PROJECTS
SHOW

Spring Security & Spring Session

https://spring.io/projects/spring-session-core

https://spring.io/projects/spring-security

汇报人：邱依良

2022.8.23

CONTENTS

目录

01/ 问题阐述

02/ 工作原理

03/ 案例实现

04/ 结果展示

PART01
问题阐述

问题阐述

HTTP协议本身是无状态的,为了保存会话信息，浏览器Cookie通过SessionID标识会话请求，服务器以SessionID为key来存储会话信息。 在 Web 项目开发中， Session 会话管理是一个很重要的部分， 用于存储与记录用户的状态或相关的数据。

- 通常情况下 session 交由容器（tomcat） 来负责存储和管理， 但是如果项目部署在多台tomcat 中， 则 session 管理存在很大的问题，多台 tomcat 之间无法共享 session， 比如用户在 tomcat A 服务器上已经登录了， 但当负载均衡跳转到tomcat B 时， 由于 tomcat B 服务器并没有用户的登录信息，session 就失效了， 用户就退出了登录。

- 一旦 tomcat容器关闭或重启也会导致 session 会话失效因此如果项目部署在多台 tomcat 中， 就需要解决 session 共享的问题。

解决方案

- session复制

早期的企业级的使用比较多的一种服务器集群session管理机制。应用服务器开启web容器的session复制功能，在集群中的几台服务器之间同步session对象，使得每台服务器上都保存所有的session信息，这样任何一台宕机都不会导致session的数据丢失，服务器使用session时，直接从本地获取。

这种方式在应用集群达到数千台的时候，就会出现瓶颈，每台都需要备份session，出现内存不够用的情况。

- 基于Nginx的ip_hash 负载均衡
- 利用cookie记录session
- Redis做缓存session的统一缓存

解决方案

- session复制
- 基于Nginx的ip_hash 负载均衡

利用hash算法，比如nginx的ip_hash,使得同一个Ip的请求分发到同一台服务器上。
这种方式不符合对系统的高可用要求，因为一旦某台服务器宕机，那么该机器上的session也就不复存在了，用户请求切换到其他机器后么有session，无法完成业务处理。

- 利用cookie记录session
- Redis做缓存session的统一缓存

解决方案

- session复制
- 基于Nginx的ip_hash 负载均衡
- 利用cookie记录session

session记录在客户端，每次请求服务器的时候，将session放在请求中发送给服务器，服务器处理完请求后再将修改后的session响应给客户端。这里的客户端就是cookie。

利用cookie记录session的也有缺点，比如受cookie大小的限制，能记录的信息有限；每次请求响应都需要传递cookie，影响性能，如果用户关闭cookie，访问就不正常。且一些重要数据容易泄露。

- Redis做缓存session的统一缓存

解决方案

- session复制
- 基于Nginx的ip_hash 负载均衡
- 利用cookie记录session
- Redis做缓存session的统一缓存

把每次用户的请求的时候生成的sessionID给放到Redis的服务器上。然后在基于Redis的特性进行设置一个失效时间的机制，这样就能保证用户在我们设置的Redis中的session失效时间内，都不需要进行再次登录。当业务场景对session管理有比较高的要求，比如利用session服务基层单点登录（sso),用户服务器等功能，就比较适合该方法。本文将使用该方法实现单点登录功能。

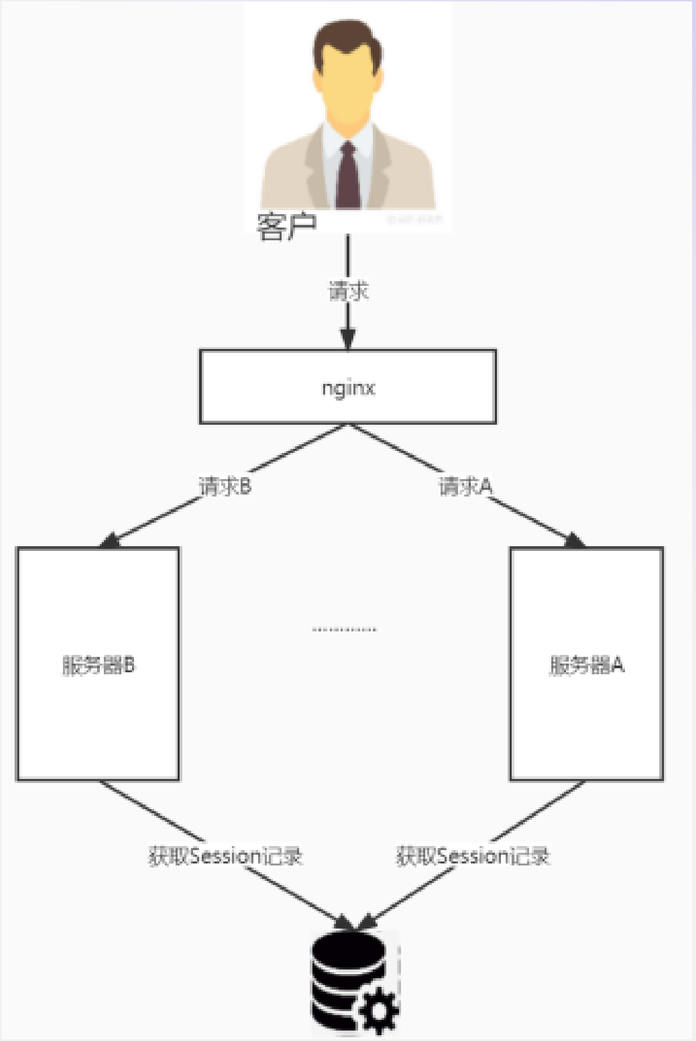
PART02
工作原理

DETAILS

工作流程

客户访问不同服务，会从redis中获取session记录。如果没有，则会被Spring Security拦截，跳转到登录界面；如果有，则会放行，获得请求结果。

登陆后，Spring Session会自动把我们Session记录存到Redis中。此后再次用同一个浏览器访问不同服务时，不再需要登录。



PART03

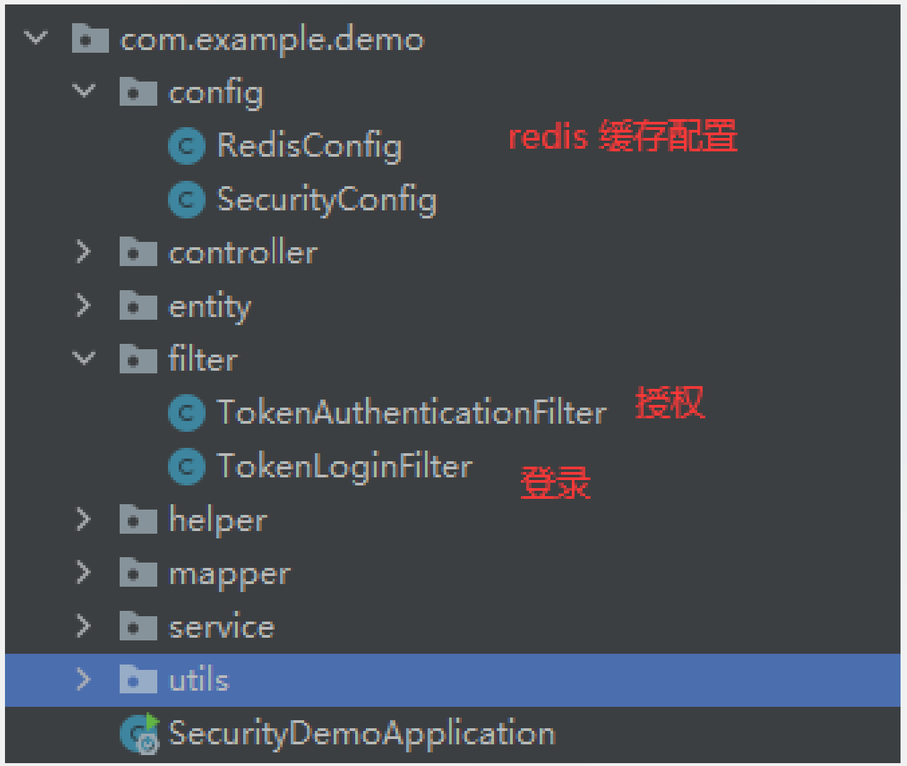
案例实现

DEMO

前期筹备： 修改pom.xml

修改yml

新增filter



项目架构图

代码实现： pom.xml新增依赖

```
<!--        redis -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<!-- spring-session-redis 管理session -->
<dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-data-redis</artifactId>
</dependency>
```

代码实现： application.yml配置redis

```yaml
redis:
  database: 0
  host: 192.168.3.125
  lettuce:
    pool:
      max-active: 20
      max-idle: 5
      max-wait: -1
      min-idle: 0
  port: 6379
  timeout: 1800000
```

代码实现：

SecurityConfig的实现：没有该配置就一切都是Security默认的配置进行拦截；新增上方红框中的两个过滤器，实现单点登陆后的授权功能；新增底部红框实现登录数量限制。

```
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    //自定义登录页面地址
    http.formLogin().loginPage("/login.html") FormLoginConfigurer<HttpSecurity>
            .loginProcessingUrl("/login")                    //登录的请求地址
            .successForwardUrl("/session")                   //成功登录之后跳转的地址
            .and().csrf().disable() HttpSecurity
            .authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .antMatchers( ...antPatterns: "/login", "/register", "/toLogin").permitAll()                // 放行哪些请求
            .anyRequest().authenticated()                    // 除了上述请求都进行拦截校验
            .and().userDetailsService(userDetailsService)// 设置后 从数据库查询数据
            .addFilter(new TokenLoginFilter(authenticationManager(), redisTemplate))        // 自定义在登录之后存权限
            .addFilter(new TokenAuthenticationFilter(authenticationManager(), redisTemplate))// 自定义验证过滤器 session是否存在
            .logout().addLogoutHandler(logoutHandler)// 自定义退出
            .and().httpBasic();
    // 这里限制最多同时在线1个用户，否则就强制下线
    http.sessionManagement().maximumSessions(1)
            .expiredSessionStrategy(sessionStrategy);
    return http.build();
```

代码实现：

TokenLoginFilter的实现：继承UsernamePasswordAuthentication来实现自定 义的登录过滤器，否则就会进入UsernamePasswordAuthentication执行默认登录验证。

```
public TokenLoginFilter(AuthenticationManager authenticationManager, RedisTemplate redisTemplate) {
    this.authenticationManager = authenticationManager;
    this.redisTemplate = redisTemplate;  redis注入
    this.setPostOnly(false);
    this.setRequiresAuthenticationRequestMatcher(new AntPathRequestMatcher( pattern: "/login", httpMethod: "POST"));  //对post形式发送的login进行登录验证
}
```

登录请求的拦截

```
@Override
public Authentication attemptAuthentication(HttpServletRequest req, HttpServletResponse res)
        throws AuthenticationException {
    //获取请求中username、password参数值
    String username = req.getParameter( s: "username");
    String password = req.getParameter( s: "password");
    User user = new User();
    user.setUsername(username);
    user.setPassword(password);
    //把表单信息放入UsernamePasswordAuthenticationToken中
    //进入authenticate方法对信息进行校验
    return authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(user.getUsername(), user.getPassword(), new ArrayList<>()));
}
```

代码实现： TokenLoginFilter的实现：继承UsernamePasswordAuthentication来实现自定 义的登录过滤器，否则就会进入UsernamePasswordAuthentication执行默认登录验证。

```
@Override
protected void successfulAuthentication(HttpServletRequest req, HttpServletResponse res, FilterChain chain,
                                        Authentication auth){
    HttpSession session = req.getSession();
    SecurityUser user = (SecurityUser) auth.getPrincipal();
    RedisBean redisBean = new RedisBean(user.getUsername(),user.getPermissionValueList());
    redisTemplate.opsForValue().set(session.getId(), redisBean); // 以Session : admin 、 权限 存入redis
    ResponseUtil.out(res, R.ok().message("登录成功"));
}
```

将SessionId和角色、权限信息塞入Redis

代码实现：

TokenAuthenticationFilter的实现：实现自定义的授权认证，实现从redis中 获取用户权限；否则将会仅按照BasicAuthenticationFilter中的逻辑进行授权；

```
public TokenAuthenticationFilter(AuthenticationManager authManager,RedisTemplate redisTemplate) {
    super(authManager);
    this.redisTemplate = redisTemplate;  注入redis
}
```

```
@Override
protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res, FilterChain chain)
        throws IOException, ServletException {

    UsernamePasswordAuthenticationToken authentication = null;
    // 获取redis中的用户信息
    authentication = getAuthentication(req);
    // 已经登录就获取信息 放入安全信息上下文
    if (authentication != null) {
        SecurityContextHolder.getContext().setAuthentication(authentication);
    }
    chain.doFilter(req, res);
}
```

通过获取session获取redis中的权限列表

塞入安全信息

代码实现：

TokenAuthenticationFilter的实现：实现自定义的授权认证，实现从redis中 获取用户权限；否则将会仅按照BasicAuthenticationFilter中的逻辑进行授权；

```
private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {
    HttpSession session = request.getSession();
    String sessionId = session.getId();
    // 从redis中获取权限
    Object object = redisTemplate.opsForValue().get(sessionId);
    if (Objects.nonNull(object)) {
        RedisBean redisBean = (RedisBean) object;
        List<String> permissionValueList = redisBean.getPermissionValueList();
        Collection<GrantedAuthority> authorities = new ArrayList<>();
        for (String permissionValue : permissionValueList) {
            if (Strings.isNullOrEmpty(permissionValue)) {
                continue;
            }
            // SimpleGrantedAuthority 权限内容为String 将String类型的权限存入SimpleGrantedAuthority类
            SimpleGrantedAuthority authority = new SimpleGrantedAuthority(permissionValue);
            authorities.add(authority);
        }
        // 把有用信息塞入UsernamePasswordAuthenticationToken后返回
        // UsernamePasswordAuthenticationToken令牌存了sessionid 和 权限
        // 登录之后，已经将信息塞入到SecurityContextHolder 故不再需要redis存信息了
        redisTemplate.delete(sessionId);
        return new UsernamePasswordAuthenticationToken(redisBean.getUsername(), credentials: null, authorities);
    }
    return null;
}
```

获取登录信息

权限的封装

返回指定的令牌，并且删除没用的redis信息

代码实现：

CustomerExpiredSessionStrategy的实现：实现自定义的Session过期策略

```
public class CustomerExpiredSessionStrategy implements SessionInformationExpiredStrategy {

    @Override                                    在被其他用户挤掉之后，返回提示信息
    public void onExpiredSessionDetected(SessionInformationExpiredEvent event) throws IOException, ServletException {
        ResponseUtil.out(event.getResponse(), R.ok().message("您的账号已经在其他地方登录").code(ResultCode.UNAUTHORIZED));
    }
}
```

代码实现：

RedisConfig的实现：新增注解@EnableRedisHttpSession

```
@EnableCaching //开启缓存
@Configuration  //配置类
@EnableRedisHttpSession     把Session交付给Spring Session进行管理
public class RedisConfig extends CachingConfigurerSupport {
```

代码实现：

RedisConfig的实现：新增注解@EnableRedisHttpSession

```
@EnableCaching //开启缓存
@Configuration  //配置类
@EnableRedisHttpSession      把Session交付给Spring Session进行管理
public class RedisConfig extends CachingConfigurerSupport {
```

代码实现：UserController:新增角色校验，这里hasRole默认匹配的会添加前缀，如下图system会变成ROLE_systeml进行校验。

```
@RestController
@RequestMapping("/user")
@PreAuthorize("hasRole('system')")
public class UserController {
```

SHOW

PART04

结果展示

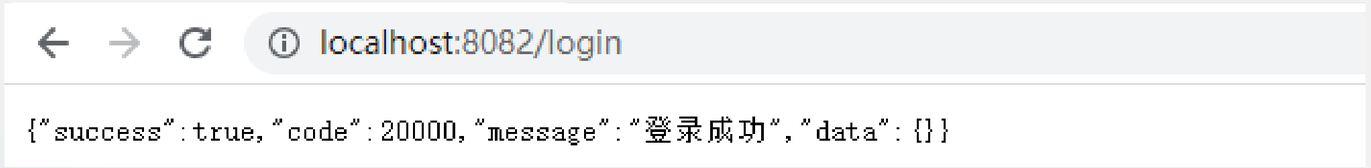
SHOW

结果展示

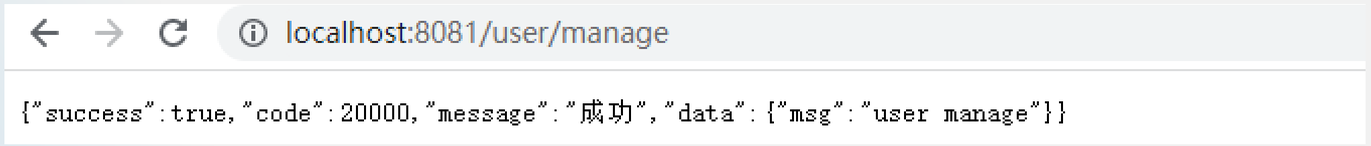
请求goods/manage商品管理

未登录，进行拦截

登陆成功



启动8081端口，访问user/manage，具备权限并放行



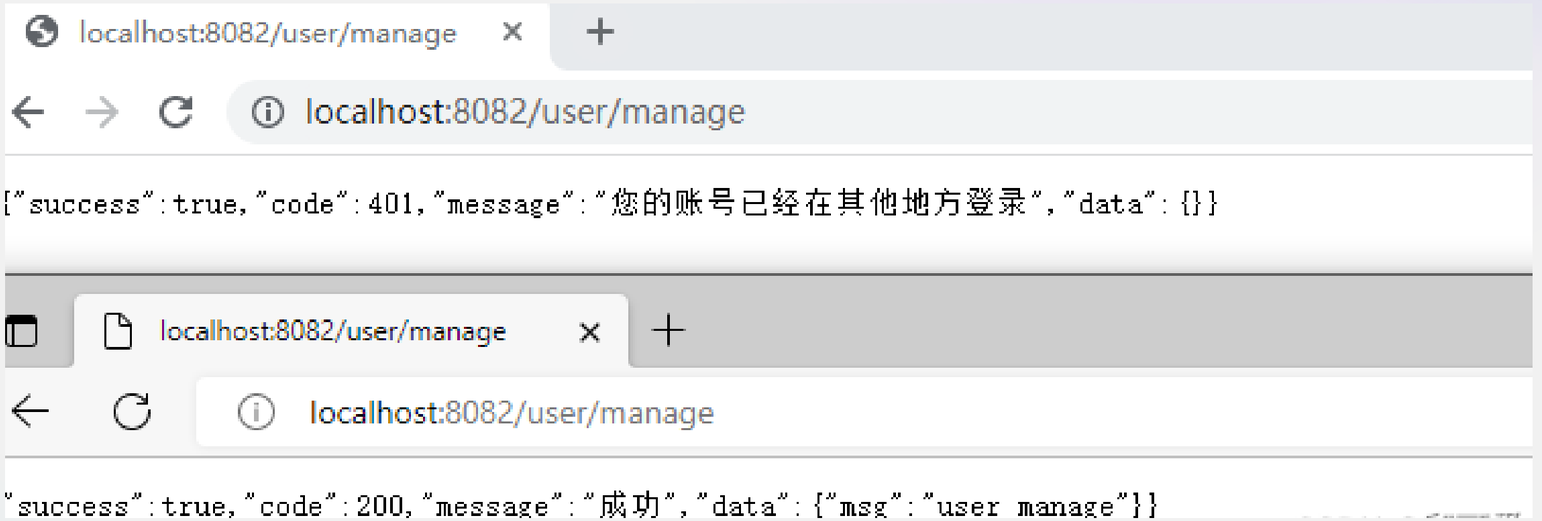
SHOW

结果展示

在另一个浏览器登录相同账号，测试结果如下

原本的账号被迫下线

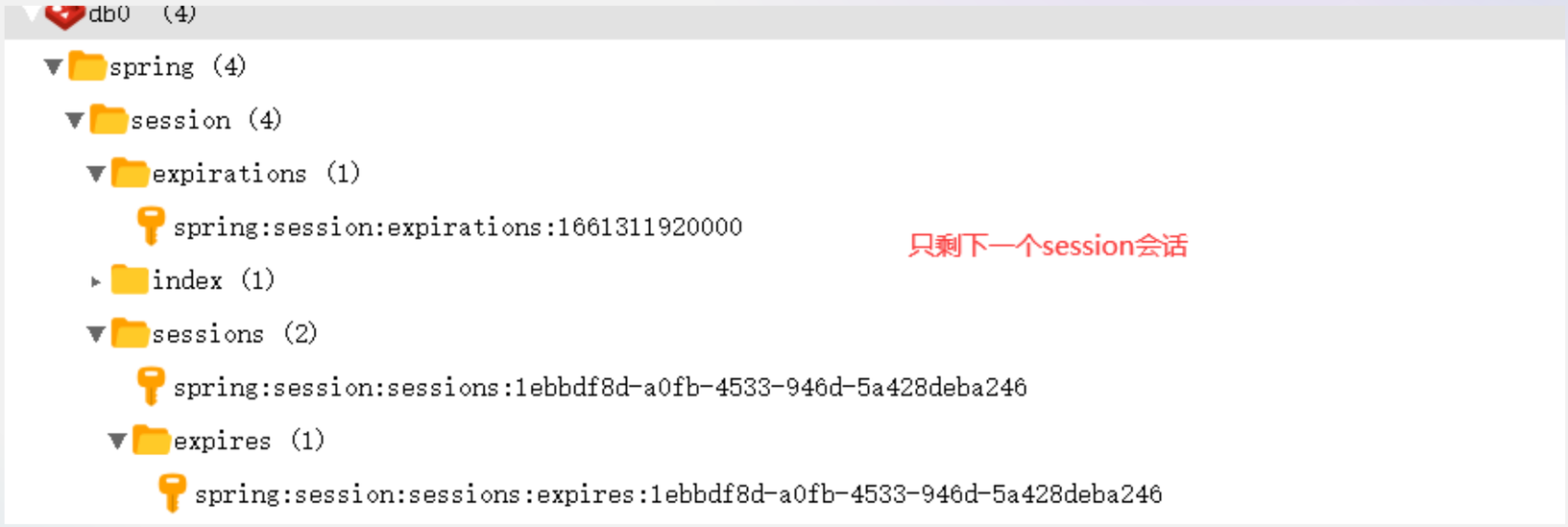
新的账号登录之后成功访问接口



SHOW

结果展示

Redis结果



谢谢观看

THANK YOU

2022.8.23