集成数据库实现认证和授权

流程

- 提供数据表：单表
- 创建Maven项目
  - 引入相关依赖
  - 配置mysql
- 实体类
- mapper和service
- controller：提供登陆接口
- 配置SpringSecurity

数据表

```
CREATE TABLE `ums_sys_user` (
  `id` bigint NOT NULL AUTO_INCREMENT COMMENT '用户ID',
  `username` varchar(30) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT
NULL COMMENT '用户账号',
  `nickname` varchar(30) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT
NULL COMMENT '用户昵称',
  `email` varchar(50) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT
'' COMMENT '用户邮箱',
  `mobile` varchar(11) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT
'' COMMENT '手机号码',
  `sex` int DEFAULT '0' COMMENT '用户性别（0男 1女 2未知）',
  `avatar` varchar(100) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT
'' COMMENT '头像地址',
  `password` varchar(100) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci
DEFAULT '' COMMENT '密码',
  `status` int DEFAULT '0' COMMENT '帐号状态（0正常 1停用）',
  `creator` bigint DEFAULT '1' COMMENT '创建者',
  `create_time` datetime DEFAULT NULL COMMENT '创建时间',
  `updater` bigint DEFAULT '1' COMMENT '更新者',
  `update_time` datetime DEFAULT NULL COMMENT '更新时间',
  `remark` varchar(500) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci DEFAULT
NULL COMMENT '备注',
  `deleted` tinyint DEFAULT '0',
  PRIMARY KEY (`id`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci ROW_FORMAT=DYNAMIC COMMENT='后台用户表';
```

创建实体类

```
package com.stt.springsecuritydemo4.domain.entity;
```

```java
import com.baomidou.mybatisplus.annotation.TableId;
import com.baomidou.mybatisplus.annotation.TableLogic;
import com.baomidou.mybatisplus.annotation.TableName;
import lombok.Data;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import java.io.Serializable;
import java.time.LocalDateTime;
import java.util.Collection;

// @Data注解可以自动生成getter、setter、无参构造
// SpringSecurity会将认证的用户信息存储到UserDetails中
@Data
@TableName("ums_sys_user")
public class UmsSysUser implements Serializable, UserDetails {

    @TableId
    private Long id;
    private String username;
    private String nickname;
    private String email;
    private Integer sex;
    private String avatar;
    private String password;
    private Integer status;
    private Long creator;
    private Long updater;
    private LocalDateTime createTime;
    private LocalDateTime updateTime;
    @TableLogic
    private Integer deleted;
    private String remark;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return null;
    }

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

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

    @Override
    public boolean isAccountNonExpired() {
        return status == 0;
    }

    @Override
    public boolean isAccountNonLocked() {
```

```
        return status == 0;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return status == 0;
    }

    @Override
    public boolean isEnabled() {
        return status == 0;
    }
}
```

认证流程

认证实现流程

- 创建一个UserDetailsService实现SpringSecurity的UserDetailsService接口
  - 写的是查询用户的逻辑
- 通过配置类对AuthenticationManager与自定义的UserDetailsService进行关联
  - SpringSecurity是通过AuthenticationManager实现的认证，会判断用户名和密码对不对
- 在登录方法所在的类中注入AuthenticationManager，调用authenticate实现认证逻辑
- 认证之后返回认证后的用户信息

UserDetailsService

```
package com.stt.springsecuritydemo4.web;

import com.baomidou.mybatisplus.core.conditions.query.LambdaQueryWrapper;
import com.stt.springsecuritydemo4.domain.entity.UmsSysUser;
import com.stt.springsecuritydemo4.mapper.UmsSysUserMapper;
import lombok.extern.slf4j.Slf4j;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service
@Slf4j
public class UmsSysUserDetailsService implements UserDetailsService {

    private final UmsSysUserMapper sysUserMapper;

    public UmsSysUserDetailsService(UmsSysUserMapper sysUserMapper) {
        this.sysUserMapper = sysUserMapper;
    }

    /**
```

```
     * 根据用户名查询用户：如果没有查到用户会抛出异常 UsernameNotFoundException【用户名不存在】
     * 返回：UserDetails，SpringSecurity定义的类，用来存储用户信息
     * UmsSysUser：实现了UserDetails接口了，根据多态，它就是一个UserDetails
     * @throws UsernameNotFoundException
     */
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        log.info("loadUserByUsername=========>{}",username);
        UmsSysUser umsSysUser = sysUserMapper.selectOne(new LambdaQueryWrapper<UmsSysUser>().eq(UmsSysUser::getUsername, username));
        log.info("loadUserByUsername=====umsSysUser====>{}",umsSysUser);
        // TODO 后期可以查看权限，角色等等
        return umsSysUser;
    }
}
```

Controller

```
package com.stt.springsecuritydemo4.controller;

import com.stt.springsecuritydemo4.domain.dto.LoginParams;
import com.stt.springsecuritydemo4.serivice.IUmsSysUserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("auth")
public class AuthController {

    private final IUmsSysUserService sysUserService;

    public AuthController(IUmsSysUserService sysUserService) {
        this.sysUserService = sysUserService;
    }

    /**
     * 登录方法：返回一个令牌
     * 用户再次访问时，在请求头 header：携带token
     */
    @PostMapping("login")
    public String login(@RequestBody LoginParams loginParams) {
        String token = sysUserService.login(loginParams);
        return token;
    }
}
```

SysUserServiceImpl

```java
package com.stt.springsecuritydemo4.serivice.impl;

import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.stt.springsecuritydemo4.domain.dto.LoginParams;
import com.stt.springsecuritydemo4.domain.entity.UmsSysUser;
import com.stt.springsecuritydemo4.mapper.UmsSysUserMapper;
import com.stt.springsecuritydemo4.serivice.IUmsSysUserService;
import lombok.extern.slf4j.Slf4j;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.BadCredentialsException;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.stereotype.Service;
import java.util.UUID;

@Service
@Slf4j
public class UmsSysUserServiceImpl extends ServiceImpl<UmsSysUserMapper,
UmsSysUser> implements IUmsSysUserService {

    private final AuthenticationManager authenticationManager;

    public UmsSysUserServiceImpl(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }

    /**
     * 这个认证就需要SpringSecurity帮我们实现了
     * @param loginParams
     * @return
     */
    @Override
    public String login(LoginParams loginParams) {
        // 传入用户名和密码 将是否认证标记设置为false
        UsernamePasswordAuthenticationToken authentication =
                new
UsernamePasswordAuthenticationToken(loginParams.getUsername(),
loginParams.getPassword());

        // 实现登录逻辑，此时就会去调用 loadUserByUsername方法
        // 返回的 Authentication 其实就是 UserDetails
        Authentication authenticate  = null;
        try{
            authenticate = authenticationManager.authenticate(authentication);
        }catch (AuthenticationException e) {
            log.error("用户名或密码错误！");
            // TODO 抛出一个业务异常
            return "用户名或密码错误！";
        }
        // 获取返回的用户
        UmsSysUser umsSysUser = (UmsSysUser) authenticate.getPrincipal();
        // 生成一个token，返回给前端
        String token = UUID.randomUUID().toString().replaceAll("-", "");
```

```
        log.info("登陆后的用户==========》{}",umsSysUser);
        return token;
    }
}
```

配置类

```
package com.stt.springsecuritydemo4.config;

import com.stt.springsecuritydemo4.web.UmsSysUserDetailsService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.ProviderManager;
import
org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecuri
ty;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private UmsSysUserDetailsService sysUserDetailsService;

    /**
     * 配置过滤器链，对login接口放行
     */
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.csrf(csrf -> csrf.disable());
        // 放行login接口
        http.authorizeHttpRequests(auth ->
auth.requestMatchers("/auth/**").permitAll()
                .anyRequest().authenticated()
        );
        return http.build();
    }

    /**
     * AuthenticationManager：负责认证的
     * DaoAuthenticationProvider：负责将 sysUserDetailsService、passwordEncoder融合
起来送到AuthenticationManager中
     * @param passwordEncoder
     * @return
     */
```

```
    @Bean
    public AuthenticationManager authenticationManager(PasswordEncoder
passwordEncoder) {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(sysUserDetailsService);
        // 关联使用的密码编码器
        provider.setPasswordEncoder(passwordEncoder);
        // 将provider放置进 AuthenticationManager 中,包含进去
        ProviderManager providerManager = new ProviderManager(provider);

        return providerManager;
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

认证原理

SpringSecurity的认证是通过AuthenticationManager的authenticate方法实现，该方法接收一个Authentication对象，通过也返回一个Authentication对象，Authentication中存储用户的主体【账号】、密码、权限等信息。

同时AuthenticationManager是一个接口，上述的例子是通过他的常用实现类ProviderManager实现的，所以看明白ProviderManager的authenticate方法就可以了。

```
UsernamePasswordAuthenticationToken authentication =
                new
UsernamePasswordAuthenticationToken(loginParams.getUsername(),
loginParams.getPassword());
```

首先上述代码是构建一个Authentication对象。通过UsernamePasswordAuthenticationToken这个实现类。该类主要是将用户名和密码读取进来，并进行存储，并设置认证标记为false。具体的认证代码如下：

```
 Authentication authenticate =
authenticationManager.authenticate(authentication);
```

首先进入`ProviderManager`类中，执行authenticate方法，方法最后将用户返回

ProviderManager.java

```
165    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
166        Class<? extends Authentication> toTest = authentication.getClass();
167        AuthenticationException lastException = null;
168        AuthenticationException parentException = null;
169        Authentication result = null;
170        Authentication parentResult = null;
171        int currentPosition = 0;
172        int size = this.providers.size();
173        for (AuthenticationProvider provider : getProviders()) {
174            if (!provider.supports(toTest)) {
175                continue;
176            }
177            if (logger.isTraceEnabled()) {
178                logger.trace(LogMessage.format( format: "Authenticating request with %s (%d/%d)",
179                        provider.getClass().getSimpleName(), ++currentPosition, size));
180            }
181            try {
182                result = provider.authenticate(authentication);
183                if (result != null) {
184                    copyDetails(authentication, result);
185                    break;
186                }
187            }
188            catch (AccountStatusException | InternalAuthenticationServiceException ex) {
189                prepareException(ex, authentication);
190                // SEC-546: Avoid polling additional providers if auth failure is due to
191                // invalid account status
192                throw ex;
193            }
194            catch (AuthenticationException ex) {
195                lastException = ex;
196            }
197        }
198        if (result == null && this.parent != null) {
```

获取Authentication的类型，此处为UserPassword.....Token

获取AuthenticationProvider，此处为配置类中配置的DaoAuthenticationProvider

认证用户逻辑，传进去一个Authentication，返回一个Authentication，
如果认证成功，修改认证标记为true

接下来看一下provider.authenticate()方法的执行，具体执行的是`AbstractUserDetailsAuthenticationProvider`类中的方法

AbstractUserDetailsAuthenticationProvider.java

```
121
122    @Override
123    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
124        Assert.isInstanceOf(UsernamePasswordAuthenticationToken.class, authentication,
125                () -> this.messages.getMessage( code: "AbstractUserDetailsAuthenticationProvider.onlySupports",
126                        defaultMessage: "Only UsernamePasswordAuthenticationToken is supported"));
127        String username = determineUsername(authentication);
128        boolean cacheWasUsed = true;
129        UserDetails user = this.userCache.getUserFromCache(username);
130        if (user == null) {
131            cacheWasUsed = false;
132            try {
133                user = retrieveUser(username, (UsernamePasswordAuthenticationToken) authentication);
134            }
135            catch (UsernameNotFoundException ex) {
136                this.logger.debug("Failed to find user '" + username + "'");
137                if (!this.hideUserNotFoundExceptions) {
138                    throw ex;
139                }
140                throw new BadCredentialsException(this.messages
141                    .getMessage( code: "AbstractUserDetailsAuthenticationProvider.badCredentials", defaultMessage: "Bad credentials"));
142            }
143            Assert.notNull(user, message: "retrieveUser returned null - a violation of the interface contract");
144        }
145        try {
146            this.preAuthenticationChecks.check(user);
147            additionalAuthenticationChecks(user, (UsernamePasswordAuthenticationToken) authentication);
148        }
149        catch (AuthenticationException ex) {
150            if (!cacheWasUsed) {
151                throw ex;
152            }
153            // There was a problem, so try again after checking
154            // we're using latest data (i.e. not from the cache)
155            cacheWasUsed = false;
156            user = retrieveUser(username, (UsernamePasswordAuthenticationToken) authentication);
157            this.preAuthenticationChecks.check(user);
158            additionalAuthenticationChecks(user, (UsernamePasswordAuthenticationToken) authentication);
159        }
160        this.postAuthenticationChecks.check(user);
161        if (!cacheWasUsed) {
162            this.userCache.putUserInCache(user);
```

获取用户名

从缓存中获取用户，第一次认证是null

此处根据用户名获取用户。其实就是调用loadUserbyUsername方法
错误就抛出UsernameNotFoundException

判断用户状态，是否过期，是否锁定等

判断密码是否正确

出现异常会进行重试

下边是retrieveUser方法的逻辑，具体是执行loadUserByUsername方法，去根据用户名查找用户

```java
@Override
protected final UserDetails retrieveUser(String username, UsernamePasswordAuthenticationToken authentication)
        throws AuthenticationException {
    prepareTimingAttackProtection();
    try {
        UserDetails loadedUser = this.getUserDetailsService().loadUserByUsername(username);
        if (loadedUser == null) {
            throw new InternalAuthenticationServiceException(
                    "UserDetailsService returned null, which is an interface contract violation");
        }
        return loadedUser;
    }
    catch (UsernameNotFoundException ex) {
        mitigateAgainstTimingAttack(authentication);
        throw ex;
    }
    catch (InternalAuthenticationServiceException ex) {
        throw ex;
    }
    catch (Exception ex) {
        throw new InternalAuthenticationServiceException(ex.getMessage(), ex);
    }
}
```

调用对应UserDetailsService的loadUserByUsername方法

接下来是this.preAuthenticationChecks.check(user);代码

```java
public class AccountStatusUserDetailsChecker implements UserDetailsChecker, MessageSourceAware {

    private final Log logger = LogFactory.getLog(getClass());

    protected MessageSourceAccessor messages = SpringSecurityMessageSource.getAccessor();

    @Override
    public void check(UserDetails user) {
        if (!user.isAccountNonLocked()) {
            this.logger.debug("Failed to authenticate since user account is locked");
            throw new LockedException(
                    this.messages.getMessage( code: "AccountStatusUserDetailsChecker.locked",  defaultMessage: "User account is locked"));
        }
        if (!user.isEnabled()) {
            this.logger.debug("Failed to authenticate since user account is disabled");
            throw new DisabledException(
                    this.messages.getMessage( code: "AccountStatusUserDetailsChecker.disabled",  defaultMessage: "User is disabled"));
        }
        if (!user.isAccountNonExpired()) {
            this.logger.debug("Failed to authenticate since user account is expired");
            throw new AccountExpiredException(
                    this.messages.getMessage( code: "AccountStatusUserDetailsChecker.expired",  defaultMessage: "User account has expired"));
        }
        if (!user.isCredentialsNonExpired()) {
            this.logger.debug("Failed to authenticate since user account credentials have expired");
            throw new CredentialsExpiredException(this.messages
                .getMessage( code: "AccountStatusUserDetailsChecker.credentialsExpired",  defaultMessage: "User credentials have expired"));
        }
    }
```

检查用户是否锁定

检查用户是否可用

检查用户是否过期

检查用户密码是否锁定

下边是`additionalAuthenticationChecks(user, (UsernamePasswordAuthenticationToken) authentication);`方法

```java
protected void additionalAuthenticationChecks(UserDetails userDetails,
        UsernamePasswordAuthenticationToken authentication) throws AuthenticationException {
    if (authentication.getCredentials() == null) {
        this.logger.debug("Failed to authenticate since no credentials provided");
        throw new BadCredentialsException(this.messages
            .getMessage( code: "AbstractUserDetailsAuthenticationProvider.badCredentials",  defaultMessage: "Bad credentials"));
    }
    String presentedPassword = authentication.getCredentials().toString();
    if (!this.passwordEncoder.matches(presentedPassword, userDetails.getPassword())) {
        this.logger.debug("Failed to authenticate since password does not match stored value");
        throw new BadCredentialsException(this.messages
            .getMessage( code: "AbstractUserDetailsAuthenticationProvider.badCredentials",  defaultMessage: "Bad credentials"));
    }
}
```

获取用户输入的密码

匹配密码是否正确

整个逻辑是：

- UsernamePasswordAuthenticationToken将用户填写的用户名密码存储下来，设置认证状态为false，它就是一个Authentication类型的对象
- 调用AuthenticationManager.authenticate(Authentication authentication)进行用户认证，并返回Authentication，其中记录用户信息和认证状态
    - 首先执行loadUserByUsername方法，根据用户名获取用户
    - 再判断用户状态
    - 再判断密码是否正确
    - 如果失败则重试，最终错误抛出异常

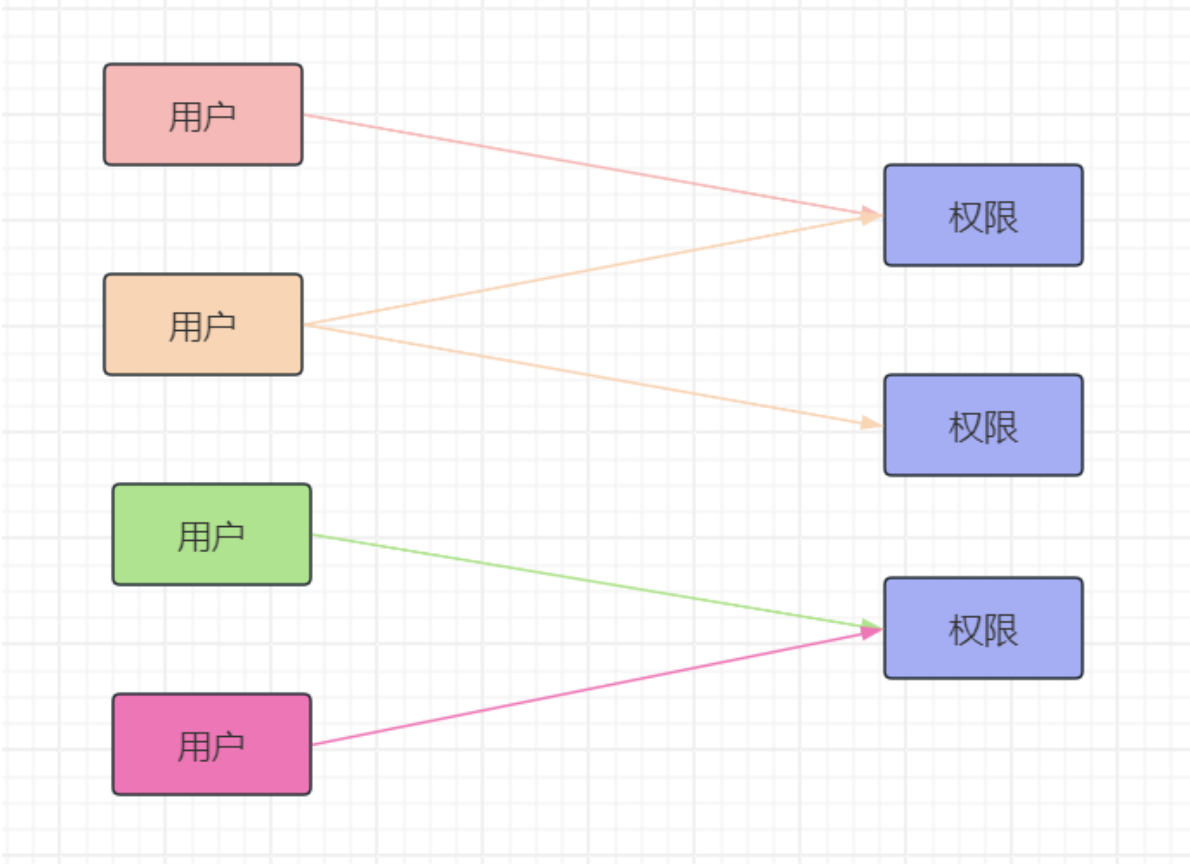
- 如果正确就返回Authentication

授权流程

此处我们介绍权限的设计和SpringSecurity权限管理，首先介绍一下比较常见的权限设计控制模型有自主访问控制（DAC）、强制访问控制（MAC）、基于角色的访问控制（RBAC）、基于属性的访问控制（ABAC）、访问控制列表（ACL）等。目前使用广泛的是RBAC权限模型。

DAC模型

允许用户自由地选择他们想要访问的资源。



以上模型的缺点在于，如果用户拥有相同的权限，新用户还需要一一关联，如果这些相同用户的权限需要修改，同样需要一一修改非常繁琐。由此引出了RBAC权限模型。

RBAC权限模型

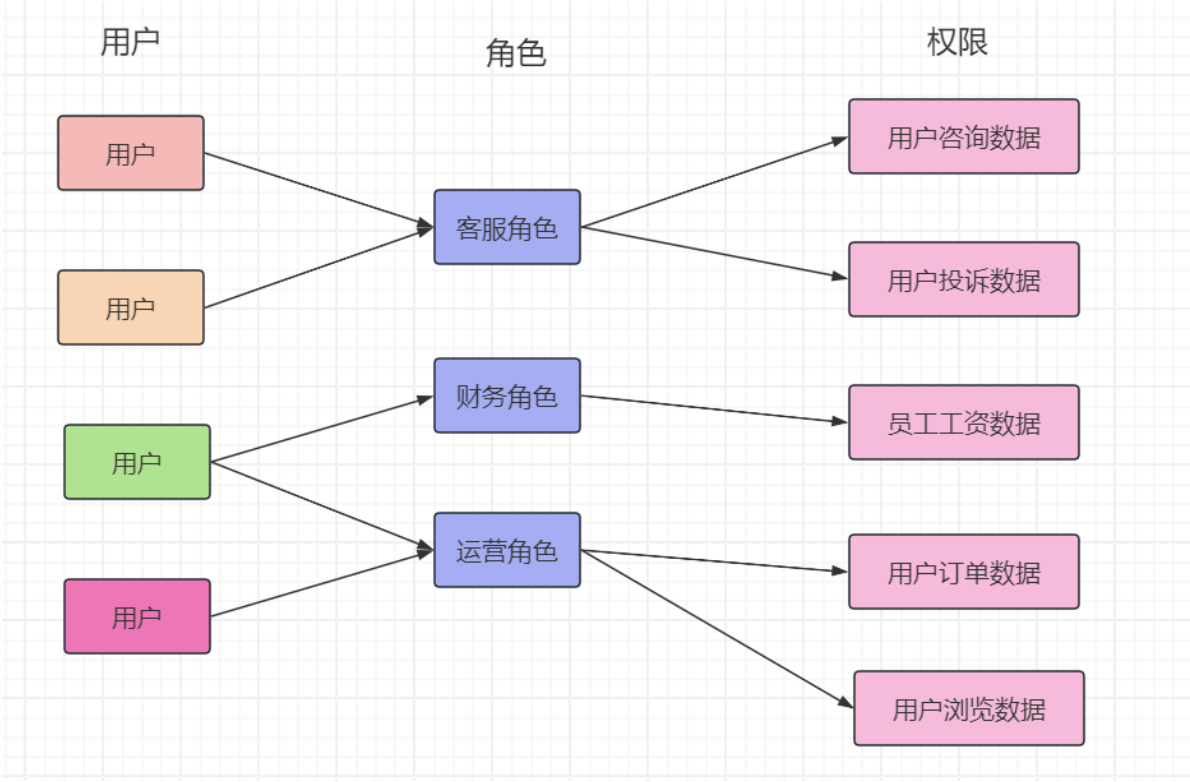
RBAC【Role-based access control】，增加了角色的概念，即基于角色的访问控制，将权限分配给角色，再将角色分配给用户。简单来说，就是【用户关联角色，角色关联权限】。

角色：一组权限的集合

用户：一组角色的集合

RBAC遵循三条安全原则：

- **最小权限原则**：给角色配置最小但能满足使用需求的权限。
- **责任分离原则**：给比较重要或者敏感的事件设置不同的角色，不同的角色间是相互约束的，由其一同参与完成。
- **数据抽象原则**：每个角色都只能访问其需要的数据，而不是全部数据，不同的角色能访问到的数据也不同。



由上图可见：

- 用户可以有多个角色，一个角色可以分配给多个用户，用户和角色是多对多关系
- 角色可以包含多个权限，一个权限也可以分配给多个角色，用户和角色也是多对多关系

如果要存储到数据库中的话表设计如下：

角色表

```
CREATE TABLE `ums_role` (
  `id` bigint NOT NULL AUTO_INCREMENT COMMENT '角色id',
  `role_label` varchar(255) DEFAULT NULL COMMENT '角色标识',
  `role_name` varchar(255) DEFAULT NULL COMMENT '角色名字',
  `sort` int DEFAULT NULL COMMENT '排序',
  `status` int DEFAULT NULL COMMENT '状态：0：可用，1：不可用',
  `deleted` int DEFAULT NULL COMMENT '是否删除：0：未删除，1：已删除',
  `remark` varchar(255) DEFAULT NULL COMMENT '备注',
  `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=2 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci;
```

权限表

```
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',
  `sort` int DEFAULT '0' COMMENT '排序',
  `menu_type` int DEFAULT NULL COMMENT '类型：0，目录，1菜单，2：按钮',
  `perms` varchar(255) DEFAULT NULL COMMENT '权限标识',
  `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=10 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci;
```

用户角色表

```
CREATE TABLE `ums_sys_user_role` (
  `id` bigint NOT NULL AUTO_INCREMENT COMMENT '主键id',
  `user_id` bigint NOT NULL COMMENT '用户id',
  `role_id` bigint NOT NULL COMMENT '角色id',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

角色权限表

```
CREATE TABLE `ums_role_menu` (
  `id` bigint NOT NULL AUTO_INCREMENT,
  `role_id` bigint DEFAULT NULL,
  `menu_id` bigint DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

SpringSecurity权限认证

SpringSecurity要求将身份认证信息存到GrantedAuthority对象列表中。代表了当前用户的权限。GrantedAuthority对象由AuthenticationManager插入到Authentication对象中，然后在做出授权决策时由AccessDecisionManager实例读取。

GrantedAuthority 接口只有一个方法

```
String getAuthority();
```

AuthorizationManager实例通过该方法来获得GrantedAuthority。通过字符串的形式表示，GrantedAuthority可以很容易地被大多数AuthorizationManager实现读取。如果GrantedAuthority不能精确地表示为String，则GrantedAuthorization被认为是复杂的，getAuthority()必须返回null

> 我们应该告诉SpringSecurity，当前登陆的用户有什么权限【登陆后使用软件的过程中，权限也可能会被修改】

告知权限的流程

- 登陆的时候需要查询用户权限，基于RBAC模型实现的权限设计
  - 先获取用户的角色，根据角色获取用户权限
  - 因为SpringSecurity识别权限的数据类型时String，所以我们需要将查询出的权限对象，封装到String类型的集合中【Set集合，数据不可重复】
- 后续操作的时候，需要携带登陆的标记【token或者cookie】，根据token获取用户的信息，在后续的操作过程中继续识别权限
  - 后续的操作尽量不要每访问一次就查一次数据库，对数据库压力很大

实体类

用户类

```
@Data
@TableName("ums_sys_user")
public class UmsSysUser implements Serializable, UserDetails {

    ......

    // 角色信息
    private Set<UmsRole> roleSet = new HashSet<>();
    //权限的信息
    private Set<String> perms = new HashSet<>();

    /**
     * SpringSecurity根据 getAuthorities 方法获取当前用户的权限信息
     * @return
     */

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        // 将权限告知SpringSecurity，通过lambda表达式将Set<String>转成
Collection<GrantedAuthority>
        if(perms != null && perms.size() > 0) {
            // 返回权限信息
            return
perms.stream().map(SimpleGrantedAuthority::new).collect(Collectors.toSet());
        }
        return null;
    }
    ......
}
```

角色类

```
@Data
@TableName("ums_role")
public class UmsRole implements Serializable {

    @TableId
    private Long roleId;
    private String roleLabel;
    private String roleName;
    private Integer sort;
```

```
    private Integer status;
    @TableLogic
    private Integer deleted;
    private String remark;
    private LocalDateTime createTime;
    private LocalDateTime updateTime;
}
```

权限类

```
@Data
@TableName("ums_menu")
public class UmsMenu implements Serializable {

    @TableId
    private Long id;
    private Long parentId;
    private String menuName;
    private Integer sort;
    private String perms;
    private Integer menuType;
    private String icon;
    @TableLogic
    private Integer deleted;
    private LocalDateTime createTime;
    private LocalDateTime updateTime;
}
```

> 记得创建对应的mapper和service

查询用户权限信息

```
@Service
@Slf4j
public class UmsSysUserDetailsService implements UserDetailsService {

    private final UmsSysUserMapper sysUserMapper;
    private final UmsMenuMapper menuMapper;

    public UmsSysUserDetailsService(UmsSysUserMapper sysUserMapper,UmsMenuMapper
menuMapper) {
        this.sysUserMapper = sysUserMapper;
        this.menuMapper = menuMapper;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {

        // 做用户信息查询，不要多次访问数据库，尽量一次查出需要的数据，多表查询不要超过3张表
        // 1、查询用户的角色信息
        UmsSysUser umsSysUser = sysUserMapper.selectUserByUsername(username);
        log.info("没有权限===>{}",umsSysUser);
        // 2、查询用户的权限信息
```

```
        if(umsSysUser != null) {
            Set<UmsRole> roleSet =  umsSysUser.getRoleSet();
            // 存储角色id，进行批量查询，不要在for循环中查询数据库
            Set<Long> roleIds = new HashSet<>(roleSet.size());
            // 获取用户的权限列表
            Set<String> perms = umsSysUser.getPerms();
            for (UmsRole umsRole : roleSet) {
                roleIds.add(umsRole.getRoleId());
            }
            // 权限查询
            Set<UmsMenu> menus = menuMapper.selectMenuByRoleId(roleIds);
            for (UmsMenu menu : menus) {
                String perm = menu.getPerms();
                // 添加用户权限到set中
                perms.add(perm);
            }
            log.info("有权限====》{}",umsSysUser);
        }
        return umsSysUser;
    }
}
```

查询角色sql

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.stt.springsecuritydemo5.mapper.UmsSysUserMapper">

    <resultMap id="SysUserResultMap"
type="com.stt.springsecuritydemo5.domain.entity.UmsSysUser">
        <id column="id" property="id"/>
        <result column="username" property="username"/>
        <result column="nickname" property="nickname"/>
        <result column="email" property="email"/>
        <result column="sex" property="sex"/>
        <result column="avatar" property="avatar"/>
        <result column="password" property="password"/>
        <result column="status" property="status"/>
        <result column="creator" property="creator"/>
        <result column="updater" property="updater"/>
        <result column="createTime" property="createTime"/>
        <result column="updateTime" property="updateTime"/>
        <result column="deleted" property="deleted"/>
        <result column="remark" property="remark"/>
        <collection property="roleSet" resultMap="RoleResultMap"/>
    </resultMap>

    <resultMap id="RoleResultMap"
type="com.stt.springsecuritydemo5.domain.entity.UmsRole">
        <id column="role_id" property="roleId"/>
        <result column="role_label" property="roleLabel"/>
        <result column="role_name" property="roleName"/>
        <result column="sort" property="sort"/>
        <result column="status" property="status"/>
```

```
        <result column="deleted" property="deleted"/>
        <result column="remark" property="remark"/>
        <result column="create_time" property="createTime"/>
        <result column="update_time" property="updateTime"/>
    </resultMap>
    <!--  根据用户名查询用户和角色信息  -->
    <select id="selectUserByUsername" resultMap="SysUserResultMap">
        select
            u.id,u.username,u.nickname,u.email,
            u.sex,
            u.avatar,
            u.password,
            u.status,
            u.creator,
            u.updater,
            u.create_time,
            u.update_time,
            u.deleted,
            u.remark,
            r.role_id,
            r.role_label,
            r.role_name,
            r.sort,
            r.status,
            r.deleted,
            r.remark,
            r.create_time,
            r.update_time
        from ums_sys_user u left join ums_sys_user_role sur on u.id =
sur.user_id
                            left join ums_role r on sur.role_id = r.role_id
        where u.deleted = 0 and r.deleted = 0 and u.username = #{username}
    </select>
</mapper>
```

查询权限sql

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.stt.springsecuritydemo5.mapper.UmsMenuMapper">

    <!--  根据角色查询权限
        1、数据在menu表中
        2、需要通过角色查询，角色和权限的关系在ums_role_menu中维护
        3、需要多表查询，关系搞清楚
        4、sql：in (1,2,3)
      -->
    <select id="selectMenuByRoleId"
resultType="com.stt.springsecuritydemo5.domain.entity.UmsMenu">
        select m.id,
        m.parent_id,
        m.menu_name,
        m.sort,
        m.perms,
```

```
        m.menu_type,
        m.icon,
        m.deleted,
        m.create_time,
        m.update_time
            from ums_menu m left join ums_role_menu urm on m.id = urm.menu_id
        where urm.role_id in
        <foreach collection="roleIds" open="(" close=")" separator="," item="roleId">
            #{roleId}
        </foreach>
    </select>
</mapper>
```

携带登录信息

HTTP协议是无状态请求，即一次会话不会记录上一次会话的内容。所以需要通过携带数据的方式告知服务器我是谁，以便服务器知道这个人有没有权限访问这个数据。最初使用cookie的方式携带

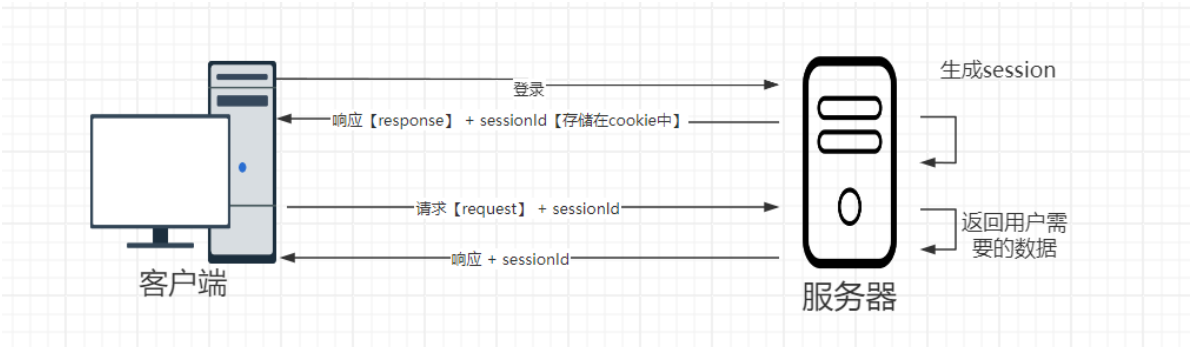
cookie

Cookie，有时也用其复数形式 Cookies。类型为“小型文本文件”，是某些网站为了辨别用户身份，进行 Session 跟踪而储存在用户本地终端上的数据（通常经过加密），由用户客户端计算机暂时或永久保存的信息 。身份信息，订单信息，浏览记录

cookie中存储用户的相关信息，但是如果信息越来越多，那么cookie就会越来越大。此时就引出了session

seesion

考虑将用户的信息存储到服务器端，请求时只携带可以识别用户身份的身份信息就可以了，再根据用户身份获取用户的其他资料比如：实名信息，订单信息，访问记录等



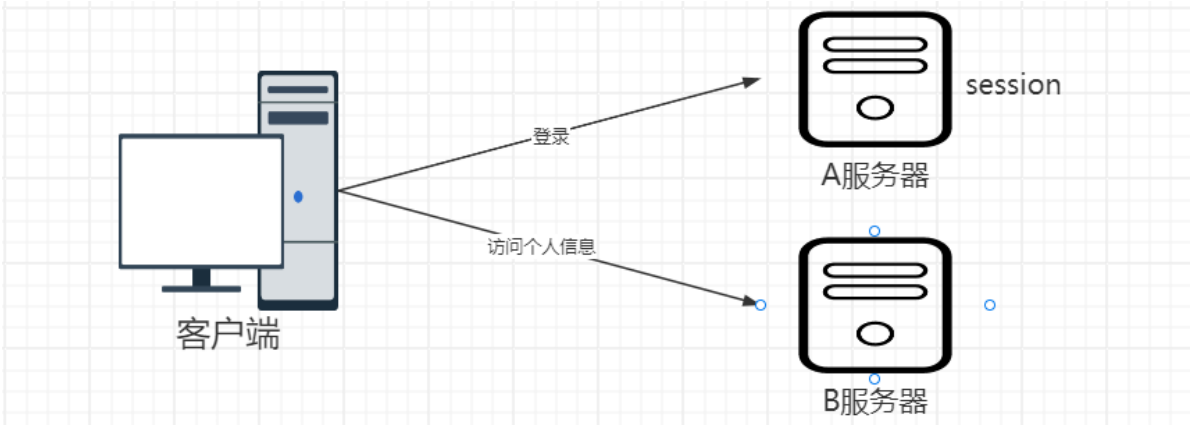
1. 首先是用户登录，服务端会生成一个session，再生成该session的唯一标识sessionId，这个sessionId可以得知是哪个用户，再将此sessionId通过cookie告知客户端

2. 之后客户端的请求都再cookie中携带这个sessionId，服务端获取sessionId，找到对应的用户，再去处理请求就可以了

session的作用就是缩小了cookie的体积，并且cookie存储在客户端，session存储在服务端。

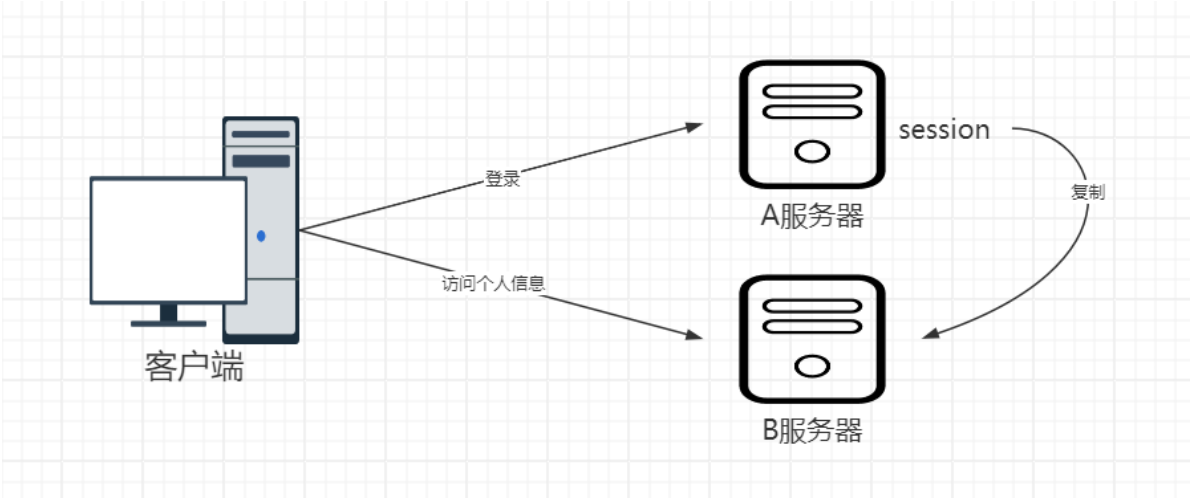
随着网民增长，软件用户增加，服务器会采用分布式，集群等方式部署，保障可以为更多用户提供服务，session就会出现问题。即

session如果在A服务器生成，后续请求如果发送到B服务器，则B服务器就无法得知是哪个用户



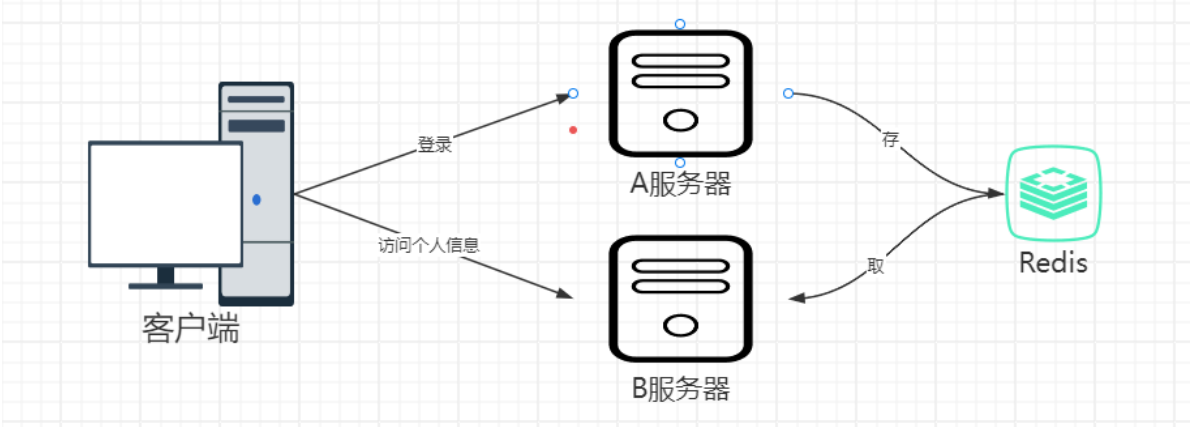
此时就可以采用：

session复制：这种方式有延迟，而且实现复杂



session共享：此方式需要频繁查询数据库

将session存储到db中，如mysql、redis等。有些开发人员就会认为查询数据库对数据库造成了不必要的压力



现象：不管用户多少，上来就想搞 微服务

token【令牌】

作用：其实就是为了知道你是谁。还有一种方式就是将用户信息加密变成字符串，直接发给客户端，客户端后边请求都携带这个加密字符串，我们称之为Token【令牌】，服务端拿到字符串之后可以解密获取到用户信息，这样就不需要查询数据库了。多个服务端加密算法相同，也就可以完成解密工作，这个生成Token的技术可以选用JWT【Json Web Tokens】。而且这个字符串我们会放到请求头【header】中，这个字段根据自己的需求定义

JWT可以将用户信息【id，username，avatar，权限。密码，支付密码千万别放】变成字符串，也可以加密。之后再次请求时携带这个字符串，可以解析为用户信息。就不用访问数据库了

重点：

- token：token是一种识别用户身份的机制，基于这种机制可以有很多种实现，就是在登陆完成之后，再请求头中添加用户标识。放到请求头中，再次访问的时候，服务器从请求头中获取token，判断用户是否是合法的
- jwt：就是生成token的一种具体实现，它可以直接将用户信息存储到字符串中，也可以根据字符串解析出用户信息，不需要再查询数据库了。也支持加密

JWT结构

JWT是一个很长的字符串，由三部分组成。但是JWT内部并没有换行，而是由.隔开

```
eyJhbGciOiJIUzI1NiJ9.
eyJpZCI6MTAwMCwiYXZhdGFyIjoi5p2p5qyQ5qe45raT77-95raT7oGE44GU6Y2N5b-T5rm06Y2n77-
9IiwidXNlcm5hbWUiOiLlr67nirHnrIEifQ.
9sIcaoCsaT-WXMqLSPVtFHj_zKh6OpwEboUF_Qit6G4
```

- Header（头部）
- Payload（负载）
- Signature（签名）

Header

Header 部分是一个 JSON 对象，描述 JWT 的元数据，通常是下面的样子

```
{
    "alg": "HS256",
    "typ": "JWT"
}
```

alg：表示签名使用的算法，默认为 HMAC SHA256（写为HS256）

typ：表示令牌的类型，JWT 令牌统一写为JWT。

最后，使用 Base64 URL 算法将上述 JSON 对象转换为字符串保存。

Payload

Payload 部分也是一个 JSON 对象，用来存放实际需要传递的数据。JWT 规定了7个官方字段，供选用

- iss (issuer)：签发人/发行人
- sub (subject)：主题
- aud (audience)：用户

- exp (expiration time)：过期时间
- nbf (Not Before)：生效时间，在此之前是无效的
- iat (Issued At)：签发时间
- jti (JWT ID)：用于标识该 JWT

除了官方字段，你还可以在这个部分定义私有字段，下面就是一个例子。

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

注意，JWT 默认是不加密的，任何人都可以读到，所以不要把秘密信息放在这个部分。

这个 JSON 对象也要使用 Base64URL 算法转成字符串。

Signature

Signature 部分是对前两部分的签名，防止数据篡改。

首先，需要指定一个密钥（secret）。这个密钥只有服务器才知道，不能泄露给用户。然后，使用 Header 里面指定的签名算法（默认是 HMAC SHA256），按照下面的公式产生签名。

算出签名以后，把 Header、Payload、Signature 三个部分拼成一个字符串，每个部分之间用"点"（.）分隔，就可以返回给用户。

Base64URL

前面提到，Header 和 Payload 串型化的算法是 Base64URL。这个算法跟 Base64 算法基本类似，但有一些小的不同。

JWT 作为一个令牌（token），有些场合可能会放到 URL（比如 api.example.com/?token=xxx）。Base64 有三个字符+、/和=，在 URL 里面有特殊含义，所以要被替换掉：=被省略、+替换成-，/替换成 _ 。这就是 Base64URL 算法

引入jwt

引入依赖

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
<!-- JDK8以上需要加入以下依赖 -->
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.1</version>
</dependency>
```

JWT两个核心操作

- 更具用户信息，生成字符串当做token
- 根据token解析出用户信息

```
package com.stt.springsecuritydemo6.web;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.springframework.stereotype.Component;
import javax.xml.crypto.Data;
import java.util.Date;
import java.util.Map;

@Component
public class JwtUtils {

    private String secret = "qwertyuioplkjnbvfdcxsaz";
    /**
     * 生成token
     */
    public String createToken(Map<String,Object> map) {
        String token = Jwts.builder()
                .setClaims(map)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + 30 * 60 *
1000))
                .signWith(SignatureAlgorithm.HS256, secret)
                .compact();
        return token;
    }

    /**
     * 根据token解析出用户信息
     */
    public Claims parseToken(String token) {
        // 解析token，需要使用和创建token时相同的秘钥
        Claims claims = Jwts.parser().setSigningKey(secret)
                .parseClaimsJws(token)
                .getBody();
        return claims;
    }
}
```

鉴权

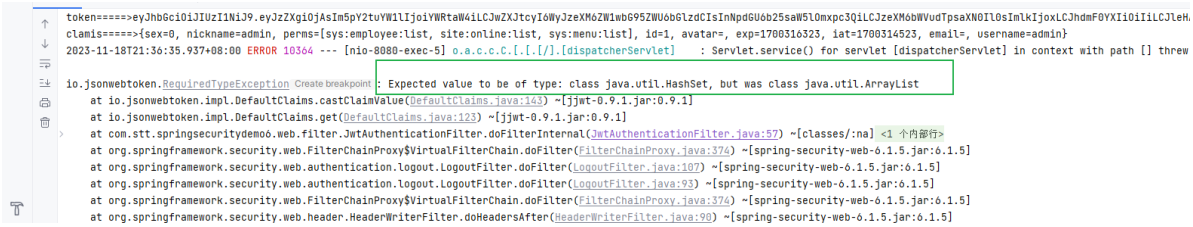
访问其他接口的时候携带token，在接口上添加权限校验【基于方法的权限校验】。我们就需要在请求时获取请求头的token字段，我们需要自定义过滤器，并且将过滤器添加到SpringSecurity的过滤器链中。【使用了责任链模式】

自定义过滤器

```
package com.stt.springsecuritydemo6.web.filter;

import com.stt.springsecuritydemo6.domain.entity.UmsSysUser;
import com.stt.springsecuritydemo6.web.JwtUtils;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.SignatureException;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

/**
 * 捕获请求中的请求头，获取token字段，判断是否可以获取用户信息
 * 我们可以继承 OncePerRequestFilter 抽象类
 *
 * 1、获取到用户信息之后，需要将用户的信息告知SpringSecurity，SpringSecurity会去判断你访问
的接口是否有相应的权限
 * 2、告知SpringSecurity 就是使用Authentication告知框架，SpringSecurity、会将信息存储到
SecurityContext中-----》SecurityContextHolder中
 *
 * 登录的时候，放置的数据是用户名和密码。是要查找用的
 * 后边请求，判断权限的时候，放置进去的数据是用户的信息。密码就不需要了，还有用户的权限
 */
@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtils jwtUtils;

    /**
     * 该方法会被doFilter调用
     */
    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain) throws ServletException,
IOException {
        // 获取token
        String token = request.getHeader("Authorization");
        System.out.println("token======>" + token);
        // login请求就没token，直接放行，因为后边有其他的过滤器
        if(token == null) {
            doFilter(request,response,filterChain);
```

```
            return;
        }
        // 有token，通过jwt工具类，解析用户信息
        Claims claims = null;
        try {
            claims = jwtUtils.parseToken(token);
        }catch (SignatureException e){
            // 需要返回401，重新登陆
            response.setCharacterEncoding("UTF-8");
            response.getWriter().write("验签失败！！！");
            return;
        }
        System.out.println("claims======>" + claims);
        // 获取到了数据，将数据取出，放到UmsSysUser中
        Long id = claims.get("id", Long.class);
        String username = claims.get("username", String.class);
        String avatar = claims.get("avatar", String.class);
        List<String> perms = claims.get("perms", ArrayList.class);
        // 将信息放到User类中
        UmsSysUser umsSysUser = new UmsSysUser();
        umsSysUser.setId(id);
        umsSysUser.setUsername(username);
        umsSysUser.setAvatar(avatar);
        umsSysUser.setPerms(perms);
        System.out.println("umsSysUser======>" + umsSysUser);
        // 将用户信息放到SecurityContext中
        UsernamePasswordAuthenticationToken authentication =
                new UsernamePasswordAuthenticationToken(umsSysUser, null,
umsSysUser.getAuthorities());
        SecurityContextHolder.getContext().setAuthentication(authentication);
        // 放行
        doFilter(request,response,filterChain);
    }
}
```

jwt解析数据时，集合类型，会转换为ArrayList



添加过滤器

```
package com.stt.springsecuritydemo6.config;

import com.stt.springsecuritydemo6.web.UmsSysUserDetailsService;
import com.stt.springsecuritydemo6.web.filter.JwtAuthenticationFilter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
```

```
import org.springframework.security.authentication.ProviderManager;
import
org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import
org.springframework.security.config.annotation.method.configuration.EnableMethod
Security;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecuri
ty;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFi
lter;

@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {

    @Autowired
    private JwtAuthenticationFilter jwtAuthenticationFilter;
    @Autowired
    private UmsSysUserDetailsService sysUserDetailsService;

    /**
     * 配置过滤器链，对login接口放行
     */
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.csrf(csrf -> csrf.disable());
        // 放行login接口
        http.authorizeHttpRequests(auth ->
auth.requestMatchers("/auth/**").permitAll()
                .anyRequest().authenticated()
        );
        // 将过滤器添加到过滤器链中
        // 将过滤器添加到 UsernamePasswordAuthenticationFilter 之前
        http.addFilterBefore(jwtAuthenticationFilter,
UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }

    /**
     * AuthenticationManager：负责认证的
     * DaoAuthenticationProvider：负责将 sysUserDetailsService、passwordEncoder融合
起来送到AuthenticationManager中
     * @param passwordEncoder
     * @return
     */
    @Bean
    public AuthenticationManager authenticationManager(PasswordEncoder
passwordEncoder) {
```

```
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(sysUserDetailsService);
        // 关联使用的密码编码器
        provider.setPasswordEncoder(passwordEncoder);
        // 将provider放置进 AuthenticationManager 中,包含进去
        ProviderManager providerManager = new ProviderManager(provider);

        return providerManager;
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

思考：

JWT有什么问题？

- **强制过期**：JWT就做不到，可以设置过期时间，但是不能强制过期。不能强制用户下线

JWT生成token---》用户信息存储到redis中。只需要删除redis中的用户信息这个token就相当于失效了

安全性

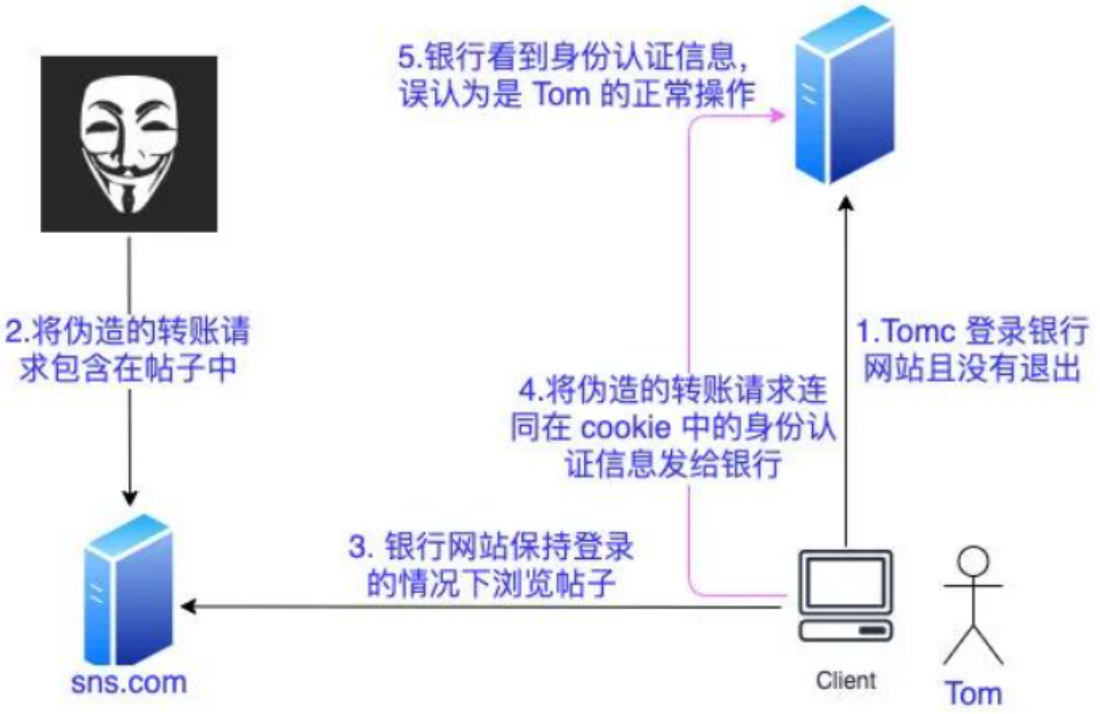
CSRF攻击

攻击者通过一些技术手段欺骗用户的浏览器去访问一个自己曾经认证过的网站并运行一些操作（如发邮件，发消息，甚至财产操作如转账和购买商品）。由于浏览器曾经认证过（cookie 里带来 sessionId 等身份认证的信息），所以被访问的网站会认为是真正的用户操作而去运行。

比如用户登录了某银行网站（假设为 http://www.examplebank.com/，并且转账地址为 http://www.examplebank.com/withdraw?amount=1000&transferTo=PayeeName），登录后 cookie 里会包含登录用户的 sessionid，攻击者可以在另一个网站上放置如下代码

```
<img src="http://www.examplebank.com/withdraw?
account=Alice&amount=1000&for=Badman">
```

那么如果正常的用户误点了上面这张图片，由于相同域名的请求会自动带上 cookie，而 cookie 里带有正常登录用户的 sessionid，类似上面这样的转账操作在 server 就会成功，会造成极大的安全风险



CSRF 攻击的根本原因在于对于同样域名的每个请求来说，它的 cookie 都会被自动带上，这个是浏览器的机制决定的，所以很多人据此认定 cookie 不安全。

使用 token 确实避免了CSRF 的问题，但正如上文所述，由于 token 保存在 local storage，它会被 JS 读取，**从存储角度来看**也不安全（实际上防护 CSRF 攻击的正确方式是用 CSRF token）

所以不管是 cookie 还是 token，从存储角度来看其实都不安全，都有暴露的风险，我们所说的安全更多的是强调传输中的安全，可以用 HTTPS 协议来传输， 这样的话请求头都能被加密，也就保证了传输中的安全。

如果有人问cookie和token有什么区别：