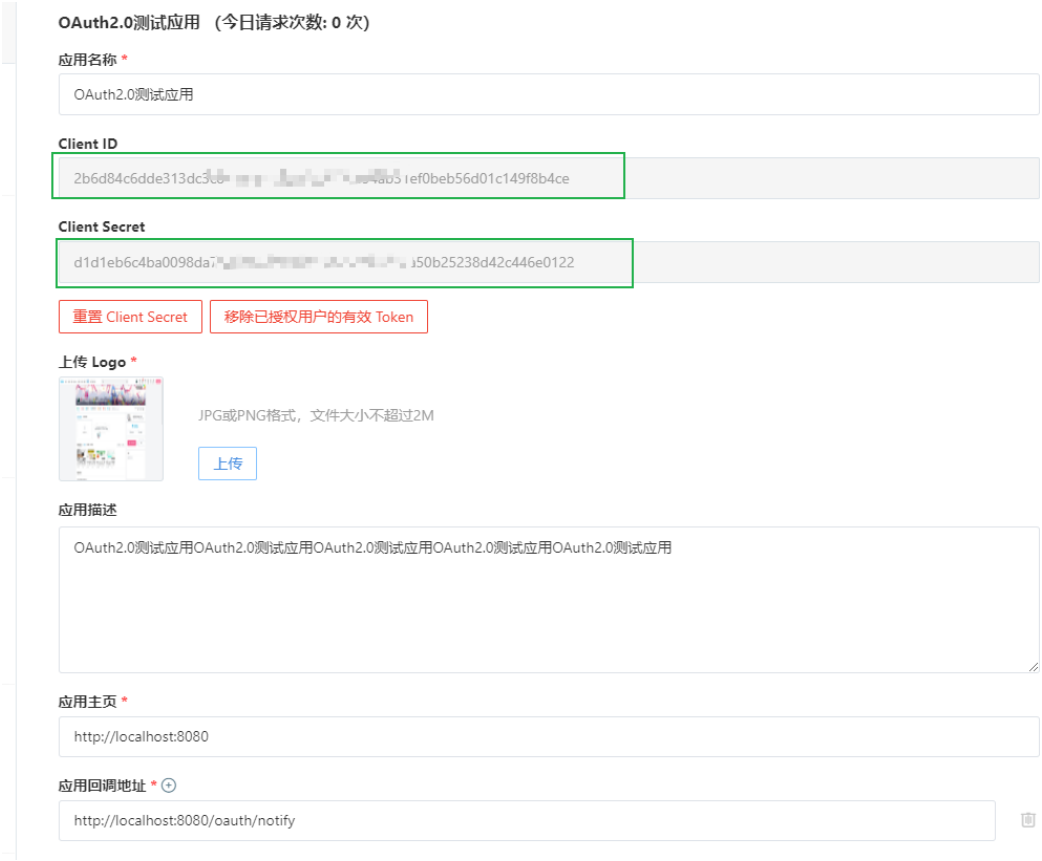
图　OAuth2 获取 AccessToken

> 此处会通过SpringSecurity默认的登录页面和自定义登录页面两种方式实现OAuth2.0的授权认证

首先创建应用

创建后获取应用信息

再点击【第三方应用】回到列表页，点击刚刚创建的应用，获取到里边的`clientId`和`secrect`，写代码要用



基于默认页面

导入依赖

导入oauth2客户端依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

yml配置

```
spring:
  security:
    oauth2:
      client:
        registration:
        # OAuth授权平台
          gitee:
            client-id: 你的clientId
            client-secret: 你的secret
            # 授权模式，固定为 authorization_code，其他值可以参考
AuthorizationGrantType
            authorization-grant-type: authorization_code
            # 创建应用时设置的回调地址，接收授权码
            redirect-uri: http://localhost:8080/oauth/notify
```

```
            # 权限范围，可配置项在码云应用信息中查看
            scope:
              - user_info # 个人用户信息，和创建的应用中勾选的权限名一致
        provider:
          gitee:
            # 申请授权地址
            authorization-uri: https://gitee.com/oauth/authorize
            # 获取访问令牌地址
            token-uri: https://gitee.com/oauth/token
            # 查询用户信息地址
            user-info-uri: https://gitee.com/api/v5/user
            # 码云用户信息中的用户名字段
            user-name-attribute: name
```

获取用户信息

此处就是我们自己编写的回调接口，接收code

```
@GetMapping("oauth/notify")
public void oauthNotify(@RequestParam(name = "code") String code) {
    // 获取token
    Map<String ,Object> map = new HashMap<>();
    map.put("grant_type","authorization_code");
    map.put("code",code);

 map.put("client_id","14049509bafee62b5e8882e459d3ede580af7a952a525faed6757475bfa5b841");
    map.put("redirect_uri","http://localhost:8080/oauth/notify");

 map.put("client_secret","1011743c66551e7838a68e28bc9932965d539d110ab56c47eb3230d3dc5fba59");
    // 通过code获取access_token
    String post = HttpUtil.post("https://gitee.com/oauth/token", map);
    GiteeOAuthLoginVO loginVO = JSONUtil.toBean(post, GiteeOAuthLoginVO.class);
    // 获取用户信息
    String userInfoStr = HttpUtil.get("https://gitee.com/api/v5/user?access_token=" + loginVO.getAccessToken());
    // 将用户信息转换为实体
}
```

启用OAuth2.0

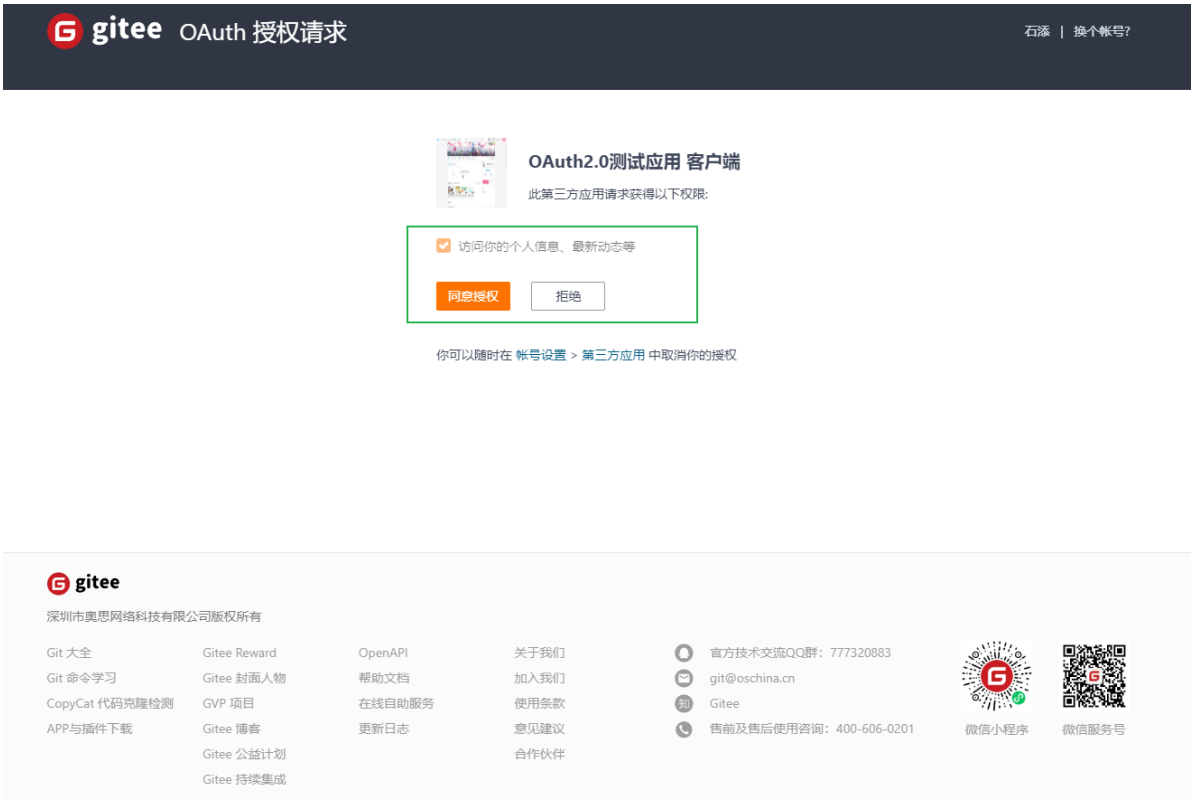
```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    // 关闭 CSRF
    http.csrf((csrf)->csrf.disable());
    // authorizeHttpRequests：指定多个授权规则，按照顺序
    http.authorizeHttpRequests((req)->
req.requestMatchers("/oauth/notify").permitAll().anyRequest().authenticated());
    // 表单登录
    http.formLogin(Customizer.withDefaults());
    // 开启 OAuth2 登录
    http.oauth2Login(Customizer.withDefaults());
    return http.build();
}
```

访问8080接口

点击之后就会跳转到gitee的授权页面，点击登陆即可

如果未登录过gitee，则先登录gitee

如果登录了gitee则，直接可看到要获取的权限，点击同意授权即可



基于自定义页面

此时通过Thymeleaf实现，没有前后端分离，引入thymeleaf依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

删除掉在yml文件中对oauth的配置，我们要自己实现，在配置类中通过注册`ClientRegistrationRepository`对象实现

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    // 配置，启用rememberMe功能
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        // 关闭csrf
        http.csrf(csrf -> csrf.disable());
        // 配置请求拦截策略
        http.authorizeHttpRequests(auth ->
auth.requestMatchers("/oauth/notify","/to_login").permitAll().anyRequest().authe
nticated());
        // 使用SpringSecurity默认的登录页面
        http.formLogin(Customizer.withDefaults());
        // 开启oauth登陆
        http.oauth2Login(Customizer.withDefaults());
        return http.build();
    }
```

```
    @Bean
    public ClientRegistrationRepository clientRegistrationRepository() {
        return new
InMemoryClientRegistrationRepository(this.giteeClientRegistration());
    }
    // 配置giee的授权登录信息
    private ClientRegistration giteeClientRegistration() {
        return ClientRegistration.withRegistrationId("gitee")
                .clientId("clientId")
                .clientSecret("secret")

.clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
                // 授权模式

.authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
                .redirectUri("http://localhost:8080/login/oauth2/code/gitee")
                // 授权范围，即创建应用时勾选的权限名
                .scope("user_info")
                .authorizationUri("https://gitee.com/oauth/authorize")
                .tokenUri("https://gitee.com/oauth/token")
                // 获取用户信息的接口
                .userInfoUri("https://gitee.com/api/v5/user")
                .userNameAttributeName("name")
                .build();
    }
}
```

再编写Thymeleaf页面，这个页面要放在`resources/templates`目录下

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>登陆页面</title>
</head>
<body>
    <!-- 这个链接从gitee的OAuth文档上获取的 需要修改为自己的clientId和回调地址-->
    <a href="https://gitee.com/oauth/authorize?
             client_id={clientId}&
             redirect_uri={你的回调地址}&
             response_type=code">gitee</a>
</body>
</html>
```

其他代码不需要修改，此时你就获取到用户的授权信息，可以将授权信息存到自己的数据库中，当做用户在你系统内的默认数据了。

退出登录

就是访问退出登录接口之后，需要做什么事情：

- 清除cookie
- 认证信息需要删除【redis】

- 记录用户的退出日志
- 跳转页面【前端做的，后端返回状态就可以了】

实现 `LogoutSuccessHandler` 做相关的逻辑操作

```
package com.stt.springsecuritydemo10.handler;

import cn.hutool.json.JSONUtil;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.extern.slf4j.Slf4j;
import org.springframework.security.core.Authentication;
import
org.springframework.security.web.authentication.logout.LogoutSuccessHandler;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.util.HashMap;

@Component
@Slf4j
public class MyLogoutSuccessHandler implements LogoutSuccessHandler {

    @Override
    public void onLogoutSuccess(HttpServletRequest request, HttpServletResponse
response, Authentication authentication) throws IOException, ServletException {
        log.info("退出登录========》");
        HashMap<String, Object> map = new HashMap<>();
        map.put("code",200);
        map.put("msg","退出登录成功");
        response.setCharacterEncoding("UTF-8");
        response.getWriter().write(JSONUtil.toJsonStr(map));
    }
}
```

配置

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private MyLogoutSuccessHandler logoutSuccessHandler;

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        // 配置退出登录
        http.logout(logout -> logout.logoutSuccessHandler(logoutSuccessHandler)
                    .deleteCookies("rememberMe"));
        return http.build();
    }
}
```

B站搜索【石添的编程哲学】