

Shiro使用redis作为缓存(解决shiro频繁访问Redis)(十一)

原文地址, 转载请注明出处: https://blog.csdn.net/qq_34021712/article/details/80791219 ©王赛超

之前写过一篇博客,使用的一个 [开源项目](#),实现了redis作为缓存 缓存用户的权限 和 session信息,还有两个功能没有修改,一个是用户并发登录限制,一个是用户密码错误次数.本篇中几个类 也是使用的开源项目中的类,只不过是拿出来了,redis单独做的配置,方便进行优化。

整合过程

1.首先是整合Redis

Redis客户端使用的是RedisTemplate,自己写了一个序列化工具继承RedisSerializer

SerializeUtils.java

```
1 package com.springboot.test.shiro.global.utils;
2
3 import org.slf4j.Logger;
4 import org.slf4j.LoggerFactory;
5 import org.springframework.data.redis.serializer.RedisSerializer;
6 import org.springframework.data.redis.serializer.SerializationException;
7
8 import java.io.*;
9
10 /**
11  * @author: wangsaichao
12  * @date: 2018/6/20
13  * @description: redis的value序列化工具
14  */
15 public class SerializeUtils implements RedisSerializer {
16
17     private static Logger logger = LoggerFactory.getLogger(SerializeUtils.class);
18
19     public static boolean isEmpty(byte[] data) {
20         return (data == null || data.length == 0);
21     }
22
23     /**
24      * 序列化
25      * @param object
26      * @return
27      * @throws SerializationException
28      */
29     @Override
30     public byte[] serialize(Object object) throws SerializationException {
31         byte[] result = null;
32
33         if (object == null) {
34             return new byte[0];
35         }
36         try {
37             ByteArrayOutputStream byteStream = new ByteArrayOutputStream(128);
38             ObjectOutputStream objectOutputStream = new ObjectOutputStream(byteStream)
39         }{
40
41             if (!(object instanceof Serializable)) {
42                 throw new IllegalArgumentException(SerializeUtils.class.getSimpleName() + " requires a Serializable payload " +
43                     "but received an object of type [" + object.getClass().getName() + "]");
44             }
45
46             objectOutputStream.writeObject(object);
47             objectOutputStream.flush();
48             result = byteStream.toByteArray();
49         } catch (Exception ex) {
50             logger.error("Failed to serialize", ex);
51         }
52         return result;
53     }
54
55     /**
56      * 反序列化
57      * @param bytes
58      * @return
59      * @throws SerializationException
60      */
61     @Override
62     public Object deserialize(byte[] bytes) throws SerializationException {
```

```
63
64     Object result = null;
65
66     if (isEmpty(bytes)) {
67         return null;
68     }
69
70     try (
71         ByteArrayInputStream byteStream = new ByteArrayInputStream(bytes);
72         ObjectInputStream objectInputStream = new ObjectInputStream(byteStream)
73     ){
74         result = objectInputStream.readObject();
75     } catch (Exception e) {
76         logger.error("Failed to deserialize",e);
77     }
78     return result;
79 }
80
81 }
```

RedisConfig.java

```
1  package com.springboot.test.shiro.config;
2
3  import com.springboot.test.shiro.global.utils.SerializeUtils;
4  import org.springframework.beans.factory.annotation.Value;
5  import org.springframework.context.annotation.Bean;
6  import org.springframework.context.annotation.Configuration;
7  import org.springframework.data.redis.connection.RedisConnectionFactory;
8  import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
9  import org.springframework.data.redis.core.RedisTemplate;
10 import org.springframework.data.redis.serializer.StringRedisSerializer;
11 import redis.clients.jedis.JedisPoolConfig;
12
13 /**
14  * @author: wangsai chao
15  * @date: 2017/11/23
16  * @description: redis配置
17  */
18 @Configuration
19 public class RedisConfig {
20
21     /**
22      * redis地址
23      */
24     @Value("${spring.redis.host}")
25     private String host;
26
27     /**
28      * redis端口号
29      */
30     @Value("${spring.redis.port}")
31     private Integer port;
32
33     /**
34      * redis密码
35      */
36     @Value("${spring.redis.password}")
37     private String password;
38
39     /**
40      * JedisPoolConfig 连接池
41      * @return
42      */
43     @Bean
44     public JedisPoolConfig jedisPoolConfig(){
45         JedisPoolConfig jedisPoolConfig=new JedisPoolConfig();
46         //最大空闲数
47         jedisPoolConfig.setMaxIdle(300);
48         //连接池的最大数据库连接数
49         jedisPoolConfig.setMaxTotal(1000);
50         //最大建立连接等待时间
51         jedisPoolConfig.setMaxWaitMillis(1000);
52         //逐出连接的最小空闲时间 默认1800000毫秒(30分钟)
53         jedisPoolConfig.setMinEvictableIdleTimeMillis(300000);
54         //每次逐出检查时 逐出的最大数目 如果为负数就是 : 1/abs(n), 默认3
55         jedisPoolConfig.setNumTestsPerEvictionRun(10);
56         //逐出扫描的时间间隔(毫秒) 如果为负数,则不运行逐出线程, 默认-1
```

```

57     jedisPoolConfig.setTimeBetweenEvictionRunsMillis(30000);
58     //是否在从池中取出连接前进行检验,如果检验失败,则从池中去除连接并尝试取出另一个
59     jedisPoolConfig.setTestOnBorrow(true);
60     //在空闲时检查有效性, 默认false
61     jedisPoolConfig.setTestWhileIdle(true);
62     return jedisPoolConfig;
63 }
64
65 /**
66  * 配置工厂
67  * @param jedisPoolConfig
68  * @return
69  */
70 @Bean
71 public JedisConnectionFactory jedisConnectionFactory(JedisPoolConfig jedisPoolConfig){
72     JedisConnectionFactory jedisConnectionFactory=new JedisConnectionFactory();
73     //连接池
74     jedisConnectionFactory.setPoolConfig(jedisPoolConfig);
75     //IP地址
76     jedisConnectionFactory.setHostName(host);
77     //端口号
78     jedisConnectionFactory.setPort(port);
79     //如果Redis设置有密码
80     jedisConnectionFactory.setPassword(password);
81     //客户端超时时间单位是毫秒
82     jedisConnectionFactory.setTimeout(5000);
83     return jedisConnectionFactory;
84 }
85
86 /**
87  * shiro redis缓存使用的模板
88  * 实例化 RedisTemplate 对象
89  * @return
90  */
91 @Bean("shiroRedisTemplate")
92 public RedisTemplate shiroRedisTemplate(RedisConnectionFactory redisConnectionFactory) {
93
94     RedisTemplate redisTemplate = new RedisTemplate();
95     redisTemplate.setKeySerializer(new StringRedisSerializer());
96     redisTemplate.setHashKeySerializer(new StringRedisSerializer());
97     redisTemplate.setHashValueSerializer(new SerializeUtils());
98     redisTemplate.setValueSerializer(new SerializeUtils());
99     //开启事务
100    //stringRedisTemplate.setEnableTransactionSupport(true);
101    redisTemplate.setConnectionFactory(redisConnectionFactory);
102    return redisTemplate;
103 }
104
105 }

```

RedisManager.java

```

1  package com.springboot.test.shiro.config.shiro;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.dao.DataAccessException;
5  import org.springframework.data.redis.connection.RedisConnection;
6  import org.springframework.data.redis.core.*;
7  import org.springframework.util.CollectionUtils;
8
9  import java.util.*;
10 import java.util.concurrent.TimeUnit;
11
12 /**
13  *
14  * @author wangsaichao
15  * 基于spring和redis的redisTemplate工具类
16  */
17 public class RedisManager {
18
19     @Autowired
20     private RedisTemplate<String, Object> redisTemplate;
21
22     //=====common=====
23     /**
24      * 指定缓存失效时间
25      * @param key 键
26      * @param time 时间(秒)

```

```

27     */
28     public void expire(String key,long time){
29         redisTemplate.expire(key, time, TimeUnit.SECONDS);
30     }
31
32     /**
33      * 判断key是否存在
34      * @param key 键
35      * @return true 存在 false不存在
36      */
37     public Boolean hasKey(String key){
38         return redisTemplate.hasKey(key);
39     }
40
41     /**
42      * 删除缓存
43      * @param key 可以传一个值 或多个
44      */
45     @SuppressWarnings("unchecked")
46     public void del(String ... key){
47         if(key!=null&&key.length>0){
48             if(key.length==1){
49                 redisTemplate.delete(key[0]);
50             }else{
51                 redisTemplate.delete(CollectionUtils.arrayToList(key));
52             }
53         }
54     }
55
56     /**
57      * 批量删除key
58      * @param keys
59      */
60     public void del(Collection keys){
61         redisTemplate.delete(keys);
62     }
63
64     //=====String=====
65     /**
66      * 普通缓存获取
67      * @param key 键
68      * @return 值
69      */
70     public Object get(String key){
71         return redisTemplate.opsForValue().get(key);
72     }
73
74     /**
75      * 普通缓存放入
76      * @param key 键
77      * @param value 值
78      */
79     public void set(String key,Object value) {
80         redisTemplate.opsForValue().set(key, value);
81     }
82
83     /**
84      * 普通缓存放入并设置时间
85      * @param key 键
86      * @param value 值
87      * @param time 时间(秒) time要大于0 如果time小于等于0 将设置无限期
88      */
89     public void set(String key,Object value,long time){
90         if(time>0){
91             redisTemplate.opsForValue().set(key, value, time, TimeUnit.SECONDS);
92         }else{
93             set(key, value);
94         }
95     }
96
97     /**
98      * 使用scan命令 查询某些前缀的key
99      * @param key
100     * @return
101     */
102     public Set<String> scan(String key){
103         Set<String> execute = this.redisTemplate.execute(new RedisCallback<Set<String>>() {
104
105             @Override

```

```

106         public Set<String> doInRedis(RedisConnection connection) throws DataAccessException {
107
108             Set<String> binaryKeys = new HashSet<>();
109
110             Cursor<byte[]> cursor = connection.scan(new ScanOptions.ScanOptionsBuilder().match(key).count(1000).build());
111             while (cursor.hasNext()) {
112                 binaryKeys.add(new String(cursor.next()));
113             }
114             return binaryKeys;
115         }
116     };
117     return execute;
118 }
119
120 /**
121  * 使用scan命令 查询某些前缀的key 有多少个
122  * 用来获取当前session数量,也就是在线用户
123  * @param key
124  * @return
125  */
126 public Long scanSize(String key){
127     long dbSize = this.redisTemplate.execute(new RedisCallback<Long>() {
128
129         @Override
130         public Long doInRedis(RedisConnection connection) throws DataAccessException {
131             long count = 0L;
132             Cursor<byte[]> cursor = connection.scan(ScanOptions.scanOptions().match(key).count(1000).build());
133             while (cursor.hasNext()) {
134                 cursor.next();
135                 count++;
136             }
137             return count;
138         }
139     });
140     return dbSize;
141 }
142 }

```

2.使用Redis作为缓存需要shiro 重写cache、cacheManager、SessionDAO

RedisCache.java

```

1 package com.springboot.test.shiro.config.shiro;
2
3 import com.springboot.test.shiro.global.exceptions.PrincipalIdNullException;
4 import com.springboot.test.shiro.global.exceptions.PrincipalInstanceException;
5 import org.apache.shiro.cache.Cache;
6 import org.apache.shiro.cache.CacheException;
7 import org.apache.shiro.subject.PrincipalCollection;
8 import org.apache.shiro.util.CollectionUtils;
9 import org.slf4j.Logger;
10 import org.slf4j.LoggerFactory;
11 import java.lang.reflect.InvocationTargetException;
12 import java.lang.reflect.Method;
13 import java.util.*;
14
15 /**
16  * @author: wangsai chao
17  * @date: 2018/6/22
18  * @description: 参考 shiro-redis 开源项目 Git地址 https://github.com/alexxyyang/shiro-redis
19  */
20 public class RedisCache<K, V> implements Cache<K, V> {
21
22     private static Logger logger = LoggerFactory.getLogger(RedisCache.class);
23
24     private RedisManager redisManager;
25     private String keyPrefix = "";
26     private int expire = 0;
27     private String principalIdFieldName = RedisCacheManager.DEFAULT_PRINCIPAL_ID_FIELD_NAME;
28
29     /**
30      * Construction
31      * @param redisManager
32      */
33     public RedisCache(RedisManager redisManager, String prefix, int expire, String principalIdFieldName) {
34         if (redisManager == null) {
35             throw new IllegalArgumentException("redisManager cannot be null.");
36         }
37     }

```

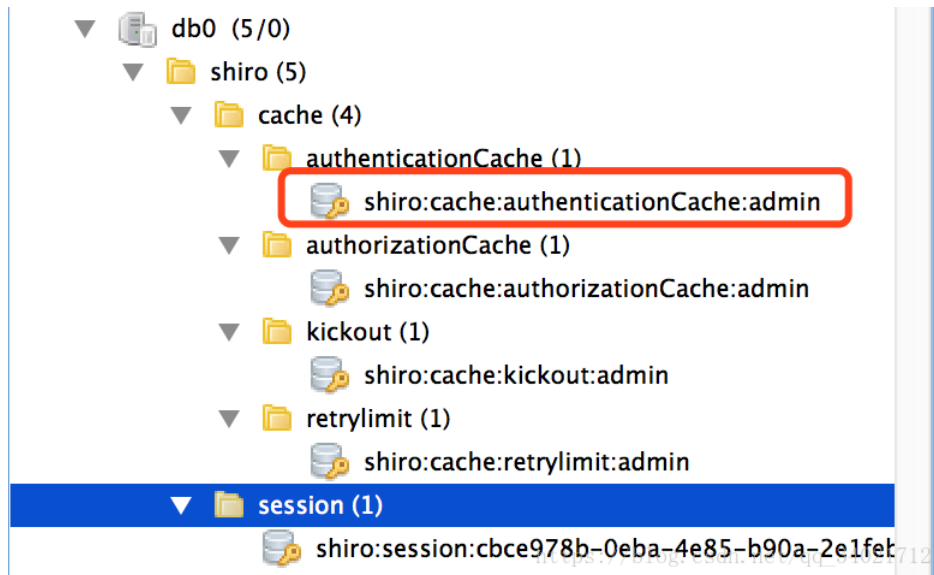
```
37     this.redisManager = redisManager;
38     if (prefix != null && !"".equals(prefix)) {
39         this.keyPrefix = prefix;
40     }
41     if (expire != -1) {
42         this.expire = expire;
43     }
44     if (principalIdFieldName != null && !"".equals(principalIdFieldName)) {
45         this.principalIdFieldName = principalIdFieldName;
46     }
47 }
48
49 @Override
50 public V get(K key) throws CacheException {
51     logger.debug("get key [{}]",key);
52
53     if (key == null) {
54         return null;
55     }
56
57     try {
58         String redisCacheKey = getRedisCacheKey(key);
59         Object rawValue = redisManager.get(redisCacheKey);
60         if (rawValue == null) {
61             return null;
62         }
63         V value = (V) rawValue;
64         return value;
65     } catch (Exception e) {
66         throw new CacheException(e);
67     }
68 }
69
70 @Override
71 public V put(K key, V value) throws CacheException {
72     logger.debug("put key [{}]",key);
73     if (key == null) {
74         logger.warn("Saving a null key is meaningless, return value directly without call Redis.");
75         return value;
76     }
77     try {
78         String redisCacheKey = getRedisCacheKey(key);
79         redisManager.set(redisCacheKey, value != null ? value : null, expire);
80         return value;
81     } catch (Exception e) {
82         throw new CacheException(e);
83     }
84 }
85
86 @Override
87 public V remove(K key) throws CacheException {
88     logger.debug("remove key [{}]",key);
89     if (key == null) {
90         return null;
91     }
92     try {
93         String redisCacheKey = getRedisCacheKey(key);
94         Object rawValue = redisManager.get(redisCacheKey);
95         V previous = (V) rawValue;
96         redisManager.del(redisCacheKey);
97         return previous;
98     } catch (Exception e) {
99         throw new CacheException(e);
100    }
101 }
102
103 private String getRedisCacheKey(K key) {
104     if (key == null) {
105         return null;
106     }
107     return this.keyPrefix + getStringRedisKey(key);
108 }
109
110 private String getStringRedisKey(K key) {
111     String redisKey;
112     if (key instanceof PrincipalCollection) {
113         redisKey = getRedisKeyFromPrincipalIdField((PrincipalCollection) key);
114     } else {
115         redisKey = key.toString();
116     }
117 }
```

```
116     }
117     return redisKey;
118 }
119
120 private String getRedisKeyFromPrincipalIdField(PrincipalCollection key) {
121     String redisKey;
122     Object principalObject = key.getPrimaryPrincipal();
123     Method principalIdGetter = null;
124     Method[] methods = principalObject.getClass().getDeclaredMethods();
125     for (Method m:methods) {
126         if (RedisCacheManager.DEFAULT_PRINCIPAL_ID_FIELD_NAME.equals(this.principalIdFieldName)
127             && ("getAuthCacheKey".equals(m.getName()) || "getId".equals(m.getName()))) {
128             principalIdGetter = m;
129             break;
130         }
131         if (m.getName().equals("get" + this.principalIdFieldName.substring(0, 1).toUpperCase() + this.principalIdFieldName.substri
132             principalIdGetter = m;
133             break;
134         }
135     }
136     if (principalIdGetter == null) {
137         throw new PrincipalInstanceException(principalObject.getClass(), this.principalIdFieldName);
138     }
139
140     try {
141         Object idObj = principalIdGetter.invoke(principalObject);
142         if (idObj == null) {
143             throw new PrincipalIdNullException(principalObject.getClass(), this.principalIdFieldName);
144         }
145         redisKey = idObj.toString();
146     } catch (IllegalAccessException e) {
147         throw new PrincipalInstanceException(principalObject.getClass(), this.principalIdFieldName, e);
148     } catch (InvocationTargetException e) {
149         throw new PrincipalInstanceException(principalObject.getClass(), this.principalIdFieldName, e);
150     }
151
152     return redisKey;
153 }
154
155
156 @Override
157 public void clear() throws CacheException {
158     logger.debug("clear cache");
159     Set<String> keys = null;
160     try {
161         keys = redisManager.scan(this.keyPrefix + "*");
162     } catch (Exception e) {
163         logger.error("get keys error", e);
164     }
165     if (keys == null || keys.size() == 0) {
166         return;
167     }
168     for (String key: keys) {
169         redisManager.del(key);
170     }
171 }
172
173 @Override
174 public int size() {
175     Long longSize = 0L;
176     try {
177         longSize = new Long(redisManager.scanSize(this.keyPrefix + "*"));
178     } catch (Exception e) {
179         logger.error("get keys error", e);
180     }
181     return longSize.intValue();
182 }
183
184 @SuppressWarnings("unchecked")
185 @Override
186 public Set<K> keys() {
187     Set<String> keys = null;
188     try {
189         keys = redisManager.scan(this.keyPrefix + "*");
190     } catch (Exception e) {
191         logger.error("get keys error", e);
192     }
193     return Collections.emptySet();
194 }
```

```
195     if (CollectionUtils.isEmpty(keys)) {
196         return Collections.emptySet();
197     }
198
199     Set<K> convertedKeys = new HashSet<K>();
200     for (String key:keys) {
201         try {
202             convertedKeys.add((K) key);
203         } catch (Exception e) {
204             logger.error("deserialize keys error", e);
205         }
206     }
207     return convertedKeys;
208 }
209
210 @Override
211 public Collection<V> values() {
212     Set<String> keys = null;
213     try {
214         keys = redisManager.scan(this.keyPrefix + "*");
215     } catch (Exception e) {
216         logger.error("get values error", e);
217         return Collections.emptySet();
218     }
219
220     if (CollectionUtils.isEmpty(keys)) {
221         return Collections.emptySet();
222     }
223
224     List<V> values = new ArrayList<V>(keys.size());
225     for (String key : keys) {
226         V value = null;
227         try {
228             value = (V) redisManager.get(key);
229         } catch (Exception e) {
230             logger.error("deserialize values= error", e);
231         }
232         if (value != null) {
233             values.add(value);
234         }
235     }
236     return Collections.unmodifiableList(values);
237 }
238
239 public String getKeyPrefix() {
240     return keyPrefix;
241 }
242
243 public void setKeyPrefix(String keyPrefix) {
244     this.keyPrefix = keyPrefix;
245 }
246
247 public String getPrincipalIdFieldName() {
248     return principalIdFieldName;
249 }
250
251 public void setPrincipalIdFieldName(String principalIdFieldName) {
252     this.principalIdFieldName = principalIdFieldName;
253 }
254 }
```

`getRedisKeyFromPrincipalIdField()` 是获取缓存的用户身份信息 和用户权限信息。 里面有一个属性`principalIdFieldName` 在`RedisCacheManager`也有这个属性,设置其中一个就可以.是为了给缓存用户身份和权限信息在Redis中的key唯一,登录用户名可能是`username` 或者 `phoneNum` 或者是Email中的一个,如 我的User实体类中 有一个

username字段,也是登录时候使用的用户名,在redis中缓存的权限信息key 如下, 这个admin 就是 通过getUsername获得的。



读取用户权限信息时,还用到两个异常类, 如下:

PrincipalInstanceException.java

```
1 package com.springboot.test.shiro.global.exceptions;
2
3 /**
4  * @author: wangsaichao
5  * @date: 2018/6/21
6  * @description:
7  */
8 public class PrincipalInstanceException extends RuntimeException {
9
10     private static final String MESSAGE = "We need a field to identify this Cache Object in Redis. "
11         + "So you need to defined an id field which you can get unique id to identify this principal. "
12         + "For example, if you use UserInfo as Principal class, the id field maybe userId, userName, email, etc. "
13         + "For example, getUserId(), getUserName(), getEmail(), etc.\n"
14         + "Default value is authCacheKey or id, that means your principal object has a method called \"getAuthCacheKey()\" or \"ge
15
16     public PrincipalInstanceException(Class clazz, String idMethodName) {
17         super(clazz + " must has getter for field: " + idMethodName + "\n" + MESSAGE);
18     }
19
20     public PrincipalInstanceException(Class clazz, String idMethodName, Exception e) {
21         super(clazz + " must has getter for field: " + idMethodName + "\n" + MESSAGE, e);
22     }
23 }
```

PrincipalIdNullException.java

```
1 package com.springboot.test.shiro.global.exceptions;
2
3 /**
4  * @author: wangsaichao
5  * @date: 2018/6/21
6  * @description:
7  */
8 public class PrincipalIdNullException extends RuntimeException {
9
10     private static final String MESSAGE = "Principal Id shouldn't be null!";
11
12     public PrincipalIdNullException(Class clazz, String idMethodName) {
13         super(clazz + " id field: " + idMethodName + ", value is null\n" + MESSAGE);
14     }
15 }
```

RedisCacheManager.java

```
1 package com.springboot.test.shiro.config.shiro;
2
3 import org.apache.shiro.cache.Cache;
```

```
4 import org.apache.shiro.cache.CacheException;
5 import org.apache.shiro.cache.CacheManager;
6 import org.slf4j.Logger;
7 import org.slf4j.LoggerFactory;
8
9 import java.util.concurrent.ConcurrentHashMap;
10 import java.util.concurrent.ConcurrentMap;
11
12 /**
13  * @author: wangsai chao
14  * @date: 2018/6/22
15  * @description: 参考 shiro-redis 开源项目 Git地址 https://github.com/alexxyyang/shiro-redis
16  */
17 public class RedisCacheManager implements CacheManager {
18
19     private final Logger logger = LoggerFactory.getLogger(RedisCacheManager.class);
20
21     /**
22      * fast lookup by name map
23      */
24     private final ConcurrentMap<String, Cache> caches = new ConcurrentHashMap<String, Cache>();
25
26     private RedisManager redisManager;
27
28     /**
29      * expire time in seconds
30      */
31     private static final int DEFAULT_EXPIRE = 1800;
32     private int expire = DEFAULT_EXPIRE;
33
34     /**
35      * The Redis key prefix for caches
36      */
37     public static final String DEFAULT_CACHE_KEY_PREFIX = "shiro:cache:";
38     private String keyPrefix = DEFAULT_CACHE_KEY_PREFIX;
39
40     public static final String DEFAULT_PRINCIPAL_ID_FIELD_NAME = "authCacheKey or id";
41     private String principalIdFieldName = DEFAULT_PRINCIPAL_ID_FIELD_NAME;
42
43     @Override
44     public <K, V> Cache<K, V> getCache(String name) throws CacheException {
45         logger.debug("get cache, name={}", name);
46
47         Cache cache = caches.get(name);
48
49         if (cache == null) {
50             cache = new RedisCache<K, V>(redisManager, keyPrefix + name + ":", expire, principalIdFieldName);
51             caches.put(name, cache);
52         }
53         return cache;
54     }
55
56     public RedisManager getRedisManager() {
57         return redisManager;
58     }
59
60     public void setRedisManager(RedisManager redisManager) {
61         this.redisManager = redisManager;
62     }
63
64     public String getKeyPrefix() {
65         return keyPrefix;
66     }
67
68     public void setKeyPrefix(String keyPrefix) {
69         this.keyPrefix = keyPrefix;
70     }
71
72     public int getExpire() {
73         return expire;
74     }
75
76     public void setExpire(int expire) {
77         this.expire = expire;
78     }
79
80     public String getPrincipalIdFieldName() {
81         return principalIdFieldName;
82     }
```

```

83
84     public void setPrincipalIdFieldName(String principalIdFieldName) {
85         this.principalIdFieldName = principalIdFieldName;
86     }
87 }

```

RedisSessionDAO.java

```

1  package com.springboot.test.shiro.config.shiro;
2
3  import org.apache.shiro.session.Session;
4  import org.apache.shiro.session.UnknownSessionException;
5  import org.apache.shiro.session.mgt.ValidatingSession;
6  import org.apache.shiro.session.mgt.eis.AbstractSessionDAO;
7  import org.slf4j.Logger;
8  import org.slf4j.LoggerFactory;
9
10 import java.io.Serializable;
11 import java.util.*;
12
13 /**
14  * @author: wangsaichao
15  * @date: 2018/6/22
16  * @description: 参考 shiro-redis 开源项目 Git地址 https://github.com/alexxyyang/shiro-redis
17  */
18 public class RedisSessionDAO extends AbstractSessionDAO {
19
20     private static Logger logger = LoggerFactory.getLogger(RedisSessionDAO.class);
21
22     private static final String DEFAULT_SESSION_KEY_PREFIX = "shiro:session:";
23     private String keyPrefix = DEFAULT_SESSION_KEY_PREFIX;
24
25     private static final long DEFAULT_SESSION_IN_MEMORY_TIMEOUT = 1000L;
26     /**
27      * doReadSession be called about 10 times when login.
28      * Save Session in ThreadLocal to resolve this problem. sessionInMemoryTimeout is expiration of Session in ThreadLocal.
29      * The default value is 1000 milliseconds (1s).
30      * Most of time, you don't need to change it.
31      */
32     private long sessionInMemoryTimeout = DEFAULT_SESSION_IN_MEMORY_TIMEOUT;
33
34     /**
35      * expire time in seconds
36      */
37     private static final int DEFAULT_EXPIRE = -2;
38     private static final int NO_EXPIRE = -1;
39
40     /**
41      * Please make sure expire is longer than session.setTimeout()
42      */
43     private int expire = DEFAULT_EXPIRE;
44
45     private static final int MILLISECONDS_IN_A_SECOND = 1000;
46
47     private RedisManager redisManager;
48     private static ThreadLocal sessionsInThread = new ThreadLocal();
49
50     @Override
51     public void update(Session session) throws UnknownSessionException {
52         //如果会话过期/停止 没必要再更新了
53         try {
54             if (session instanceof ValidatingSession && !((ValidatingSession) session).isValid()) {
55                 return;
56             }
57
58             if (session instanceof ShiroSession) {
59                 // 如果没有主要字段(除lastAccessTime以外其他字段)发生改变
60                 ShiroSession ss = (ShiroSession) session;
61                 if (!ss.isChanged()) {
62                     return;
63                 }
64                 //如果没有返回 证明有调用 setAttribute往redis 放的时候永远设置为false
65                 ss.setChanged(false);
66             }
67
68             this.saveSession(session);
69         } catch (Exception e) {
70             logger.warn("update Session is failed", e);
71         }
72     }
73
74     private void saveSession(Session session) {
75         // TODO: Implement saveSession method
76     }
77
78     private Session readSession(long sessionId) {
79         // TODO: Implement readSession method
80     }
81
82     private void deleteSession(long sessionId) {
83         // TODO: Implement deleteSession method
84     }
85
86     private void clear() {
87         // TODO: Implement clear method
88     }
89
90     private void setKeyPrefix(String keyPrefix) {
91         this.keyPrefix = keyPrefix;
92     }
93
94     private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
95         this.sessionInMemoryTimeout = sessionInMemoryTimeout;
96     }
97
98     private void setExpire(int expire) {
99         this.expire = expire;
100    }
101
102    private void setRedisManager(RedisManager redisManager) {
103        this.redisManager = redisManager;
104    }
105
106    private void setSessionsInThread(ThreadLocal sessionsInThread) {
107        this.sessionsInThread = sessionsInThread;
108    }
109
110    private void setLogger(Logger logger) {
111        this.logger = logger;
112    }
113
114    private void setKeyPrefix(String keyPrefix) {
115        this.keyPrefix = keyPrefix;
116    }
117
118    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
119        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
120    }
121
122    private void setExpire(int expire) {
123        this.expire = expire;
124    }
125
126    private void setRedisManager(RedisManager redisManager) {
127        this.redisManager = redisManager;
128    }
129
130    private void setSessionsInThread(ThreadLocal sessionsInThread) {
131        this.sessionsInThread = sessionsInThread;
132    }
133
134    private void setLogger(Logger logger) {
135        this.logger = logger;
136    }
137
138    private void setKeyPrefix(String keyPrefix) {
139        this.keyPrefix = keyPrefix;
140    }
141
142    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
143        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
144    }
145
146    private void setExpire(int expire) {
147        this.expire = expire;
148    }
149
150    private void setRedisManager(RedisManager redisManager) {
151        this.redisManager = redisManager;
152    }
153
154    private void setSessionsInThread(ThreadLocal sessionsInThread) {
155        this.sessionsInThread = sessionsInThread;
156    }
157
158    private void setLogger(Logger logger) {
159        this.logger = logger;
160    }
161
162    private void setKeyPrefix(String keyPrefix) {
163        this.keyPrefix = keyPrefix;
164    }
165
166    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
167        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
168    }
169
170    private void setExpire(int expire) {
171        this.expire = expire;
172    }
173
174    private void setRedisManager(RedisManager redisManager) {
175        this.redisManager = redisManager;
176    }
177
178    private void setSessionsInThread(ThreadLocal sessionsInThread) {
179        this.sessionsInThread = sessionsInThread;
180    }
181
182    private void setLogger(Logger logger) {
183        this.logger = logger;
184    }
185
186    private void setKeyPrefix(String keyPrefix) {
187        this.keyPrefix = keyPrefix;
188    }
189
190    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
191        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
192    }
193
194    private void setExpire(int expire) {
195        this.expire = expire;
196    }
197
198    private void setRedisManager(RedisManager redisManager) {
199        this.redisManager = redisManager;
200    }
201
202    private void setSessionsInThread(ThreadLocal sessionsInThread) {
203        this.sessionsInThread = sessionsInThread;
204    }
205
206    private void setLogger(Logger logger) {
207        this.logger = logger;
208    }
209
210    private void setKeyPrefix(String keyPrefix) {
211        this.keyPrefix = keyPrefix;
212    }
213
214    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
215        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
216    }
217
218    private void setExpire(int expire) {
219        this.expire = expire;
220    }
221
222    private void setRedisManager(RedisManager redisManager) {
223        this.redisManager = redisManager;
224    }
225
226    private void setSessionsInThread(ThreadLocal sessionsInThread) {
227        this.sessionsInThread = sessionsInThread;
228    }
229
230    private void setLogger(Logger logger) {
231        this.logger = logger;
232    }
233
234    private void setKeyPrefix(String keyPrefix) {
235        this.keyPrefix = keyPrefix;
236    }
237
238    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
239        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
240    }
241
242    private void setExpire(int expire) {
243        this.expire = expire;
244    }
245
246    private void setRedisManager(RedisManager redisManager) {
247        this.redisManager = redisManager;
248    }
249
250    private void setSessionsInThread(ThreadLocal sessionsInThread) {
251        this.sessionsInThread = sessionsInThread;
252    }
253
254    private void setLogger(Logger logger) {
255        this.logger = logger;
256    }
257
258    private void setKeyPrefix(String keyPrefix) {
259        this.keyPrefix = keyPrefix;
260    }
261
262    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
263        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
264    }
265
266    private void setExpire(int expire) {
267        this.expire = expire;
268    }
269
270    private void setRedisManager(RedisManager redisManager) {
271        this.redisManager = redisManager;
272    }
273
274    private void setSessionsInThread(ThreadLocal sessionsInThread) {
275        this.sessionsInThread = sessionsInThread;
276    }
277
278    private void setLogger(Logger logger) {
279        this.logger = logger;
280    }
281
282    private void setKeyPrefix(String keyPrefix) {
283        this.keyPrefix = keyPrefix;
284    }
285
286    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
287        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
288    }
289
290    private void setExpire(int expire) {
291        this.expire = expire;
292    }
293
294    private void setRedisManager(RedisManager redisManager) {
295        this.redisManager = redisManager;
296    }
297
298    private void setSessionsInThread(ThreadLocal sessionsInThread) {
299        this.sessionsInThread = sessionsInThread;
300    }
301
302    private void setLogger(Logger logger) {
303        this.logger = logger;
304    }
305
306    private void setKeyPrefix(String keyPrefix) {
307        this.keyPrefix = keyPrefix;
308    }
309
310    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
311        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
312    }
313
314    private void setExpire(int expire) {
315        this.expire = expire;
316    }
317
318    private void setRedisManager(RedisManager redisManager) {
319        this.redisManager = redisManager;
320    }
321
322    private void setSessionsInThread(ThreadLocal sessionsInThread) {
323        this.sessionsInThread = sessionsInThread;
324    }
325
326    private void setLogger(Logger logger) {
327        this.logger = logger;
328    }
329
330    private void setKeyPrefix(String keyPrefix) {
331        this.keyPrefix = keyPrefix;
332    }
333
334    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
335        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
336    }
337
338    private void setExpire(int expire) {
339        this.expire = expire;
340    }
341
342    private void setRedisManager(RedisManager redisManager) {
343        this.redisManager = redisManager;
344    }
345
346    private void setSessionsInThread(ThreadLocal sessionsInThread) {
347        this.sessionsInThread = sessionsInThread;
348    }
349
350    private void setLogger(Logger logger) {
351        this.logger = logger;
352    }
353
354    private void setKeyPrefix(String keyPrefix) {
355        this.keyPrefix = keyPrefix;
356    }
357
358    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
359        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
360    }
361
362    private void setExpire(int expire) {
363        this.expire = expire;
364    }
365
366    private void setRedisManager(RedisManager redisManager) {
367        this.redisManager = redisManager;
368    }
369
370    private void setSessionsInThread(ThreadLocal sessionsInThread) {
371        this.sessionsInThread = sessionsInThread;
372    }
373
374    private void setLogger(Logger logger) {
375        this.logger = logger;
376    }
377
378    private void setKeyPrefix(String keyPrefix) {
379        this.keyPrefix = keyPrefix;
380    }
381
382    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
383        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
384    }
385
386    private void setExpire(int expire) {
387        this.expire = expire;
388    }
389
390    private void setRedisManager(RedisManager redisManager) {
391        this.redisManager = redisManager;
392    }
393
394    private void setSessionsInThread(ThreadLocal sessionsInThread) {
395        this.sessionsInThread = sessionsInThread;
396    }
397
398    private void setLogger(Logger logger) {
399        this.logger = logger;
400    }
401
402    private void setKeyPrefix(String keyPrefix) {
403        this.keyPrefix = keyPrefix;
404    }
405
406    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
407        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
408    }
409
410    private void setExpire(int expire) {
411        this.expire = expire;
412    }
413
414    private void setRedisManager(RedisManager redisManager) {
415        this.redisManager = redisManager;
416    }
417
418    private void setSessionsInThread(ThreadLocal sessionsInThread) {
419        this.sessionsInThread = sessionsInThread;
420    }
421
422    private void setLogger(Logger logger) {
423        this.logger = logger;
424    }
425
426    private void setKeyPrefix(String keyPrefix) {
427        this.keyPrefix = keyPrefix;
428    }
429
430    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
431        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
432    }
433
434    private void setExpire(int expire) {
435        this.expire = expire;
436    }
437
438    private void setRedisManager(RedisManager redisManager) {
439        this.redisManager = redisManager;
440    }
441
442    private void setSessionsInThread(ThreadLocal sessionsInThread) {
443        this.sessionsInThread = sessionsInThread;
444    }
445
446    private void setLogger(Logger logger) {
447        this.logger = logger;
448    }
449
450    private void setKeyPrefix(String keyPrefix) {
451        this.keyPrefix = keyPrefix;
452    }
453
454    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
455        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
456    }
457
458    private void setExpire(int expire) {
459        this.expire = expire;
460    }
461
462    private void setRedisManager(RedisManager redisManager) {
463        this.redisManager = redisManager;
464    }
465
466    private void setSessionsInThread(ThreadLocal sessionsInThread) {
467        this.sessionsInThread = sessionsInThread;
468    }
469
470    private void setLogger(Logger logger) {
471        this.logger = logger;
472    }
473
474    private void setKeyPrefix(String keyPrefix) {
475        this.keyPrefix = keyPrefix;
476    }
477
478    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
479        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
480    }
481
482    private void setExpire(int expire) {
483        this.expire = expire;
484    }
485
486    private void setRedisManager(RedisManager redisManager) {
487        this.redisManager = redisManager;
488    }
489
490    private void setSessionsInThread(ThreadLocal sessionsInThread) {
491        this.sessionsInThread = sessionsInThread;
492    }
493
494    private void setLogger(Logger logger) {
495        this.logger = logger;
496    }
497
498    private void setKeyPrefix(String keyPrefix) {
499        this.keyPrefix = keyPrefix;
500    }
501
502    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
503        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
504    }
505
506    private void setExpire(int expire) {
507        this.expire = expire;
508    }
509
510    private void setRedisManager(RedisManager redisManager) {
511        this.redisManager = redisManager;
512    }
513
514    private void setSessionsInThread(ThreadLocal sessionsInThread) {
515        this.sessionsInThread = sessionsInThread;
516    }
517
518    private void setLogger(Logger logger) {
519        this.logger = logger;
520    }
521
522    private void setKeyPrefix(String keyPrefix) {
523        this.keyPrefix = keyPrefix;
524    }
525
526    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
527        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
528    }
529
530    private void setExpire(int expire) {
531        this.expire = expire;
532    }
533
534    private void setRedisManager(RedisManager redisManager) {
535        this.redisManager = redisManager;
536    }
537
538    private void setSessionsInThread(ThreadLocal sessionsInThread) {
539        this.sessionsInThread = sessionsInThread;
540    }
541
542    private void setLogger(Logger logger) {
543        this.logger = logger;
544    }
545
546    private void setKeyPrefix(String keyPrefix) {
547        this.keyPrefix = keyPrefix;
548    }
549
550    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
551        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
552    }
553
554    private void setExpire(int expire) {
555        this.expire = expire;
556    }
557
558    private void setRedisManager(RedisManager redisManager) {
559        this.redisManager = redisManager;
560    }
561
562    private void setSessionsInThread(ThreadLocal sessionsInThread) {
563        this.sessionsInThread = sessionsInThread;
564    }
565
566    private void setLogger(Logger logger) {
567        this.logger = logger;
568    }
569
570    private void setKeyPrefix(String keyPrefix) {
571        this.keyPrefix = keyPrefix;
572    }
573
574    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
575        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
576    }
577
578    private void setExpire(int expire) {
579        this.expire = expire;
580    }
581
582    private void setRedisManager(RedisManager redisManager) {
583        this.redisManager = redisManager;
584    }
585
586    private void setSessionsInThread(ThreadLocal sessionsInThread) {
587        this.sessionsInThread = sessionsInThread;
588    }
589
590    private void setLogger(Logger logger) {
591        this.logger = logger;
592    }
593
594    private void setKeyPrefix(String keyPrefix) {
595        this.keyPrefix = keyPrefix;
596    }
597
598    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
599        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
600    }
601
602    private void setExpire(int expire) {
603        this.expire = expire;
604    }
605
606    private void setRedisManager(RedisManager redisManager) {
607        this.redisManager = redisManager;
608    }
609
610    private void setSessionsInThread(ThreadLocal sessionsInThread) {
611        this.sessionsInThread = sessionsInThread;
612    }
613
614    private void setLogger(Logger logger) {
615        this.logger = logger;
616    }
617
618    private void setKeyPrefix(String keyPrefix) {
619        this.keyPrefix = keyPrefix;
620    }
621
622    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
623        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
624    }
625
626    private void setExpire(int expire) {
627        this.expire = expire;
628    }
629
630    private void setRedisManager(RedisManager redisManager) {
631        this.redisManager = redisManager;
632    }
633
634    private void setSessionsInThread(ThreadLocal sessionsInThread) {
635        this.sessionsInThread = sessionsInThread;
636    }
637
638    private void setLogger(Logger logger) {
639        this.logger = logger;
640    }
641
642    private void setKeyPrefix(String keyPrefix) {
643        this.keyPrefix = keyPrefix;
644    }
645
646    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
647        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
648    }
649
650    private void setExpire(int expire) {
651        this.expire = expire;
652    }
653
654    private void setRedisManager(RedisManager redisManager) {
655        this.redisManager = redisManager;
656    }
657
658    private void setSessionsInThread(ThreadLocal sessionsInThread) {
659        this.sessionsInThread = sessionsInThread;
660    }
661
662    private void setLogger(Logger logger) {
663        this.logger = logger;
664    }
665
666    private void setKeyPrefix(String keyPrefix) {
667        this.keyPrefix = keyPrefix;
668    }
669
670    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
671        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
672    }
673
674    private void setExpire(int expire) {
675        this.expire = expire;
676    }
677
678    private void setRedisManager(RedisManager redisManager) {
679        this.redisManager = redisManager;
680    }
681
682    private void setSessionsInThread(ThreadLocal sessionsInThread) {
683        this.sessionsInThread = sessionsInThread;
684    }
685
686    private void setLogger(Logger logger) {
687        this.logger = logger;
688    }
689
690    private void setKeyPrefix(String keyPrefix) {
691        this.keyPrefix = keyPrefix;
692    }
693
694    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
695        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
696    }
697
698    private void setExpire(int expire) {
699        this.expire = expire;
700    }
701
702    private void setRedisManager(RedisManager redisManager) {
703        this.redisManager = redisManager;
704    }
705
706    private void setSessionsInThread(ThreadLocal sessionsInThread) {
707        this.sessionsInThread = sessionsInThread;
708    }
709
710    private void setLogger(Logger logger) {
711        this.logger = logger;
712    }
713
714    private void setKeyPrefix(String keyPrefix) {
715        this.keyPrefix = keyPrefix;
716    }
717
718    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
719        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
720    }
721
722    private void setExpire(int expire) {
723        this.expire = expire;
724    }
725
726    private void setRedisManager(RedisManager redisManager) {
727        this.redisManager = redisManager;
728    }
729
730    private void setSessionsInThread(ThreadLocal sessionsInThread) {
731        this.sessionsInThread = sessionsInThread;
732    }
733
734    private void setLogger(Logger logger) {
735        this.logger = logger;
736    }
737
738    private void setKeyPrefix(String keyPrefix) {
739        this.keyPrefix = keyPrefix;
740    }
741
742    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
743        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
744    }
745
746    private void setExpire(int expire) {
747        this.expire = expire;
748    }
749
750    private void setRedisManager(RedisManager redisManager) {
751        this.redisManager = redisManager;
752    }
753
754    private void setSessionsInThread(ThreadLocal sessionsInThread) {
755        this.sessionsInThread = sessionsInThread;
756    }
757
758    private void setLogger(Logger logger) {
759        this.logger = logger;
760    }
761
762    private void setKeyPrefix(String keyPrefix) {
763        this.keyPrefix = keyPrefix;
764    }
765
766    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
767        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
768    }
769
770    private void setExpire(int expire) {
771        this.expire = expire;
772    }
773
774    private void setRedisManager(RedisManager redisManager) {
775        this.redisManager = redisManager;
776    }
777
778    private void setSessionsInThread(ThreadLocal sessionsInThread) {
779        this.sessionsInThread = sessionsInThread;
780    }
781
782    private void setLogger(Logger logger) {
783        this.logger = logger;
784    }
785
786    private void setKeyPrefix(String keyPrefix) {
787        this.keyPrefix = keyPrefix;
788    }
789
790    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
791        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
792    }
793
794    private void setExpire(int expire) {
795        this.expire = expire;
796    }
797
798    private void setRedisManager(RedisManager redisManager) {
799        this.redisManager = redisManager;
800    }
801
802    private void setSessionsInThread(ThreadLocal sessionsInThread) {
803        this.sessionsInThread = sessionsInThread;
804    }
805
806    private void setLogger(Logger logger) {
807        this.logger = logger;
808    }
809
810    private void setKeyPrefix(String keyPrefix) {
811        this.keyPrefix = keyPrefix;
812    }
813
814    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
815        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
816    }
817
818    private void setExpire(int expire) {
819        this.expire = expire;
820    }
821
822    private void setRedisManager(RedisManager redisManager) {
823        this.redisManager = redisManager;
824    }
825
826    private void setSessionsInThread(ThreadLocal sessionsInThread) {
827        this.sessionsInThread = sessionsInThread;
828    }
829
830    private void setLogger(Logger logger) {
831        this.logger = logger;
832    }
833
834    private void setKeyPrefix(String keyPrefix) {
835        this.keyPrefix = keyPrefix;
836    }
837
838    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
839        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
840    }
841
842    private void setExpire(int expire) {
843        this.expire = expire;
844    }
845
846    private void setRedisManager(RedisManager redisManager) {
847        this.redisManager = redisManager;
848    }
849
850    private void setSessionsInThread(ThreadLocal sessionsInThread) {
851        this.sessionsInThread = sessionsInThread;
852    }
853
854    private void setLogger(Logger logger) {
855        this.logger = logger;
856    }
857
858    private void setKeyPrefix(String keyPrefix) {
859        this.keyPrefix = keyPrefix;
860    }
861
862    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
863        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
864    }
865
866    private void setExpire(int expire) {
867        this.expire = expire;
868    }
869
870    private void setRedisManager(RedisManager redisManager) {
871        this.redisManager = redisManager;
872    }
873
874    private void setSessionsInThread(ThreadLocal sessionsInThread) {
875        this.sessionsInThread = sessionsInThread;
876    }
877
878    private void setLogger(Logger logger) {
879        this.logger = logger;
880    }
881
882    private void setKeyPrefix(String keyPrefix) {
883        this.keyPrefix = keyPrefix;
884    }
885
886    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
887        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
888    }
889
890    private void setExpire(int expire) {
891        this.expire = expire;
892    }
893
894    private void setRedisManager(RedisManager redisManager) {
895        this.redisManager = redisManager;
896    }
897
898    private void setSessionsInThread(ThreadLocal sessionsInThread) {
899        this.sessionsInThread = sessionsInThread;
900    }
901
902    private void setLogger(Logger logger) {
903        this.logger = logger;
904    }
905
906    private void setKeyPrefix(String keyPrefix) {
907        this.keyPrefix = keyPrefix;
908    }
909
910    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
911        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
912    }
913
914    private void setExpire(int expire) {
915        this.expire = expire;
916    }
917
918    private void setRedisManager(RedisManager redisManager) {
919        this.redisManager = redisManager;
920    }
921
922    private void setSessionsInThread(ThreadLocal sessionsInThread) {
923        this.sessionsInThread = sessionsInThread;
924    }
925
926    private void setLogger(Logger logger) {
927        this.logger = logger;
928    }
929
930    private void setKeyPrefix(String keyPrefix) {
931        this.keyPrefix = keyPrefix;
932    }
933
934    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
935        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
936    }
937
938    private void setExpire(int expire) {
939        this.expire = expire;
940    }
941
942    private void setRedisManager(RedisManager redisManager) {
943        this.redisManager = redisManager;
944    }
945
946    private void setSessionsInThread(ThreadLocal sessionsInThread) {
947        this.sessionsInThread = sessionsInThread;
948    }
949
950    private void setLogger(Logger logger) {
951        this.logger = logger;
952    }
953
954    private void setKeyPrefix(String keyPrefix) {
955        this.keyPrefix = keyPrefix;
956    }
957
958    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
959        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
960    }
961
962    private void setExpire(int expire) {
963        this.expire = expire;
964    }
965
966    private void setRedisManager(RedisManager redisManager) {
967        this.redisManager = redisManager;
968    }
969
970    private void setSessionsInThread(ThreadLocal sessionsInThread) {
971        this.sessionsInThread = sessionsInThread;
972    }
973
974    private void setLogger(Logger logger) {
975        this.logger = logger;
976    }
977
978    private void setKeyPrefix(String keyPrefix) {
979        this.keyPrefix = keyPrefix;
980    }
981
982    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
983        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
984    }
985
986    private void setExpire(int expire) {
987        this.expire = expire;
988    }
989
990    private void setRedisManager(RedisManager redisManager) {
991        this.redisManager = redisManager;
992    }
993
994    private void setSessionsInThread(ThreadLocal sessionsInThread) {
995        this.sessionsInThread = sessionsInThread;
996    }
997
998    private void setLogger(Logger logger) {
999        this.logger = logger;
1000    }
1001
1002    private void setKeyPrefix(String keyPrefix) {
1003        this.keyPrefix = keyPrefix;
1004    }
1005
1006    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
1007        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
1008    }
1009
1010    private void setExpire(int expire) {
1011        this.expire = expire;
1012    }
1013
1014    private void setRedisManager(RedisManager redisManager) {
1015        this.redisManager = redisManager;
1016    }
1017
1018    private void setSessionsInThread(ThreadLocal sessionsInThread) {
1019        this.sessionsInThread = sessionsInThread;
1020    }
1021
1022    private void setLogger(Logger logger) {
1023        this.logger = logger;
1024    }
1025
1026    private void setKeyPrefix(String keyPrefix) {
1027        this.keyPrefix = keyPrefix;
1028    }
1029
1030    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
1031        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
1032    }
1033
1034    private void setExpire(int expire) {
1035        this.expire = expire;
1036    }
1037
1038    private void setRedisManager(RedisManager redisManager) {
1039        this.redisManager = redisManager;
1040    }
1041
1042    private void setSessionsInThread(ThreadLocal sessionsInThread) {
1043        this.sessionsInThread = sessionsInThread;
1044    }
1045
1046    private void setLogger(Logger logger) {
1047        this.logger = logger;
1048    }
1049
1050    private void setKeyPrefix(String keyPrefix) {
1051        this.keyPrefix = keyPrefix;
1052    }
1053
1054    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
1055        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
1056    }
1057
1058    private void setExpire(int expire) {
1059        this.expire = expire;
1060    }
1061
1062    private void setRedisManager(RedisManager redisManager) {
1063        this.redisManager = redisManager;
1064    }
1065
1066    private void setSessionsInThread(ThreadLocal sessionsInThread) {
1067        this.sessionsInThread = sessionsInThread;
1068    }
1069
1070    private void setLogger(Logger logger) {
1071        this.logger = logger;
1072    }
1073
1074    private void setKeyPrefix(String keyPrefix) {
1075        this.keyPrefix = keyPrefix;
1076    }
1077
1078    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
1079        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
1080    }
1081
1082    private void setExpire(int expire) {
1083        this.expire = expire;
1084    }
1085
1086    private void setRedisManager(RedisManager redisManager) {
1087        this.redisManager = redisManager;
1088    }
1089
1090    private void setSessionsInThread(ThreadLocal sessionsInThread) {
1091        this.sessionsInThread = sessionsInThread;
1092    }
1093
1094    private void setLogger(Logger logger) {
1095        this.logger = logger;
1096    }
1097
1098    private void setKeyPrefix(String keyPrefix) {
1099        this.keyPrefix = keyPrefix;
1100    }
1101
1102    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
1103        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
1104    }
1105
1106    private void setExpire(int expire) {
1107        this.expire = expire;
1108    }
1109
1110    private void setRedisManager(RedisManager redisManager) {
1111        this.redisManager = redisManager;
1112    }
1113
1114    private void setSessionsInThread(ThreadLocal sessionsInThread) {
1115        this.sessionsInThread = sessionsInThread;
1116    }
1117
1118    private void setLogger(Logger logger) {
1119        this.logger = logger;
1120    }
1121
1122    private void setKeyPrefix(String keyPrefix) {
1123        this.keyPrefix = keyPrefix;
1124    }
1125
1126    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
1127        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
1128    }
1129
1130    private void setExpire(int expire) {
1131        this.expire = expire;
1132    }
1133
1134    private void setRedisManager(RedisManager redisManager) {
1135        this.redisManager = redisManager;
1136    }
1137
1138    private void setSessionsInThread(ThreadLocal sessionsInThread) {
1139        this.sessionsInThread = sessionsInThread;
1140    }
1141
1142    private void setLogger(Logger logger) {
1143        this.logger = logger;
1144    }
1145
1146    private void setKeyPrefix(String keyPrefix) {
1147        this.keyPrefix = keyPrefix;
1148    }
1149
1150    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
1151        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
1152    }
1153
1154    private void setExpire(int expire) {
1155        this.expire = expire;
1156    }
1157
1158    private void setRedisManager(RedisManager redisManager) {
1159        this.redisManager = redisManager;
1160    }
1161
1162    private void setSessionsInThread(ThreadLocal sessionsInThread) {
1163        this.sessionsInThread = sessionsInThread;
1164    }
1165
1166    private void setLogger(Logger logger) {
1167        this.logger = logger;
1168    }
1169
1170    private void setKeyPrefix(String keyPrefix) {
1171        this.keyPrefix = keyPrefix;
1172    }
1173
1174    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
1175        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
1176    }
1177
1178    private void setExpire(int expire) {
1179        this.expire = expire;
1180    }
1181
1182    private void setRedisManager(RedisManager redisManager) {
1183        this.redisManager = redisManager;
1184    }
1185
1186    private void setSessionsInThread(ThreadLocal sessionsInThread) {
1187        this.sessionsInThread = sessionsInThread;
1188    }
1189
1190    private void setLogger(Logger logger) {
1191        this.logger = logger;
1192    }
1193
1194    private void setKeyPrefix(String keyPrefix) {
1195        this.keyPrefix = keyPrefix;
1196    }
1197
1198    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
1199        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
1200    }
1201
1202    private void setExpire(int expire) {
1203        this.expire = expire;
1204    }
1205
1206    private void setRedisManager(RedisManager redisManager) {
1207        this.redisManager = redisManager;
1208    }
1209
1210    private void setSessionsInThread(ThreadLocal sessionsInThread) {
1211        this.sessionsInThread = sessionsInThread;
1212    }
1213
1214    private void setLogger(Logger logger) {
1215        this.logger = logger;
1216    }
1217
1218    private void setKeyPrefix(String keyPrefix) {
1219        this.keyPrefix = keyPrefix;
1220    }
1221
1222    private void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
1223        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
1224    }
1225
1226    private void setExpire(int expire) {
1227        this.expire = expire;
1228    }
1229
1230    private void setRedisManager(RedisManager redisManager) {
1231        this.redisManager = redisManager;
1232    }
1
```

```

71     }
72 }
73
74 /**
75  * save session
76  * @param session
77  * @throws UnknownSessionException
78  */
79 private void saveSession(Session session) throws UnknownSessionException {
80     if (session == null || session.getId() == null) {
81         logger.error("session or session id is null");
82         throw new UnknownSessionException("session or session id is null");
83     }
84     String key = getRedisSessionKey(session.getId());
85     if (expire == DEFAULT_EXPIRE) {
86         this.redisManager.set(key, session, (int) (session.getTimeout() / MILLISECONDS_IN_A_SECOND));
87         return;
88     }
89     if (expire != NO_EXPIRE && expire * MILLISECONDS_IN_A_SECOND < session.getTimeout()) {
90         logger.warn("Redis session expire time: "
91             + (expire * MILLISECONDS_IN_A_SECOND)
92             + " is less than Session timeout: "
93             + session.getTimeout()
94             + " . It may cause some problems.");
95     }
96     this.redisManager.set(key, session, expire);
97 }
98
99 @Override
100 public void delete(Session session) {
101     if (session == null || session.getId() == null) {
102         logger.error("session or session id is null");
103         return;
104     }
105     try {
106         redisManager.del(getRedisSessionKey(session.getId()));
107     } catch (Exception e) {
108         logger.error("delete session error. session id= {}", session.getId());
109     }
110 }
111
112 @Override
113 public Collection<Session> getActiveSessions() {
114     Set<Session> sessions = new HashSet<Session>();
115     try {
116         Set<String> keys = redisManager.scan(this.keyPrefix + "*");
117         if (keys != null && keys.size() > 0) {
118             for (String key:keys) {
119                 Session s = (Session) redisManager.get(key);
120                 sessions.add(s);
121             }
122         }
123     } catch (Exception e) {
124         logger.error("get active sessions error.");
125     }
126     return sessions;
127 }
128
129 public Long getActiveSessionsSize() {
130     Long size = 0L;
131     try {
132         size = redisManager.scanSize(this.keyPrefix + "*");
133     } catch (Exception e) {
134         logger.error("get active sessions error.");
135     }
136     return size;
137 }
138
139 @Override
140 protected Serializable doCreate(Session session) {
141     if (session == null) {
142         logger.error("session is null");
143         throw new UnknownSessionException("session is null");
144     }
145     Serializable sessionId = this.generateSessionId(session);
146     this.assignSessionId(session, sessionId);
147     this.saveSession(session);
148     return sessionId;
149 }

```

```
150
151 @Override
152 protected Session doReadSession(Serializable sessionId) {
153     if (sessionId == null) {
154         logger.warn("session id is null");
155         return null;
156     }
157     Session s = getSessionFromThreadLocal(sessionId);
158
159     if (s != null) {
160         return s;
161     }
162
163     logger.debug("read session from redis");
164     try {
165         s = (Session) redisManager.get(getRedisSessionKey(sessionId));
166         setSessionToThreadLocal(sessionId, s);
167     } catch (Exception e) {
168         logger.error("read session error. sessionId= {}", sessionId);
169     }
170     return s;
171 }
172
173 private void setSessionToThreadLocal(Serializable sessionId, Session s) {
174     Map<Serializable, SessionInMemory> sessionMap = (Map<Serializable, SessionInMemory>) sessionsInThread.get();
175     if (sessionMap == null) {
176         sessionMap = new HashMap<Serializable, SessionInMemory>();
177         sessionsInThread.set(sessionMap);
178     }
179     SessionInMemory sessionInMemory = new SessionInMemory();
180     sessionInMemory.setCreateTime(new Date());
181     sessionInMemory.setSession(s);
182     sessionMap.put(sessionId, sessionInMemory);
183 }
184
185 private Session getSessionFromThreadLocal(Serializable sessionId) {
186     Session s = null;
187
188     if (sessionsInThread.get() == null) {
189         return null;
190     }
191
192     Map<Serializable, SessionInMemory> sessionMap = (Map<Serializable, SessionInMemory>) sessionsInThread.get();
193     SessionInMemory sessionInMemory = sessionMap.get(sessionId);
194     if (sessionInMemory == null) {
195         return null;
196     }
197     Date now = new Date();
198     long duration = now.getTime() - sessionInMemory.getCreateTime().getTime();
199     if (duration < sessionInMemoryTimeout) {
200         s = sessionInMemory.getSession();
201         logger.debug("read session from memory");
202     } else {
203         sessionMap.remove(sessionId);
204     }
205
206     return s;
207 }
208
209 private String getRedisSessionKey(Serializable sessionId) {
210     return this.keyPrefix + sessionId;
211 }
212
213 public RedisManager getRedisManager() {
214     return redisManager;
215 }
216
217 public void setRedisManager(RedisManager redisManager) {
218     this.redisManager = redisManager;
219 }
220
221 public String getKeyPrefix() {
222     return keyPrefix;
223 }
224
225 public void setKeyPrefix(String keyPrefix) {
226     this.keyPrefix = keyPrefix;
227 }
228
```

```
229     public long getSessionInMemoryTimeout() {
230         return sessionInMemoryTimeout;
231     }
232
233     public void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
234         this.sessionInMemoryTimeout = sessionInMemoryTimeout;
235     }
236
237     public int getExpire() {
238         return expire;
239     }
240
241     public void setExpire(int expire) {
242         this.expire = expire;
243     }
244 }
```

3.Shiro配置

ShiroConfig.java

```
1  package com.springboot.test.shiro.config;
2
3  import at.pollux.thymeleaf.shiro.dialect.ShiroDialect;
4  import com.springboot.test.shiro.config.shiro.*;
5  import org.apache.shiro.codec.Base64;
6  import org.apache.shiro.session.SessionListener;
7  import org.apache.shiro.session.mgt.SessionManager;
8  import org.apache.shiro.session.mgt.eis.JavaUuidSessionIdGenerator;
9  import org.apache.shiro.session.mgt.eis.SessionDAO;
10 import org.apache.shiro.session.mgt.eis.SessionIdGenerator;
11 import org.apache.shiro.spring.LifecycleBeanPostProcessor;
12 import org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor;
13 import org.apache.shiro.spring.web.ShiroFilterFactoryBean;
14 import org.apache.shiro.mgt.SecurityManager;
15 import org.apache.shiro.web.filter.authc.FormAuthenticationFilter;
16 import org.apache.shiro.web.mgt.CookieRememberMeManager;
17 import org.apache.shiro.web.mgt.DefaultWebSecurityManager;
18 import org.apache.shiro.web.servlet.SimpleCookie;
19 import org.apache.shiro.web.session.mgt.DefaultWebSessionManager;
20 import org.springframework.beans.factory.annotation.Qualifier;
21 import org.springframework.beans.factory.config.MethodInvokingFactoryBean;
22 import org.springframework.boot.context.embedded.ConfigurableEmbeddedServletContainer;
23 import org.springframework.boot.context.embedded.EmbeddedServletContainerCustomizer;
24 import org.springframework.boot.web.servlet.ErrorPage;
25 import org.springframework.context.annotation.Bean;
26 import org.springframework.context.annotation.Configuration;
27 import org.springframework.http.HttpStatus;
28 import org.springframework.web.servlet.handler.SimpleMappingExceptionResolver;
29
30 import javax.servlet.Filter;
31 import java.util.ArrayList;
32 import java.util.Collection;
33 import java.util.LinkedHashMap;
34 import java.util.Properties;
35
36 /**
37  * @author: wangsai chao
38  * @date: 2018/5/10
39  * @description: Shiro配置
40  */
41 @Configuration
42 public class ShiroConfig {
43
44
45     /**
46      * ShiroFilterFactoryBean 处理拦截资源文件问题。
47      * 注意：初始化ShiroFilterFactoryBean的时候需要注入： SecurityManager
48      * Web应用中,Shiro可控制的Web请求必须经过Shiro主过滤器的拦截
49      * @param securityManager
50      * @return
51      */
52     @Bean(name = "shirFilter")
53     public ShiroFilterFactoryBean shiroFilter(@Qualifier("securityManager") SecurityManager securityManager) {
54
55         ShiroFilterFactoryBean shiroFilterFactoryBean = new ShiroFilterFactoryBean();
56
57         //必须设置 SecurityManager,Shiro的核心安全接口
```

```

58     shiroFilterFactoryBean.setSecurityManager(securityManager);
59     //这里的/login是后台的接口名,非页面, 如果不设置默认会自动寻找Web工程根目录下的"/login.jsp"页面
60     shiroFilterFactoryBean.setLoginUrl("/");
61     //这里的/index是后台的接口名,非页面, 登录成功后要跳转的链接
62     shiroFilterFactoryBean.setSuccessUrl("/index");
63     //未授权界面,该配置无效, 并不会进行页面跳转
64     shiroFilterFactoryBean.setUnauthorizedUrl("/unauthorized");
65
66     //自定义拦截器限制并发人数,参考博客:
67     LinkedHashMap<String, Filter> filtersMap = new LinkedHashMap<>();
68     //限制同一帐号同时在线的个数
69     filtersMap.put("kickout", kickoutSessionControlFilter());
70     //统计登录人数
71     shiroFilterFactoryBean.setFilters(filtersMap);
72
73     // 配置访问权限 必须是LinkedHashMap, 因为它必须保证有序
74     // 过滤链定义,从上向下顺序执行,一般将 /**放在最为下边 --> : 这是一个坑,一不小心代码就不好使了
75     LinkedHashMap<String, String> filterChainDefinitionMap = new LinkedHashMap<>();
76     //配置不登录可以访问的资源, anon 表示资源都可以匿名访问
77     //配置记住我或认证通过可以访问的地址
78     filterChainDefinitionMap.put("/login", "anon");
79     filterChainDefinitionMap.put("/", "anon");
80     filterChainDefinitionMap.put("/css/**", "anon");
81     filterChainDefinitionMap.put("/js/**", "anon");
82     filterChainDefinitionMap.put("/img/**", "anon");
83     filterChainDefinitionMap.put("/druid/**", "anon");
84     //解锁用户专用 测试用的
85     filterChainDefinitionMap.put("/unlockAccount", "anon");
86     filterChainDefinitionMap.put("/Captcha.jpg", "anon");
87     //logout是shiro提供的过滤器
88     filterChainDefinitionMap.put("/logout", "logout");
89     //此时访问/user/delete需要delete权限,在自定义Realm中为用户授权。
90     //filterChainDefinitionMap.put("/user/delete", "perms[\"user:delete\"]");
91
92     //其他资源都需要认证  authc 表示需要认证才能进行访问 user表示配置记住我或认证通过可以访问的地址
93     //如果开启限制同一帐号登录,改为 .put("/**", "kickout,user");
94     filterChainDefinitionMap.put("/**", "kickout,user");
95
96     shiroFilterFactoryBean.setFilterChainDefinitionMap(filterChainDefinitionMap);
97
98     return shiroFilterFactoryBean;
99 }
100
101 /**
102  * 配置核心安全事务管理器
103  * @return
104  */
105 @Bean(name="securityManager")
106 public SecurityManager securityManager() {
107     DefaultWebSecurityManager securityManager = new DefaultWebSecurityManager();
108     //设置自定义realm。
109     securityManager.setRealm(shiroRealm());
110     //配置记住我
111     securityManager.setRememberMeManager(rememberMeManager());
112     //配置redis缓存
113     securityManager.setCacheManager(cacheManager());
114     //配置自定义session管理,使用redis
115     securityManager.setSessionManager(sessionManager());
116     return securityManager;
117 }
118
119 /**
120  * 配置Shiro生命周期处理器
121  * @return
122  */
123 @Bean(name = "lifecycleBeanPostProcessor")
124 public LifecycleBeanPostProcessor lifecycleBeanPostProcessor() {
125     return new LifecycleBeanPostProcessor();
126 }
127
128 /**
129  * 身份认证realm; (这个需要自己写, 账号密码校验; 权限等)
130  * @return
131  */
132 @Bean
133 public ShiroRealm shiroRealm(){
134     ShiroRealm shiroRealm = new ShiroRealm();
135     shiroRealm.setCachingEnabled(true);
136     //启用身份验证缓存, 即缓存AuthenticationInfo信息, 默认false

```

```
137     shiroRealm.setAuthenticationCachingEnabled(true);
138     //缓存AuthenticationInfo信息的缓存名称 在ehcache-shiro.xml中有对应缓存的配置
139     shiroRealm.setAuthenticationCacheName("authenticationCache");
140     //启用授权缓存, 即缓存AuthorizationInfo信息, 默认false
141     shiroRealm.setAuthorizationCachingEnabled(true);
142     //缓存AuthorizationInfo信息的缓存名称 在ehcache-shiro.xml中有对应缓存的配置
143     shiroRealm.setAuthorizationCacheName("authorizationCache");
144     //配置自定义密码比较器
145     shiroRealm.setCredentialsMatcher(retryLimitHashedCredentialsMatcher());
146     return shiroRealm;
147 }
148
149 /**
150  * 必须 (thymeleaf页面使用shiro标签控制按钮是否显示)
151  * 未引入thymeleaf包, Caused by: java.lang.ClassNotFoundException: org.thymeleaf.dialect.AbstractProcessorDialect
152  * @return
153  */
154 @Bean
155 public ShiroDialect shiroDialect() {
156     return new ShiroDialect();
157 }
158
159 /**
160  * 开启shiro 注解模式
161  * 可以在controller中的方法前加上注解
162  * 如 @RequiresPermissions("userInfo:add")
163  * @param securityManager
164  * @return
165  */
166 @Bean
167 public AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor(@Qualifier("securityManager") SecurityManager securityManager) {
168     AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor = new AuthorizationAttributeSourceAdvisor();
169     authorizationAttributeSourceAdvisor.setSecurityManager(securityManager);
170     return authorizationAttributeSourceAdvisor;
171 }
172
173 /**
174  * 解决: 无权限页面不跳转 shiroFilterFactoryBean.setUnauthorizedUrl("/unauthorized") 无效
175  * shiro的源代码ShiroFilterFactoryBean.Java定义的filter必须满足filter instanceof AuthorizationFilter,
176  * 只有perms, roles, ssl, rest, port才是属于AuthorizationFilter, 而anon, authcBasic, authc, user是AuthenticationFilter,
177  * 所以unauthorizedUrl设置后页面不跳转 Shiro注解模式下, 登录失败与无权限都是通过抛出异常。
178  * 并且默认并没有去处理或者捕获这些异常。在SpringMVC下需要配置捕获相应异常来通知用户信息
179  * @return
180  */
181 @Bean
182 public SimpleMappingExceptionResolver simpleMappingExceptionResolver() {
183     SimpleMappingExceptionResolver simpleMappingExceptionResolver=new SimpleMappingExceptionResolver();
184     Properties properties=new Properties();
185     //这里的 /unauthorized 是页面, 不是访问的路径
186     properties.setProperty("org.apache.shiro.authz.UnauthorizedException", "/unauthorized");
187     properties.setProperty("org.apache.shiro.authz.UnauthenticatedException", "/unauthorized");
188     simpleMappingExceptionResolver.setExceptionMappings(properties);
189     return simpleMappingExceptionResolver;
190 }
191
192 /**
193  * 解决spring-boot Whitelabel Error Page
194  * @return
195  */
196 @Bean
197 public EmbeddedServletContainerCustomizer containerCustomizer() {
198
199     return new EmbeddedServletContainerCustomizer() {
200         @Override
201         public void customize(ConfigurableEmbeddedServletContainer container) {
202
203             ErrorPage error401Page = new ErrorPage(HttpStatus.UNAUTHORIZED, "/unauthorized.html");
204             ErrorPage error404Page = new ErrorPage(HttpStatus.NOT_FOUND, "/404.html");
205             ErrorPage error500Page = new ErrorPage(HttpStatus.INTERNAL_SERVER_ERROR, "/500.html");
206
207             container.addErrorPages(error401Page, error404Page, error500Page);
208         }
209     };
210 }
211
212 /**
213  * cookie对象;会话Cookie模板 ,默认为: JSESSIONID 问题: 与SERVLET容器名冲突,重新定义为sid或rememberMe, 自定义
214  * @return
215  */
```



```
216 @Bean
217 public SimpleCookie rememberMeCookie(){
218     //这个参数是cookie的名称, 对应前端的checkbox的name = rememberMe
219     SimpleCookie simpleCookie = new SimpleCookie("rememberMe");
220     //setcookie的httponly属性如果设为true的话, 会增加对xss防护的安全系数。它有以下特点:
221     //setcookie()的第七个参数
222     //设为true后, 只能通过http访问, javascript无法访问
223     //防止xss读取cookie
224     simpleCookie.setHttpOnly(true);
225     simpleCookie.setPath("/");
226     //<!-- 记住我cookie生效时间30天 ,单位秒;-->
227     simpleCookie.setMaxAge(2592000);
228     return simpleCookie;
229 }
230
231 /**
232  * cookie管理对象;记住我功能, rememberMe管理器
233  * @return
234  */
235 @Bean
236 public CookieRememberMeManager rememberMeManager(){
237     CookieRememberMeManager cookieRememberMeManager = new CookieRememberMeManager();
238     cookieRememberMeManager.setCookie(rememberMeCookie());
239     //rememberMe cookie加密的密钥 建议每个项目都不一样 默认AES算法 密钥长度(128 256 512 位)
240     cookieRememberMeManager.setCipherKey(Base64.decode("4AvVhmFLUs0KTA3Kprsdag=="));
241     return cookieRememberMeManager;
242 }
243
244 /**
245  * FormAuthenticationFilter 过滤器 过滤记住我
246  * @return
247  */
248 @Bean
249 public FormAuthenticationFilter formAuthenticationFilter(){
250     FormAuthenticationFilter formAuthenticationFilter = new FormAuthenticationFilter();
251     //对应前端的checkbox的name = rememberMe
252     formAuthenticationFilter.setRememberMeParam("rememberMe");
253     return formAuthenticationFilter;
254 }
255
256 /**
257  * shiro缓存管理器;
258  * 需要添加到securityManager中
259  * @return
260  */
261 @Bean
262 public RedisCacheManager cacheManager(){
263     RedisCacheManager redisCacheManager = new RedisCacheManager();
264     redisCacheManager.setRedisManager(redisManager());
265     //redis中针对不同用户缓存
266     redisCacheManager.setPrincipalIdFieldName("username");
267     //用户权限信息缓存时间
268     redisCacheManager.setExpire(200000);
269     return redisCacheManager;
270 }
271
272 /**
273  * 让某个实例的某个方法的返回值注入为Bean的实例
274  * Spring静态注入
275  * @return
276  */
277 @Bean
278 public MethodInvokingFactoryBean getMethodInvokingFactoryBean(){
279     MethodInvokingFactoryBean factoryBean = new MethodInvokingFactoryBean();
280     factoryBean.setStaticMethod("org.apache.shiro.SecurityUtils.setSecurityManager");
281     factoryBean.setArguments(new Object[]{securityManager()});
282     return factoryBean;
283 }
284
285 /**
286  * 配置session监听
287  * @return
288  */
289 @Bean("sessionListener")
290 public ShiroSessionListener sessionListener(){
291     ShiroSessionListener sessionListener = new ShiroSessionListener();
292     return sessionListener;
293 }
294
```

```
295     /**
296     * 配置会话ID生成器
297     * @return
298     */
299     @Bean
300     public SessionIdGenerator sessionIdGenerator() {
301         return new JavaUuidSessionIdGenerator();
302     }
303
304     @Bean
305     public RedisManager redisManager(){
306         RedisManager redisManager = new RedisManager();
307         return redisManager;
308     }
309
310     @Bean("sessionFactory")
311     public ShiroSessionFactory sessionFactory(){
312         ShiroSessionFactory sessionFactory = new ShiroSessionFactory();
313         return sessionFactory;
314     }
315
316     /**
317     * SessionDAO的作用是为Session提供CRUD并进行持久化的一个shiro组件
318     * MemorySessionDAO 直接在内存中进行会话维护
319     * EnterpriseCacheSessionDAO 提供了缓存功能的会话维护，默认情况下使用MapCache实现，内部使用ConcurrentHashMap保存缓存的会话。
320     * @return
321     */
322     @Bean
323     public SessionDAO sessionDAO() {
324         RedisSessionDAO redisSessionDAO = new RedisSessionDAO();
325         redisSessionDAO.setRedisManager(redisManager());
326         //session在redis中的保存时间,最好大于session会话超时时间
327         redisSessionDAO.setExpire(12000);
328         return redisSessionDAO;
329     }
330
331     /**
332     * 配置保存sessionId的cookie
333     * 注意：这里的cookie 不是上面的记住我 cookie 记住我需要一个cookie session管理 也需要自己的cookie
334     * 默认为： JSESSIONID 问题：与SERVLET容器名冲突,重新定义为sid
335     * @return
336     */
337     @Bean("sessionIdCookie")
338     public SimpleCookie sessionIdCookie(){
339         //这个参数是cookie的名称
340         SimpleCookie simpleCookie = new SimpleCookie("sid");
341         //setcookie的httponly属性如果设为true的话，会增加对xss防护的安全系数。它有以下特点：
342
343         //setcookie()的第七个参数
344         //设为true后，只能通过http访问，javascript无法访问
345         //防止xss读取cookie
346         simpleCookie.setHttpOnly(true);
347         simpleCookie.setPath("/");
348         //maxAge=-1表示浏览器关闭时失效此Cookie
349         simpleCookie.setMaxAge(-1);
350         return simpleCookie;
351     }
352
353     /**
354     * 配置会话管理器，设定会话超时及保存
355     * @return
356     */
357     @Bean("sessionManager")
358     public SessionManager sessionManager() {
359         ShiroSessionManager sessionManager = new ShiroSessionManager();
360         Collection<SessionListener> listeners = new ArrayList<SessionListener>();
361         //配置监听
362         listeners.add(sessionListener());
363         sessionManager.setSessionListeners(listeners);
364         sessionManager.setSessionIdCookie(sessionIdCookie());
365         sessionManager.setSessionDAO(sessionDAO());
366         sessionManager.setCacheManager(cacheManager());
367         sessionManager.setSessionFactory(sessionFactory());
368
369         //全局会话超时时间（单位毫秒），默认30分钟 暂时设置为10秒钟 用来测试
370         sessionManager.setGlobalSessionTimeout(1800000);
371         //是否开启删除无效的session对象 默认为true
372         sessionManager.setDeleteInvalidSessions(true);
373         //是否开启定时调度器进行检测过期session 默认为true
```

```

374     sessionManager.setSessionValidationSchedulerEnabled(true);
375     //设置session失效的扫描时间，清理用户直接关闭浏览器造成的孤立会话 默认为 1个小时
376     //设置该属性 就不需要设置 ExecutorServiceSessionValidationScheduler 底层也是默认自动调用ExecutorServiceSessionValidationScheduler
377     //暂时设置为 5秒 用来测试
378     sessionManager.setSessionValidationInterval(3600000);
379     //取消url 后面的 JSESSIONID
380     sessionManager.setSessionIdUrlRewritingEnabled(false);
381     return sessionManager;
382
383 }
384
385 /**
386  * 并发登录控制
387  * @return
388  */
389 @Bean
390 public KickoutSessionControlFilter kickoutSessionControlFilter(){
391     KickoutSessionControlFilter kickoutSessionControlFilter = new KickoutSessionControlFilter();
392     //用于根据会话ID，获取会话进行踢出操作的；
393     kickoutSessionControlFilter.setSessionManager(sessionManager());
394     //使用cacheManager获取相应的cache来缓存用户登录的会话；用于保存用户-会话之间的关系的；
395     kickoutSessionControlFilter.setRedisManager(redisManager());
396     //是否踢出后来登录的，默认是false；即后者登录的用户踢出前者登录的用户；
397     kickoutSessionControlFilter.setKickoutAfter(false);
398     //同一个用户最大的会话数，默认1；比如2的意思是同一个用户允许最多同时两个人登录；
399     kickoutSessionControlFilter.setMaxSession(1);
400     //被踢出后重定向到的地址；
401     kickoutSessionControlFilter.setKickoutUrl("/login?kickout=1");
402     return kickoutSessionControlFilter;
403 }
404
405 /**
406  * 配置密码比较器
407  * @return
408  */
409 @Bean("credentialsMatcher")
410 public RetryLimitHashedCredentialsMatcher retryLimitHashedCredentialsMatcher(){
411     RetryLimitHashedCredentialsMatcher retryLimitHashedCredentialsMatcher = new RetryLimitHashedCredentialsMatcher();
412     retryLimitHashedCredentialsMatcher.setRedisManager(redisManager());
413
414     //如果密码加密,可以打开下面配置
415     //加密算法的名称
416     //retryLimitHashedCredentialsMatcher.setHashAlgorithmName("MD5");
417     //配置加密的次数
418     //retryLimitHashedCredentialsMatcher.setHashIterations(1024);
419     //是否存储为16进制
420     //retryLimitHashedCredentialsMatcher.setStoredCredentialsHexEncoded(true);
421
422     return retryLimitHashedCredentialsMatcher;
423 }
424
425 }

```

ShiroRealm.java

```

1 package com.springboot.test.shiro.config.shiro;
2
3 import com.springboot.test.shiro.modules.user.dao.PermissionMapper;
4 import com.springboot.test.shiro.modules.user.dao.RoleMapper;
5 import com.springboot.test.shiro.modules.user.dao.entity.Permission;
6 import com.springboot.test.shiro.modules.user.dao.entity.Role;
7 import com.springboot.test.shiro.modules.user.dao.UserMapper;
8 import com.springboot.test.shiro.modules.user.dao.entity.User;
9 import org.apache.shiro.SecurityUtils;
10 import org.apache.shiro.authc.*;
11 import org.apache.shiro.authz.AuthorizationInfo;
12 import org.apache.shiro.authz.SimpleAuthorizationInfo;
13 import org.apache.shiro.realm.AuthorizingRealm;
14 import org.apache.shiro.subject.PrincipalCollection;
15 import org.springframework.beans.factory.annotation.Autowired;
16
17 import java.util.Set;
18 import java.util.concurrent.ConcurrentHashMap;
19
20 /**
21  * @author: wangsaichao
22  * @date: 2018/5/10
23  * @description: 在Shiro中，最终是通过Realm来获取应用程序中的用户、角色及权限信息的

```

```
24  * 在Realm中会直接从我们的数据源中获取Shiro需要的验证信息。可以说, Realm是专用于安全框架的DAO。
25  */
26  public class ShiroRealm extends AuthorizingRealm {
27
28      @Autowired
29      private UserMapper userMapper;
30
31      @Autowired
32      private RoleMapper roleMapper;
33
34      @Autowired
35      private PermissionMapper permissionMapper;
36
37      /**
38       * 验证用户身份
39       * @param authenticationToken
40       * @return
41       * @throws AuthenticationException
42       */
43      @Override
44      protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) throws AuthenticationException {
45
46          //获取用户名密码 第一种方式
47          //String username = (String) authenticationToken.getPrincipal();
48          //String password = new String((char[]) authenticationToken.getCredentials());
49
50          //获取用户名 密码 第二种方式
51          UsernamePasswordToken usernamePasswordToken = (UsernamePasswordToken) authenticationToken;
52          String username = usernamePasswordToken.getUsername();
53          String password = new String(usernamePasswordToken.getPassword());
54
55          //从数据库查询用户信息
56          User user = this.userMapper.findByUserName(username);
57
58          //可以在这里直接对用户名称校验,或者调用 CredentialsMatcher 校验
59          if (user == null) {
60              throw new UnknownAccountException("用户名或密码错误! ");
61          }
62          //这里将 密码对比 注销掉,否则 无法锁定 要将密码对比 交给 密码比较器
63          //if (!password.equals(user.getPassword())) {
64          //    throw new IncorrectCredentialsException("用户名或密码错误! ");
65          //}
66          if ("1".equals(user.getState())) {
67              throw new LockedAccountException("账号已被锁定,请联系管理员! ");
68          }
69
70          //调用 CredentialsMatcher 校验 还需要创建一个类 继承CredentialsMatcher 如果在上面校验了,这个就不需要了
71          //配置自定义权限登录器 参考博客:
72
73          SimpleAuthenticationInfo info = new SimpleAuthenticationInfo(user, user.getPassword(), getName());
74          return info;
75      }
76
77      /**
78       * 授权用户权限
79       * 授权的方法是在碰到<shiro:hasPermission name=''></shiro:hasPermission>标签的时候调用的
80       * 它会去检测shiro框架中的权限(这里的permissions)是否包含有该标签的name值,如果有,里面的内容显示
81       * 如果没有,里面的内容不予显示(这就完成了对于权限的认证。)
82       *
83       * shiro的权限授权是通过继承AuthorizingRealm抽象类,重载doGetAuthorizationInfo();
84       * 当访问到页面的时候, 链接配置了相应的权限或者shiro标签才会执行此方法否则不会执行
85       * 所以如果只是简单的身份认证没有权限的控制的话, 那么这个方法可以不进行实现, 直接返回null即可。
86       *
87       * 在这个方法中主要是使用类: SimpleAuthorizationInfo 进行角色的添加和权限的添加。
88       * authorizationInfo.addRole(role.getRole()); authorizationInfo.addStringPermission(p.getPermission());
89       *
90       * 当然也可以添加set集合: roles是从数据库查询的当前用户的角色, stringPermissions是从数据库查询的当前用户对应的权限
91       * authorizationInfo.setRoles(roles); authorizationInfo.setStringPermissions(stringPermissions);
92       *
93       * 就是说如果在shiro配置文件中添加了filterChainDefinitionMap.put("/add", "perms[权限添加]");
94       * 就说明访问/add这个链接必须要有“权限添加”这个权限才可以访问
95       *
96       * 如果在shiro配置文件中添加了filterChainDefinitionMap.put("/add", "roles[100002], perms[权限添加]");
97       * 就说明访问/add这个链接必须要有 “权限添加” 这个权限和具有 “100002” 这个角色才可以访问
98       * @param principalCollection
99       * @return
100      */
101      @Override
102      protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {
```

```
103
104     System.out.println("查询权限方法调用了!!!");
105
106     //获取用户
107     User user = (User) SecurityUtils.getSubject().getPrincipal();
108
109     //获取用户角色
110     Set<Role> roles =this.roleMapper.findRolesByUserId(user.getUid());
111     //添加角色
112     SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();
113     for (Role role : roles) {
114         authorizationInfo.addRole(role.getRole());
115     }
116
117     //获取用户权限
118     Set<Permission> permissions = this.permissionMapper.findPermissionsByRoleId(roles);
119     //添加权限
120     for (Permission permission:permissions) {
121         authorizationInfo.addStringPermission(permission.getPermission());
122     }
123
124     return authorizationInfo;
125 }
126
127 /**
128  * 重写方法,清除当前用户的 授权缓存
129  * @param principals
130  */
131 @Override
132 public void clearCachedAuthorizationInfo(PrincipalCollection principals) {
133     super.clearCachedAuthorizationInfo(principals);
134 }
135
136 /**
137  * 重写方法,清除当前用户的 认证缓存
138  * @param principals
139  */
140 @Override
141 public void clearCachedAuthenticationInfo(PrincipalCollection principals) {
142     super.clearCachedAuthenticationInfo(principals);
143 }
144
145 @Override
146 public void clearCache(PrincipalCollection principals) {
147     super.clearCache(principals);
148 }
149
150 /**
151  * 自定义方法: 清除所有 授权缓存
152  */
153 public void clearAllCachedAuthorizationInfo() {
154     getAuthorizationCache().clear();
155 }
156
157 /**
158  * 自定义方法: 清除所有 认证缓存
159  */
160 public void clearAllCachedAuthenticationInfo() {
161     getAuthenticationCache().clear();
162 }
163
164 /**
165  * 自定义方法: 清除所有的 认证缓存 和 授权缓存
166  */
167 public void clearAllCache() {
168     clearAllCachedAuthenticationInfo();
169     clearAllCachedAuthorizationInfo();
170 }
171
172 }
```

KickoutSessionControlFilter.java(限制 并发 登录人数)

```
1 package com.springboot.test.shiro.config.shiro;
2
3 import java.io.Serializable;
4 import java.util.Deque;
5 import java.util.LinkedList;
```

```
6 import javax.servlet.ServletException;
7 import javax.servlet.ServletResponse;
8 import javax.servlet.http.HttpServletRequest;
9
10 import com.springboot.test.shiro.modules.user.dao.entity.User;
11 import org.apache.shiro.session.Session;
12 import org.apache.shiro.session.mgt.DefaultSessionKey;
13 import org.apache.shiro.session.mgt.SessionManager;
14 import org.apache.shiro.subject.Subject;
15 import org.apache.shiro.web.filter.AccessControlFilter;
16 import org.apache.shiro.web.util.WebUtils;
17 import org.springframework.beans.factory.annotation.Autowired;
18 import org.springframework.web.servlet.resource.ResourceUrlProvider;
19
20 /**
21  * @author: WangSaiChao
22  * @date: 2018/5/23
23  * @description: shiro 自定义filter 实现 并发登录控制
24  */
25 public class KickoutSessionControlFilter extends AccessControlFilter{
26
27     @Autowired
28     private ResourceUrlProvider resourceUrlProvider;
29
30     /** 踢出后到的地址 */
31     private String kickoutUrl;
32
33     /** 踢出之前登录的/之后登录的用户 默认踢出之前登录的用户 */
34     private boolean kickoutAfter = false;
35
36     /** 同一个帐号最大会话数 默认1 */
37     private int maxSession = 1;
38     private SessionManager sessionManager;
39
40     private RedisManager redisManager;
41
42     public static final String DEFAULT_KICKOUT_CACHE_KEY_PREFIX = "shiro:cache:kickout:";
43     private String keyPrefix = DEFAULT_KICKOUT_CACHE_KEY_PREFIX;
44
45     public void setKickoutUrl(String kickoutUrl) {
46         this.kickoutUrl = kickoutUrl;
47     }
48
49     public void setKickoutAfter(boolean kickoutAfter) {
50         this.kickoutAfter = kickoutAfter;
51     }
52
53     public void setMaxSession(int maxSession) {
54         this.maxSession = maxSession;
55     }
56
57     public void setSessionManager(SessionManager sessionManager) {
58         this.sessionManager = sessionManager;
59     }
60
61     public void setRedisManager(RedisManager redisManager) {
62         this.redisManager = redisManager;
63     }
64
65     public String getKeyPrefix() {
66         return keyPrefix;
67     }
68
69     public void setKeyPrefix(String keyPrefix) {
70         this.keyPrefix = keyPrefix;
71     }
72
73     private String getRedisKickoutKey(String username) {
74         return this.keyPrefix + username;
75     }
76
77     /**
78      * 是否允许访问, 返回true表示允许
79      */
80     @Override
81     protected boolean isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) throws Exception {
82         return false;
83     }
84     /**
```

```
85      * 表示访问拒绝时是否自己处理, 如果返回true表示自己不处理且继续拦截器链执行, 返回false表示自己已经处理了 (比如重定向到另一个页面)。  
86      */  
87      @Override  
88      protected boolean onAccessDenied(ServletRequest request, ServletResponse response) throws Exception {  
89          Subject subject = getSubject(request, response);  
90          if(!subject.isAuthenticated() && !subject.isRemembered()) {  
91              //如果没有登录, 直接进行之后的流程  
92              return true;  
93          }  
94  
95          //如果有登录, 判断是否访问的为静态资源, 如果是游客允许访问的静态资源, 直接返回true  
96          HttpServletRequest httpRequest = (HttpServletRequest) request;  
97          String path = httpRequest.getServletPath();  
98          // 如果是静态文件, 则返回true  
99          if (isStaticFile(path)){  
100              return true;  
101          }  
102  
103  
104          Session session = subject.getSession();  
105          //这里获取的User是实体 因为我在 自定义ShiroRealm中的doGetAuthenticationInfo方法中  
106          //new SimpleAuthenticationInfo(user, password, getName()); 传的是 User实体 所以这里拿到的也是实体, 如果传的是userName 这里拿到的就是user!  
107          String username = ((User) subject.getPrincipal()).getUsername();  
108          Serializable sessionId = session.getId();  
109  
110          // 初始化用户的队列放到缓存里  
111          Deque<Serializable> deque = (Deque<Serializable>) redisManager.get(getRedisKickoutKey(username));  
112          if(deque == null || deque.size()==0) {  
113              deque = new LinkedList<Serializable>();  
114          }  
115  
116          //如果队列里没有此sessionId, 且用户没有被踢出; 放入队列  
117          if(!deque.contains(sessionId) && session.getAttribute("kickout") == null) {  
118              deque.push(sessionId);  
119          }  
120  
121          //如果队列里的sessionId数超出最大会话数, 开始踢人  
122          while(deque.size() > maxSession) {  
123              Serializable kickoutSessionId = null;  
124              if(kickoutAfter) { //如果踢出后者  
125                  kickoutSessionId = deque.getFirst();  
126                  kickoutSessionId = deque.removeFirst();  
127              } else { //否则踢出前者  
128                  kickoutSessionId = deque.removeLast();  
129              }  
130              try {  
131                  Session kickoutSession = sessionManager.getSession(new DefaultSessionKey(kickoutSessionId));  
132                  if(kickoutSession != null) {  
133                      //设置会话的kickout属性表示踢出了  
134                      kickoutSession.setAttribute("kickout", true);  
135                  }  
136              } catch (Exception e) { //ignore exception  
137                  e.printStackTrace();  
138              }  
139          }  
140  
141          redisManager.set(getRedisKickoutKey(username), deque);  
142  
143          //如果被踢出了, 直接退出, 重定向到踢出后的地址  
144          if (session.getAttribute("kickout") != null) {  
145              //会话被踢出了  
146              try {  
147                  subject.logout();  
148              } catch (Exception e) {  
149              }  
150              WebUtils.issueRedirect(request, response, kickoutUrl);  
151              return false;  
152          }  
153          return true;  
154      }  
155  
156      private boolean isStaticFile(String path) {  
157          String staticUri = resourceUrlProvider.getForLookupPath(path);  
158          return staticUri != null;  
159      }  
160  
161  }
```

RetryLimitHashedCredentialsMatcher.java(登录错误次数限制)

```
1 package com.springboot.test.shiro.config.shiro;
2
3 import java.util.concurrent.atomic.AtomicInteger;
4
5 import com.springboot.test.shiro.modules.user.dao.UserMapper;
6 import com.springboot.test.shiro.modules.user.dao.entity.User;
7 import org.apache.log4j.Logger;
8 import org.apache.shiro.authc.AuthenticationInfo;
9 import org.apache.shiro.authc.AuthenticationToken;
10 import org.apache.shiro.authc.LockedAccountException;
11 import org.apache.shiro.authc.credential.SimpleCredentialsMatcher;
12 import org.apache.shiro.cache.Cache;
13 import org.apache.shiro.cache.CacheManager;
14 import org.springframework.beans.factory.annotation.Autowired;
15
16
17 /**
18  * @author: WangSaiChao
19  * @date: 2018/5/25
20  * @description: 登陆次数限制
21  */
22 public class RetryLimitHashedCredentialsMatcher extends SimpleCredentialsMatcher {
23
24     private static final Logger logger = Logger.getLogger(RetryLimitHashedCredentialsMatcher.class);
25
26     public static final String DEFAULT_RETRYLIMIT_CACHE_KEY_PREFIX = "shiro:cache:retrylimit:";
27     private String keyPrefix = DEFAULT_RETRYLIMIT_CACHE_KEY_PREFIX;
28     @Autowired
29     private UserMapper userMapper;
30     private RedisManager redisManager;
31
32     public void setRedisManager(RedisManager redisManager) {
33         this.redisManager = redisManager;
34     }
35
36     private String getRedisKickoutKey(String username) {
37         return this.keyPrefix + username;
38     }
39
40     @Override
41     public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {
42
43         //获取用户名
44         String username = (String)token.getPrincipal();
45         //获取用户登录次数
46         AtomicInteger retryCount = (AtomicInteger)redisManager.get(getRedisKickoutKey(username));
47         if (retryCount == null) {
48             //如果用户没有登陆过,登陆次数加1 并放入缓存
49             retryCount = new AtomicInteger(0);
50         }
51         if (retryCount.incrementAndGet() > 5) {
52             //如果用户登陆失败次数大于5次 抛出锁定用户异常 并修改数据库字段
53             User user = userMapper.findByUserName(username);
54             if (user != null && "0".equals(user.getState())){
55                 //数据库字段 默认为 0 就是正常状态 所以 要改为1
56                 //修改数据库的状态字段为锁定
57                 user.setState("1");
58                 userMapper.update(user);
59             }
60             logger.info("锁定用户" + user.getUsername());
61             //抛出用户锁定异常
62             throw new LockedAccountException();
63         }
64         //判断用户账号和密码是否正确
65         boolean matches = super.doCredentialsMatch(token, info);
66         if (matches) {
67             //如果正确,从缓存中将用户登录计数 清除
68             redisManager.del(getRedisKickoutKey(username));
69         }{
70             redisManager.set(getRedisKickoutKey(username), retryCount);
71         }
72         return matches;
73     }
74
75     /**
76      * 根据用户名 解锁用户
77      * @param username
```



```
78     * @return
79     */
80     public void unlockAccount(String username){
81         User user = userMapper.findByUserName(username);
82         if (user != null){
83             //修改数据库的状态字段为锁定
84             user.setState("0");
85             userMapper.update(user);
86             redisManager.del(getRedisKickoutKey(username));
87         }
88     }
89
90 }
```

ShiroSessionListener.java(session 监听)

```
1  package com.springboot.test.shiro.config.shiro;
2
3  import com.springboot.test.shiro.Application;
4  import com.springboot.test.shiro.modules.user.dao.entity.User;
5  import org.apache.shiro.SecurityUtils;
6  import org.apache.shiro.session.Session;
7  import org.apache.shiro.session.SessionListener;
8  import org.springframework.beans.factory.annotation.Autowired;
9
10 import javax.servlet.ServletContextEvent;
11 import javax.servlet.ServletContextListener;
12 import javax.servlet.http.HttpSessionAttributeListener;
13 import javax.servlet.http.HttpSessionBindingEvent;
14 import java.util.concurrent.ConcurrentHashMap;
15 import java.util.concurrent.atomic.AtomicInteger;
16
17 /**
18  * @author: wangsaihao
19  * @date: 2018/5/15
20  * @description: 配置session监听器,
21  */
22 public class ShiroSessionListener implements SessionListener{
23
24     /**
25      * 统计在线人数
26      * juc包下线程安全自增
27      */
28     private final AtomicInteger sessionCount = new AtomicInteger(0);
29
30     /**
31      * 会话创建时触发
32      * @param session
33      */
34     @Override
35     public void onStart(Session session) {
36         //会话创建, 在线人数加一
37         sessionCount.incrementAndGet();
38     }
39
40     /**
41      * 退出会话时触发
42      * @param session
43      */
44     @Override
45     public void onStop(Session session) {
46         //会话退出, 在线人数减一
47         sessionCount.decrementAndGet();
48     }
49
50     /**
51      * 会话过期时触发
52      * @param session
53      */
54     @Override
55     public void onExpiration(Session session) {
56         //会话过期, 在线人数减一
57         sessionCount.decrementAndGet();
58     }
59
60     /**
61      * 获取在线人数使用
62      * @return
```

2022/6/5 11:21

(48条消息) Shiro使用redis作为缓存(解决shiro频繁访问Redis)(十一)_这个名字想了很久的博客-CSDN博客_shiro 使用redis

```
63      */
64      public AtomicInteger getSessionCount() {
65          return sessionCount;
66      }
67  }
```

上面的类中有一些依赖类,并没有贴出来,该类是为了解决Shiro整合Redis 频繁获取或更新 Session 将在下一篇博客中讲,依赖的一些类,也在下篇博客中贴出来。点击进入下一篇博客: