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

```java
package com.springboot.test.shiro.global.utils;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.redis.serializer.RedisSerializer;
import org.springframework.data.redis.serializer.SerializationException;

import java.io.*;

/**
 * @author: wangsaichao
 * @date: 2018/6/20
 * @description: redis的value序列化工具
 */
public class SerializeUtils implements RedisSerializer {

    private static Logger logger = LoggerFactory.getLogger(SerializeUtils.class);

    public static boolean isEmpty(byte[] data) {
        return (data == null || data.length == 0);
    }

    /**
     * 序列化
     * @param object
     * @return
     * @throws SerializationException
     */
    @Override
    public byte[] serialize(Object object) throws SerializationException {
        byte[] result = null;

        if (object == null) {
            return new byte[0];
        }
        try (
                ByteArrayOutputStream byteStream = new ByteArrayOutputStream(128);
                ObjectOutputStream objectOutputStream = new ObjectOutputStream(byteStream)
        ){

            if (!(object instanceof Serializable)) {
                throw new IllegalArgumentException(SerializeUtils.class.getSimpleName() + " requires a Serializable payload " +
                        "but received an object of type [" + object.getClass().getName() + "]");
            }

            objectOutputStream.writeObject(object);
            objectOutputStream.flush();
            result =  byteStream.toByteArray();
        } catch (Exception ex) {
            logger.error("Failed to serialize",ex);
        }
        return result;
    }

    /**
     * 反序列化
     * @param bytes
     * @return
     * @throws SerializationException
     */
    @Override
    public Object deserialize(byte[] bytes) throws SerializationException {

        Object result = null;

        if (isEmpty(bytes)) {
            return null;
        }

        try (
                ByteArrayInputStream byteStream = new ByteArrayInputStream(bytes);
                ObjectInputStream objectInputStream = new ObjectInputStream(byteStream)
        ){
            result = objectInputStream.readObject();
        } catch (Exception e) {
            logger.error("Failed to deserialize",e);
        }
        return result;
    }

}
```

RedisConfig.java

```java
package com.springboot.test.shiro.config;

import com.springboot.test.shiro.global.utils.SerializeUtils;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.StringRedisSerializer;
import redis.clients.jedis.JedisPoolConfig;

/**
 * @author: wangsaichao
 * @date: 2017/11/23
 * @description: redis配置
 */
@Configuration
public class RedisConfig {

    /**
     * redis地址
     */
    @Value("${spring.redis.host}")
    private String host;

    /**
     * redis端口号
     */
    @Value("${spring.redis.port}")
    private Integer port;

    /**
     * redis密码
     */
    @Value("${spring.redis.password}")
    private String password;

    /**
     * JedisPoolConfig 连接池
     * @return
     */
    @Bean
    public JedisPoolConfig jedisPoolConfig(){
        JedisPoolConfig jedisPoolConfig=new JedisPoolConfig();
        //最大空闲数
        jedisPoolConfig.setMaxIdle(300);
        //连接池的最大数据库连接数
        jedisPoolConfig.setMaxTotal(1000);
        //最大建立连接等待时间
        jedisPoolConfig.setMaxWaitMillis(1000);
        //逐出连接的最小空闲时间 默认1800000毫秒(30分钟)
        jedisPoolConfig.setMinEvictableIdleTimeMillis(300000);
        //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
        jedisPoolConfig.setNumTestsPerEvictionRun(10);
        //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
        jedisPoolConfig.setTimeBetweenEvictionRunsMillis(30000);
        //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
        jedisPoolConfig.setTestOnBorrow(true);
        //在空闲时检查有效性, 默认false
        jedisPoolConfig.setTestWhileIdle(true);
        return jedisPoolConfig;
    }

    /**
     * 配置工厂
     * @param jedisPoolConfig
     * @return
     */
    @Bean
    public JedisConnectionFactory jedisConnectionFactory(JedisPoolConfig jedisPoolConfig){
        JedisConnectionFactory jedisConnectionFactory=new JedisConnectionFactory();
        //连接池
        jedisConnectionFactory.setPoolConfig(jedisPoolConfig);
        //IP地址
        jedisConnectionFactory.setHostName(host);
        //端口号
        jedisConnectionFactory.setPort(port);
        //如果Redis设置有密码
        jedisConnectionFactory.setPassword(password);
        //客户端超时时间单位是毫秒
        jedisConnectionFactory.setTimeout(5000);
        return jedisConnectionFactory;
    }

    /**
     * shiro redis缓存使用的模板
     * 实例化 RedisTemplate 对象
     * @return
     */
    @Bean("shiroRedisTemplate")
    public RedisTemplate shiroRedisTemplate(RedisConnectionFactory redisConnectionFactory) {

        RedisTemplate redisTemplate = new RedisTemplate();
        redisTemplate.setKeySerializer(new StringRedisSerializer());
        redisTemplate.setHashKeySerializer(new StringRedisSerializer());
        redisTemplate.setHashValueSerializer(new SerializeUtils());
        redisTemplate.setValueSerializer(new SerializeUtils());
        //开启事务
        //stringRedisTemplate.setEnableTransactionSupport(true);
        redisTemplate.setConnectionFactory(redisConnectionFactory);
        return redisTemplate;
    }

}
```

RedisManager.java

```
package com.springboot.test.shiro.config.shiro;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.dao.DataAccessException;
import org.springframework.data.redis.connection.RedisConnection;
import org.springframework.data.redis.core.*;
import org.springframework.util.CollectionUtils;

import java.util.*;
import java.util.concurrent.TimeUnit;

/**
 *
 * @author wangsaichao
 * 基于spring和redis的redisTemplate工具类
 */
public class RedisManager {

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    //=============================common============================
    /**
     * 指定缓存失效时间
     * @param key 键
     * @param time 时间(秒)
```

```java
     */
    public void expire(String key,long time){
        redisTemplate.expire(key, time, TimeUnit.SECONDS);
    }

    /**
     * 判断key是否存在
     * @param key 键
     * @return true 存在 false不存在
     */
    public Boolean hasKey(String key){
        return redisTemplate.hasKey(key);
    }

    /**
     * 删除缓存
     * @param key 可以传一个值 或多个
     */
    @SuppressWarnings("unchecked")
    public void del(String ... key){
        if(key!=null&&key.length>0){
            if(key.length==1){
                redisTemplate.delete(key[0]);
            }else{
                redisTemplate.delete(CollectionUtils.arrayToList(key));
            }
        }
    }

    /**
     * 批量删除key
     * @param keys
     */
    public void del(Collection keys){
        redisTemplate.delete(keys);
    }

    //============================String=============================
    /**
     * 普通缓存获取
     * @param key 键
     * @return 值
     */
    public Object get(String key){
        return redisTemplate.opsForValue().get(key);
    }

    /**
     * 普通缓存放入
     * @param key 键
     * @param value 值
     */
    public void set(String key,Object value) {
        redisTemplate.opsForValue().set(key, value);
    }

    /**
     * 普通缓存放入并设置时间
     * @param key 键
     * @param value 值
     * @param time 时间(秒) time要大于0 如果time小于等于0 将设置无限期
     */
    public void set(String key,Object value,long time){
        if(time>0){
            redisTemplate.opsForValue().set(key, value, time, TimeUnit.SECONDS);
        }else{
            set(key, value);
        }
    }

    /**
     * 使用scan命令 查询某些前缀的key
     * @param key
     * @return
     */
    public Set<String> scan(String key){
        Set<String> execute = this.redisTemplate.execute(new RedisCallback<Set<String>>() {

            @Override
            public Set<String> doInRedis(RedisConnection connection) throws DataAccessException {

                Set<String> binaryKeys = new HashSet<>();

                Cursor<byte[]> cursor = connection.scan(new ScanOptions.ScanOptionsBuilder().match(key).count(1000).build());
                while (cursor.hasNext()) {
                    binaryKeys.add(new String(cursor.next()));
                }
                return binaryKeys;
            }
        });
        return execute;
    }

    /**
     * 使用scan命令 查询某些前缀的key 有多少个
     * 用来获取当前session数量,也就是在线用户
     * @param key
     * @return
     */
    public Long scanSize(String key){
        long dbSize = this.redisTemplate.execute(new RedisCallback<Long>() {

            @Override
            public Long doInRedis(RedisConnection connection) throws DataAccessException {
                long count = 0L;
                Cursor<byte[]> cursor = connection.scan(ScanOptions.scanOptions().match(key).count(1000).build());
                while (cursor.hasNext()) {
                    cursor.next();
                    count++;
                }
                return count;
            }
        });
        return dbSize;
    }
}
```

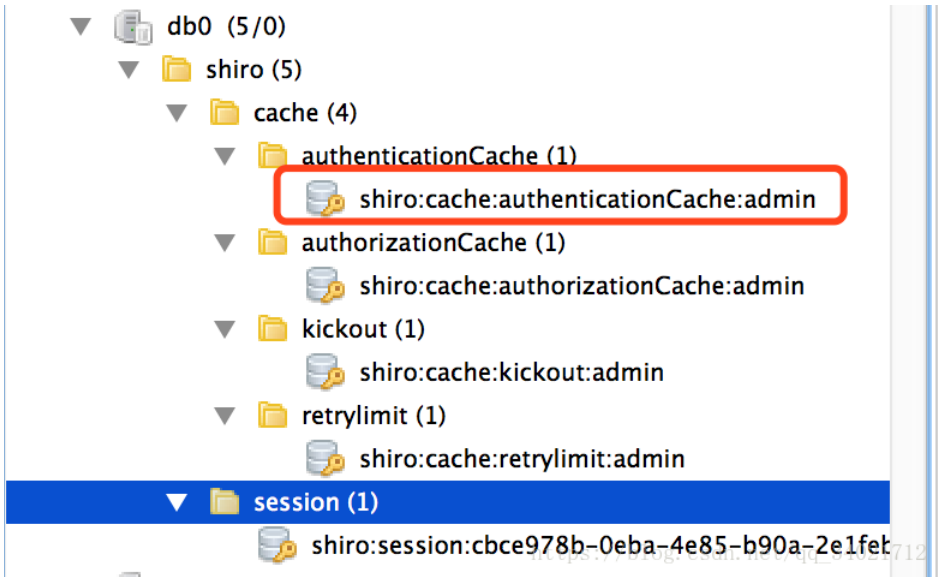
2.使用Redis作为缓存需要shiro 重写cache、cacheManager、SessionDAO

RedisCache.java

```java
package com.springboot.test.shiro.config.shiro;

import com.springboot.test.shiro.global.exceptions.PrincipalIdNullException;
import com.springboot.test.shiro.global.exceptions.PrincipalInstanceException;
import org.apache.shiro.cache.Cache;
import org.apache.shiro.cache.CacheException;
import org.apache.shiro.subject.PrincipalCollection;
import org.apache.shiro.util.CollectionUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.*;

/**
 * @author: wangsaichao
 * @date: 2018/6/22
 * @description: 参考 shiro-redis 开源项目 Git地址 https://github.com/alexxiyang/shiro-redis
 */
public class RedisCache<K, V> implements Cache<K, V> {

    private static Logger logger = LoggerFactory.getLogger(RedisCache.class);

    private RedisManager redisManager;
    private String keyPrefix = "";
    private int expire = 0;
    private String principalIdFieldName = RedisCacheManager.DEFAULT_PRINCIPAL_ID_FIELD_NAME;

    /**
     * Construction
     * @param redisManager
     */
    public RedisCache(RedisManager redisManager, String prefix, int expire, String principalIdFieldName) {
        if (redisManager == null) {
            throw new IllegalArgumentException("redisManager cannot be null.");
        }
        this.redisManager = redisManager;
        if (prefix != null && !"".equals(prefix)) {
            this.keyPrefix = prefix;
        }
        if (expire != -1) {
            this.expire = expire;
        }
        if (principalIdFieldName != null && !"".equals(principalIdFieldName)) {
            this.principalIdFieldName = principalIdFieldName;
        }
    }

    @Override
    public V get(K key) throws CacheException {
        logger.debug("get key [{}]",key);

        if (key == null) {
            return null;
        }

        try {
            String redisCacheKey = getRedisCacheKey(key);
            Object rawValue = redisManager.get(redisCacheKey);
            if (rawValue == null) {
                return null;
            }
            V value = (V) rawValue;
            return value;
        } catch (Exception e) {
            throw new CacheException(e);
        }
    }

    @Override
    public V put(K key, V value) throws CacheException {
        logger.debug("put key [{}]",key);
        if (key == null) {
            logger.warn("Saving a null key is meaningless, return value directly without call Redis.");
            return value;
        }
        try {
            String redisCacheKey = getRedisCacheKey(key);
            redisManager.set(redisCacheKey, value != null ? value : null, expire);
            return value;
        } catch (Exception e) {
            throw new CacheException(e);
        }
    }

    @Override
    public V remove(K key) throws CacheException {
        logger.debug("remove key [{}]",key);
        if (key == null) {
            return null;
        }
        try {
            String redisCacheKey = getRedisCacheKey(key);
            Object rawValue = redisManager.get(redisCacheKey);
            V previous = (V) rawValue;
            redisManager.del(redisCacheKey);
            return previous;
        } catch (Exception e) {
            throw new CacheException(e);
        }
    }

    private String getRedisCacheKey(K key) {
        if (key == null) {
            return null;
        }
        return this.keyPrefix + getStringRedisKey(key);
    }

    private String getStringRedisKey(K key) {
        String redisKey;
        if (key instanceof PrincipalCollection) {
            redisKey = getRedisKeyFromPrincipalIdField((PrincipalCollection) key);
        } else {
            redisKey = key.toString();
        }
        return redisKey;
    }

    private String getRedisKeyFromPrincipalIdField(PrincipalCollection key) {
        String redisKey;
        Object principalObject = key.getPrimaryPrincipal();
        Method pincipalIdGetter = null;
        Method[] methods = principalObject.getClass().getDeclaredMethods();
        for (Method m:methods) {
            if (RedisCacheManager.DEFAULT_PRINCIPAL_ID_FIELD_NAME.equals(this.principalIdFieldName)
                    && ("getAuthCacheKey".equals(m.getName()) || "getId".equals(m.getName()))) {
                pincipalIdGetter = m;
                break;
            }
            if (m.getName().equals("get" + this.principalIdFieldName.substring(0, 1).toUpperCase() + this.principalIdFieldName.substri
                pincipalIdGetter = m;
                break;
            }
        }
        if (pincipalIdGetter == null) {
            throw new PrincipalInstanceException(principalObject.getClass(), this.principalIdFieldName);
        }

        try {
            Object idObj = pincipalIdGetter.invoke(principalObject);
            if (idObj == null) {
                throw new PrincipalIdNullException(principalObject.getClass(), this.principalIdFieldName);
            }
            redisKey = idObj.toString();
        } catch (IllegalAccessException e) {
            throw new PrincipalInstanceException(principalObject.getClass(), this.principalIdFieldName, e);
        } catch (InvocationTargetException e) {
            throw new PrincipalInstanceException(principalObject.getClass(), this.principalIdFieldName, e);
        }

        return redisKey;
    }


    @Override
    public void clear() throws CacheException {
        logger.debug("clear cache");
        Set<String> keys = null;
        try {
            keys = redisManager.scan(this.keyPrefix + "*");
        } catch (Exception e) {
            logger.error("get keys error", e);
        }
        if (keys == null || keys.size() == 0) {
            return;
        }
        for (String key: keys) {
            redisManager.del(key);
        }
    }

    @Override
    public int size() {
        Long longSize = 0L;
        try {
            longSize = new Long(redisManager.scanSize(this.keyPrefix + "*"));
        } catch (Exception e) {
            logger.error("get keys error", e);
        }
        return longSize.intValue();
    }

    @SuppressWarnings("unchecked")
    @Override
    public Set<K> keys() {
        Set<String> keys = null;
        try {
            keys = redisManager.scan(this.keyPrefix + "*");
        } catch (Exception e) {
            logger.error("get keys error", e);
            return Collections.emptySet();
        }

        if (CollectionUtils.isEmpty(keys)) {
            return Collections.emptySet();
        }

        Set<K> convertedKeys = new HashSet<K>();
        for (String key:keys) {
            try {
                convertedKeys.add((K) key);
            } catch (Exception e) {
                logger.error("deserialize keys error", e);
            }
        }
        return convertedKeys;
    }

    @Override
    public Collection<V> values() {
        Set<String> keys = null;
        try {
            keys = redisManager.scan(this.keyPrefix + "*");
        } catch (Exception e) {
            logger.error("get values error", e);
            return Collections.emptySet();
        }

        if (CollectionUtils.isEmpty(keys)) {
            return Collections.emptySet();
        }

        List<V> values = new ArrayList<V>(keys.size());
        for (String key : keys) {
            V value = null;
            try {
                value = (V) redisManager.get(key);
            } catch (Exception e) {
                logger.error("deserialize values= error", e);
            }
            if (value != null) {
                values.add(value);
            }
        }
        return Collections.unmodifiableList(values);
    }

    public String getKeyPrefix() {
        return keyPrefix;
    }

    public void setKeyPrefix(String keyPrefix) {
        this.keyPrefix = keyPrefix;
    }

    public String getPrincipalIdFieldName() {
        return principalIdFieldName;
    }

    public void setPrincipalIdFieldName(String principalIdFieldName) {
        this.principalIdFieldName = principalIdFieldName;
    }
}
```

`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

usernane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。


db0 (5/0)
shiro (5)
cache (4)
authenticationCache (1)
shiro:cache:authenticationCache:admin
authorizationCache (1)
shiro:cache:authorizationCache:admin
kickout (1)
shiro:cache:kickout:admin
retrylimit (1)
shiro:cache:retrylimit:admin
session (1)
shiro:session:cbce978b-0eba-4e85-b90a-2e1feb

读取用户权限信息时,还用到两个异常类，如下:

PrincipalInstanceException.java

```java
package com.springboot.test.shiro.global.exceptions;

/**
 * @author: wangsaichao
 * @date: 2018/6/21
 * @description:
 */
public class PrincipalInstanceException extends RuntimeException  {

    private static final String MESSAGE = "We need a field to identify this Cache Object in Redis. "
            + "So you need to defined an id field which you can get unique id to identify this principal. "
            + "For example, if you use UserInfo as Principal class, the id field maybe userId, userName, email, etc. "
            + "For example, getUserId(), getUserName(), getEmail(), etc.\n"
            + "Default value is authCacheKey or id, that means your principal object has a method called \"getAuthCacheKey()\" or \"ge

    public PrincipalInstanceException(Class clazz, String idMethodName) {
        super(clazz + " must has getter for field: " +  idMethodName + "\n" + MESSAGE);
    }

    public PrincipalInstanceException(Class clazz, String idMethodName, Exception e) {
        super(clazz + " must has getter for field: " +  idMethodName + "\n" + MESSAGE, e);
    }
}
```

PrincipalIdNullException.java

```java
package com.springboot.test.shiro.global.exceptions;

/**
 * @author: wangsaichao
 * @date: 2018/6/21
 * @description:
 */
public class PrincipalIdNullException extends RuntimeException  {

    private static final String MESSAGE = "Principal Id shouldn't be null!";

    public PrincipalIdNullException(Class clazz, String idMethodName) {
        super(clazz + " id field: " +  idMethodName + ", value is null\n" + MESSAGE);
    }
}
```

RedisCacheManager.java

```java
package com.springboot.test.shiro.config.shiro;

import org.apache.shiro.cache.Cache;
import org.apache.shiro.cache.CacheException;
import org.apache.shiro.cache.CacheManager;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;

/**
 * @author: wangsaichao
 * @date: 2018/6/22
 * @description: 参考 shiro-redis 开源项目 Git地址 https://github.com/alexxiyang/shiro-redis
 */
public class RedisCacheManager implements CacheManager {

    private final Logger logger = LoggerFactory.getLogger(RedisCacheManager.class);

    /**
     * fast lookup by name map
     */
    private final ConcurrentMap<String, Cache> caches = new ConcurrentHashMap<String, Cache>();

    private RedisManager redisManager;

    /**
     * expire time in seconds
     */
    private static final int DEFAULT_EXPIRE = 1800;
    private int expire = DEFAULT_EXPIRE;

    /**
     * The Redis key prefix for caches
     */
    public static final String DEFAULT_CACHE_KEY_PREFIX = "shiro:cache:";
    private String keyPrefix = DEFAULT_CACHE_KEY_PREFIX;

    public static final String DEFAULT_PRINCIPAL_ID_FIELD_NAME = "authCacheKey or id";
    private String principalIdFieldName = DEFAULT_PRINCIPAL_ID_FIELD_NAME;

    @Override
    public <K, V> Cache<K, V> getCache(String name) throws CacheException {
        logger.debug("get cache, name={}",name);

        Cache cache = caches.get(name);

        if (cache == null) {
            cache = new RedisCache<K, V>(redisManager,keyPrefix + name + ":", expire, principalIdFieldName);
            caches.put(name, cache);
        }
        return cache;
    }

    public RedisManager getRedisManager() {
        return redisManager;
    }

    public void setRedisManager(RedisManager redisManager) {
        this.redisManager = redisManager;
    }

    public String getKeyPrefix() {
        return keyPrefix;
    }

    public void setKeyPrefix(String keyPrefix) {
        this.keyPrefix = keyPrefix;
    }

    public int getExpire() {
        return expire;
    }

    public void setExpire(int expire) {
        this.expire = expire;
    }

    public String getPrincipalIdFieldName() {
        return principalIdFieldName;
    }

    public void setPrincipalIdFieldName(String principalIdFieldName) {
        this.principalIdFieldName = principalIdFieldName;
    }
}
```

RedisSessionDAO.java

```java
package com.springboot.test.shiro.config.shiro;

import org.apache.shiro.session.Session;
import org.apache.shiro.session.UnknownSessionException;
import org.apache.shiro.session.mgt.ValidatingSession;
import org.apache.shiro.session.mgt.eis.AbstractSessionDAO;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.Serializable;
import java.util.*;

/**
 * @author: wangsaichao
 * @date: 2018/6/22
 * @description: 参考 shiro-redis 开源项目 Git地址 https://github.com/alexxiyang/shiro-redis
 */
public class RedisSessionDAO extends AbstractSessionDAO {

    private static Logger logger = LoggerFactory.getLogger(RedisSessionDAO.class);

    private static final String DEFAULT_SESSION_KEY_PREFIX = "shiro:session:";
    private String keyPrefix = DEFAULT_SESSION_KEY_PREFIX;

    private static final long DEFAULT_SESSION_IN_MEMORY_TIMEOUT = 1000L;
    /**
     * doReadSession be called about 10 times when login.
     * Save Session in ThreadLocal to resolve this problem. sessionInMemoryTimeout is expiration of Session in ThreadLocal.
     * The default value is 1000 milliseconds (1s).
     * Most of time, you don't need to change it.
     */
    private long sessionInMemoryTimeout = DEFAULT_SESSION_IN_MEMORY_TIMEOUT;

    /**
     * expire time in seconds
     */
    private static final int DEFAULT_EXPIRE = -2;
    private static final int NO_EXPIRE = -1;

    /**
     * Please make sure expire is longer than sesion.getTimeout()
     */
    private int expire = DEFAULT_EXPIRE;

    private static final int MILLISECONDS_IN_A_SECOND = 1000;

    private RedisManager redisManager;
    private static ThreadLocal sessionsInThread = new ThreadLocal();

    @Override
    public void update(Session session) throws UnknownSessionException {
        //如果会话过期/停止 没必要再更新了
        try {
            if (session instanceof ValidatingSession && !((ValidatingSession) session).isValid()) {
                return;
            }

            if (session instanceof ShiroSession) {
                // 如果没有主要字段(除lastAccessTime以外其他字段)发生改变
                ShiroSession ss = (ShiroSession) session;
                if (!ss.isChanged()) {
                    return;
                }
                //如果没有返回 证明有调用 setAttribute往redis 放的时候永远设置为false
                ss.setChanged(false);
            }

            this.saveSession(session);
        } catch (Exception e) {
            logger.warn("update Session is failed", e);
        }
    }

    /**
     * save session
     * @param session
     * @throws UnknownSessionException
     */
    private void saveSession(Session session) throws UnknownSessionException {
        if (session == null || session.getId() == null) {
            logger.error("session or session id is null");
            throw new UnknownSessionException("session or session id is null");
        }
        String key = getRedisSessionKey(session.getId());
        if (expire == DEFAULT_EXPIRE) {
            this.redisManager.set(key, session, (int) (session.getTimeout() / MILLISECONDS_IN_A_SECOND));
            return;
        }
        if (expire != NO_EXPIRE && expire * MILLISECONDS_IN_A_SECOND < session.getTimeout()) {
            logger.warn("Redis session expire time: "
                    + (expire * MILLISECONDS_IN_A_SECOND)
                    + " is less than Session timeout: "
                    + session.getTimeout()
                    + " . It may cause some problems.");
        }
        this.redisManager.set(key, session, expire);
    }

    @Override
    public void delete(Session session) {
        if (session == null || session.getId() == null) {
            logger.error("session or session id is null");
            return;
        }
        try {
            redisManager.del(getRedisSessionKey(session.getId()));
        } catch (Exception e) {
            logger.error("delete session error. session id= {}",session.getId());
        }
    }

    @Override
    public Collection<Session> getActiveSessions() {
        Set<Session> sessions = new HashSet<Session>();
        try {
            Set<String> keys = redisManager.scan(this.keyPrefix + "*");
            if (keys != null && keys.size() > 0) {
                for (String key:keys) {
                    Session s = (Session) redisManager.get(key);
                    sessions.add(s);
                }
            }
        } catch (Exception e) {
            logger.error("get active sessions error.");
        }
        return sessions;
    }

    public Long getActiveSessionsSize() {
        Long size = 0L;
        try {
            size = redisManager.scanSize(this.keyPrefix + "*");
        } catch (Exception e) {
            logger.error("get active sessions error.");
        }
        return size;
    }

    @Override
    protected Serializable doCreate(Session session) {
        if (session == null) {
            logger.error("session is null");
            throw new UnknownSessionException("session is null");
        }
        Serializable sessionId = this.generateSessionId(session);
        this.assignSessionId(session, sessionId);
        this.saveSession(session);
        return sessionId;
    }
```

```java

    @Override
    protected Session doReadSession(Serializable sessionId) {
        if (sessionId == null) {
            logger.warn("session id is null");
            return null;
        }
        Session s = getSessionFromThreadLocal(sessionId);

        if (s != null) {
            return s;
        }

        logger.debug("read session from redis");
        try {
            s = (Session) redisManager.get(getRedisSessionKey(sessionId));
            setSessionToThreadLocal(sessionId, s);
        } catch (Exception e) {
            logger.error("read session error. settionId= {}",sessionId);
        }
        return s;
    }

    private void setSessionToThreadLocal(Serializable sessionId, Session s) {
        Map<Serializable, SessionInMemory> sessionMap = (Map<Serializable, SessionInMemory>) sessionsInThread.get();
        if (sessionMap == null) {
            sessionMap = new HashMap<Serializable, SessionInMemory>();
            sessionsInThread.set(sessionMap);
        }
        SessionInMemory sessionInMemory = new SessionInMemory();
        sessionInMemory.setCreateTime(new Date());
        sessionInMemory.setSession(s);
        sessionMap.put(sessionId, sessionInMemory);
    }

    private Session getSessionFromThreadLocal(Serializable sessionId) {
        Session s = null;

        if (sessionsInThread.get() == null) {
            return null;
        }

        Map<Serializable, SessionInMemory> sessionMap = (Map<Serializable, SessionInMemory>) sessionsInThread.get();
        SessionInMemory sessionInMemory = sessionMap.get(sessionId);
        if (sessionInMemory == null) {
            return null;
        }
        Date now = new Date();
        long duration = now.getTime() - sessionInMemory.getCreateTime().getTime();
        if (duration < sessionInMemoryTimeout) {
            s = sessionInMemory.getSession();
            logger.debug("read session from memory");
        } else {
            sessionMap.remove(sessionId);
        }

        return s;
    }

    private String getRedisSessionKey(Serializable sessionId) {
        return this.keyPrefix + sessionId;
    }

    public RedisManager getRedisManager() {
        return redisManager;
    }

    public void setRedisManager(RedisManager redisManager) {
        this.redisManager = redisManager;
    }

    public String getKeyPrefix() {
        return keyPrefix;
    }

    public void setKeyPrefix(String keyPrefix) {
        this.keyPrefix = keyPrefix;
    }

    public long getSessionInMemoryTimeout() {
        return sessionInMemoryTimeout;
    }

    public void setSessionInMemoryTimeout(long sessionInMemoryTimeout) {
        this.sessionInMemoryTimeout = sessionInMemoryTimeout;
    }

    public int getExpire() {
        return expire;
    }

    public void setExpire(int expire) {
        this.expire = expire;
    }
}
```

3.Shiro配置

ShiroConfig.java

```java
package com.springboot.test.shiro.config;

import at.pollux.thymeleaf.shiro.dialect.ShiroDialect;
import com.springboot.test.shiro.config.shiro.*;
import org.apache.shiro.codec.Base64;
import org.apache.shiro.session.SessionListener;
import org.apache.shiro.session.mgt.SessionManager;
import org.apache.shiro.session.mgt.eis.JavaUuidSessionIdGenerator;
import org.apache.shiro.session.mgt.eis.SessionDAO;
import org.apache.shiro.session.mgt.eis.SessionIdGenerator;
import org.apache.shiro.spring.LifecycleBeanPostProcessor;
import org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor;
import org.apache.shiro.spring.web.ShiroFilterFactoryBean;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.web.filter.authc.FormAuthenticationFilter;
import org.apache.shiro.web.mgt.CookieRememberMeManager;
import org.apache.shiro.web.mgt.DefaultWebSecurityManager;
import org.apache.shiro.web.servlet.SimpleCookie;
import org.apache.shiro.web.session.mgt.DefaultWebSessionManager;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.config.MethodInvokingFactoryBean;
import org.springframework.boot.context.embedded.ConfigurableEmbeddedServletContainer;
import org.springframework.boot.context.embedded.EmbeddedServletContainerCustomizer;
import org.springframework.boot.web.servlet.ErrorPage;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpStatus;
import org.springframework.web.servlet.handler.SimpleMappingExceptionResolver;

import javax.servlet.Filter;
import java.util.ArrayList;
import java.util.Collection;
import java.util.LinkedHashMap;
import java.util.Properties;

/**
 * @author: wangsaichao
 * @date: 2018/5/10
 * @description: Shiro配置
 */
@Configuration
public class ShiroConfig {


    /**
     * ShiroFilterFactoryBean 处理拦截资源文件问题。
     * 注意：初始化ShiroFilterFactoryBean的时候需要注入：SecurityManager
     * Web应用中,Shiro可控制的Web请求必须经过Shiro主过滤器的拦截
     * @param securityManager
     * @return
     */
    @Bean(name = "shirFilter")
    public ShiroFilterFactoryBean shiroFilter(@Qualifier("securityManager") SecurityManager securityManager) {

        ShiroFilterFactoryBean shiroFilterFactoryBean = new ShiroFilterFactoryBean();

        //必须设置 SecurityManager,Shiro的核心安全接口
        shiroFilterFactoryBean.setSecurityManager(securityManager);
        //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
        shiroFilterFactoryBean.setLoginUrl("/");
        //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
        shiroFilterFactoryBean.setSuccessUrl("/index");
        //未授权界面,该配置无效，并不会进行页面跳转
        shiroFilterFactoryBean.setUnauthorizedUrl("/unauthorized");

        //自定义拦截器限制并发人数,参考博客:
        LinkedHashMap<String, Filter> filtersMap = new LinkedHashMap<>();
        //限制同一帐号同时在线的个数
        filtersMap.put("kickout", kickoutSessionControlFilter());
        //统计登录人数
        shiroFilterFactoryBean.setFilters(filtersMap);

        // 配置访问权限 必须是LinkedHashMap，因为它必须保证有序
        // 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
        LinkedHashMap<String, String> filterChainDefinitionMap = new LinkedHashMap<>();
        //配置不登录可以访问的资源，anon 表示资源都可以匿名访问
        //配置记住我或认证通过可以访问的地址
        filterChainDefinitionMap.put("/login", "anon");
        filterChainDefinitionMap.put("/", "anon");
        filterChainDefinitionMap.put("/css/**", "anon");
        filterChainDefinitionMap.put("/js/**", "anon");
        filterChainDefinitionMap.put("/img/**", "anon");
        filterChainDefinitionMap.put("/druid/**", "anon");
        //解锁用户专用 测试用的
        filterChainDefinitionMap.put("/unlockAccount","anon");
        filterChainDefinitionMap.put("/Captcha.jpg","anon");
        //logout是shiro提供的过滤器
        filterChainDefinitionMap.put("/logout", "logout");
        //此时访问/user/delete需要delete权限,在自定义Realm中为用户授权。
        //filterChainDefinitionMap.put("/user/delete", "perms[\"user:delete\"]");

        //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
        //如果开启限制同一账号登录,改为 .put("/**", "kickout,user");
        filterChainDefinitionMap.put("/**", "kickout,user");

        shiroFilterFactoryBean.setFilterChainDefinitionMap(filterChainDefinitionMap);

        return shiroFilterFactoryBean;
    }

    /**
     * 配置核心安全事务管理器
     * @return
     */
    @Bean(name="securityManager")
    public SecurityManager securityManager() {
        DefaultWebSecurityManager securityManager =  new DefaultWebSecurityManager();
        //设置自定义realm.
        securityManager.setRealm(shiroRealm());
        //配置记住我
        securityManager.setRememberMeManager(rememberMeManager());
        //配置redis缓存
        securityManager.setCacheManager(cacheManager());
        //配置自定义session管理，使用redis
        securityManager.setSessionManager(sessionManager());
        return securityManager;
    }

    /**
     * 配置Shiro生命周期处理器
     * @return
     */
    @Bean(name = "lifecycleBeanPostProcessor")
    public LifecycleBeanPostProcessor lifecycleBeanPostProcessor() {
        return new LifecycleBeanPostProcessor();
    }

    /**
     *  身份认证realm; (这个需要自己写，账号密码校验；权限等)
     * @return
     */
    @Bean
    public ShiroRealm shiroRealm(){
        ShiroRealm shiroRealm = new ShiroRealm();
        shiroRealm.setCachingEnabled(true);
        //启用身份验证缓存，即缓存AuthenticationInfo信息，默认false
```

```java
        shiroRealm.setAuthenticationCachingEnabled(true);
        //缓存AuthenticationInfo信息的缓存名称 在ehcache-shiro.xml中有对应缓存的配置
        shiroRealm.setAuthenticationCacheName("authenticationCache");
        //启用授权缓存，即缓存AuthorizationInfo信息，默认false
        shiroRealm.setAuthorizationCachingEnabled(true);
        //缓存AuthorizationInfo信息的缓存名称  在ehcache-shiro.xml中有对应缓存的配置
        shiroRealm.setAuthorizationCacheName("authorizationCache");
        //配置自定义密码比较器
        shiroRealm.setCredentialsMatcher(retryLimitHashedCredentialsMatcher());
        return shiroRealm;
    }

    /**
     * 必须（thymeleaf页面使用shiro标签控制按钮是否显示）
     * 未引入thymeleaf包，Caused by: java.lang.ClassNotFoundException: org.thymeleaf.dialect.AbstractProcessorDialect
     * @return
     */
    @Bean
    public ShiroDialect shiroDialect() {
        return new ShiroDialect();
    }

    /**
     * 开启shiro 注解模式
     * 可以在controller中的方法前加上注解
     * 如 @RequiresPermissions("userInfo:add")
     * @param securityManager
     * @return
     */
    @Bean
    public AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor(@Qualifier("securityManager") SecurityManager secur
        AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor = new AuthorizationAttributeSourceAdvisor();
        authorizationAttributeSourceAdvisor.setSecurityManager(securityManager);
        return authorizationAttributeSourceAdvisor;
    }

    /**
     * 解决： 无权限页面不跳转 shiroFilterFactoryBean.setUnauthorizedUrl("/unauthorized") 无效
     * shiro的源代码ShiroFilterFactoryBean.Java定义的filter必须满足filter instanceof AuthorizationFilter,
     * 只有perms，roles，ssl，rest，port才是属于AuthorizationFilter，而anon，authcBasic，auchc，user是AuthenticationFilter,
     * 所以unauthorizedUrl设置后页面不跳转 Shiro注解模式下，登录失败与没有权限都是通过抛出异常。
     * 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
     * @return
     */
    @Bean
    public SimpleMappingExceptionResolver simpleMappingExceptionResolver() {
        SimpleMappingExceptionResolver simpleMappingExceptionResolver=new SimpleMappingExceptionResolver();
        Properties properties=new Properties();
        //这里的 /unauthorized 是页面，不是访问的路径
        properties.setProperty("org.apache.shiro.authz.UnauthorizedException","/unauthorized");
        properties.setProperty("org.apache.shiro.authz.UnauthenticatedException","/unauthorized");
        simpleMappingExceptionResolver.setExceptionMappings(properties);
        return simpleMappingExceptionResolver;
    }

    /**
     * 解决spring-boot Whitelabel Error Page
     * @return
     */
    @Bean
    public EmbeddedServletContainerCustomizer containerCustomizer() {

        return new EmbeddedServletContainerCustomizer() {
            @Override
            public void customize(ConfigurableEmbeddedServletContainer container) {

                ErrorPage error401Page = new ErrorPage(HttpStatus.UNAUTHORIZED, "/unauthorized.html");
                ErrorPage error404Page = new ErrorPage(HttpStatus.NOT_FOUND, "/404.html");
                ErrorPage error500Page = new ErrorPage(HttpStatus.INTERNAL_SERVER_ERROR, "/500.html");

                container.addErrorPages(error401Page, error404Page, error500Page);
            }
        };
    }

    /**
     * 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
     * @return
     */
    @Bean
    public SimpleCookie rememberMeCookie(){
        //这个参数是cookie的名称，对应前端的checkbox的name = rememberMe
        SimpleCookie simpleCookie = new SimpleCookie("rememberMe");
        //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:
        //setcookie()的第七个参数
        //设为true后，只能通过http访问，javascript无法访问
        //防止xss读取cookie
        simpleCookie.setHttpOnly(true);
        simpleCookie.setPath("/");
        //<!-- 记住我cookie生效时间30天 ,单位秒;-->
        simpleCookie.setMaxAge(2592000);
        return simpleCookie;
    }

    /**
     * cookie管理对象;记住我功能,rememberMe管理器
     * @return
     */
    @Bean
    public CookieRememberMeManager rememberMeManager(){
        CookieRememberMeManager cookieRememberMeManager = new CookieRememberMeManager();
        cookieRememberMeManager.setCookie(rememberMeCookie());
        //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)
        cookieRememberMeManager.setCipherKey(Base64.decode("4AvVhmFLUs0KTA3Kprsdag=="));
        return cookieRememberMeManager;
    }

    /**
     * FormAuthenticationFilter 过滤器 过滤记住我
     * @return
     */
    @Bean
    public FormAuthenticationFilter formAuthenticationFilter(){
        FormAuthenticationFilter formAuthenticationFilter = new FormAuthenticationFilter();
        //对应前端的checkbox的name = rememberMe
        formAuthenticationFilter.setRememberMeParam("rememberMe");
        return formAuthenticationFilter;
    }

    /**
     * shiro缓存管理器;
     * 需要添加到securityManager中
     * @return
     */
    @Bean
    public RedisCacheManager cacheManager(){
        RedisCacheManager redisCacheManager = new RedisCacheManager();
        redisCacheManager.setRedisManager(redisManager());
        //redis中针对不同用户缓存
        redisCacheManager.setPrincipalIdFieldName("username");
        //用户权限信息缓存时间
        redisCacheManager.setExpire(200000);
        return redisCacheManager;
    }

    /**
     * 让某个实例的某个方法的返回值注入为Bean的实例
     * Spring静态注入
     * @return
     */
    @Bean
    public MethodInvokingFactoryBean getMethodInvokingFactoryBean(){
        MethodInvokingFactoryBean factoryBean = new MethodInvokingFactoryBean();
        factoryBean.setStaticMethod("org.apache.shiro.SecurityUtils.setSecurityManager");
        factoryBean.setArguments(new Object[]{securityManager()});
        return factoryBean;
    }

    /**
     * 配置session监听
     * @return
     */
    @Bean("sessionListener")
    public ShiroSessionListener sessionListener(){
        ShiroSessionListener sessionListener = new ShiroSessionListener();
        return sessionListener;
    }

    /**
     * 配置会话ID生成器
     * @return
     */
    @Bean
    public SessionIdGenerator sessionIdGenerator() {
        return new JavaUuidSessionIdGenerator();
    }

    @Bean
    public RedisManager redisManager(){
        RedisManager redisManager = new RedisManager();
        return redisManager;
    }

    @Bean("sessionFactory")
    public ShiroSessionFactory sessionFactory(){
        ShiroSessionFactory sessionFactory = new ShiroSessionFactory();
        return sessionFactory;
    }

    /**
     * SessionDAO的作用是为Session提供CRUD并进行持久化的一个shiro组件
     * MemorySessionDAO 直接在内存中进行会话维护
     * 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。
     * @return
     */
    @Bean
    public SessionDAO sessionDAO() {
        RedisSessionDAO redisSessionDAO = new RedisSessionDAO();
        redisSessionDAO.setRedisManager(redisManager());
        //session在redis中的保存时间,最好大于session会话超时时间
        redisSessionDAO.setExpire(12000);
        return redisSessionDAO;
    }

    /**
     * 配置保存sessionId的cookie
     * 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
     * 默认为: JSESSIONID 问题: 与SERVLET容器名冲突,重新定义为sid
     * @return
     */
    @Bean("sessionIdCookie")
    public SimpleCookie sessionIdCookie(){
        //这个参数是cookie的名称
        SimpleCookie simpleCookie = new SimpleCookie("sid");
        //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：

        //setcookie()的第七个参数
        //设为true后，只能通过http访问，javascript无法访问
        //防止xss读取cookie
        simpleCookie.setHttpOnly(true);
        simpleCookie.setPath("/");
        //maxAge=-1表示浏览器关闭时失效此Cookie
        simpleCookie.setMaxAge(-1);
        return simpleCookie;
    }

    /**
     * 配置会话管理器，设定会话超时及保存
     * @return
     */
    @Bean("sessionManager")
    public SessionManager sessionManager() {
        ShiroSessionManager sessionManager = new ShiroSessionManager();
        Collection<SessionListener> listeners = new ArrayList<SessionListener>();
        //配置监听
        listeners.add(sessionListener());
        sessionManager.setSessionListeners(listeners);
        sessionManager.setSessionIdCookie(sessionIdCookie());
        sessionManager.setSessionDAO(sessionDAO());
        sessionManager.setCacheManager(cacheManager());
        sessionManager.setSessionFactory(sessionFactory());

        //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
        sessionManager.setGlobalSessionTimeout(1800000);
        //是否开启删除无效的session对象  默认为true
        sessionManager.setDeleteInvalidSessions(true);
        //是否开启定时调度器进行检测过期session 默认为true
        sessionManager.setSessionValidationSchedulerEnabled(true);
        //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
        //设置该属性 就不需要设置 ExecutorServiceSessionValidationScheduler 底层也是默认自动调用ExecutorServiceSessionValidationScheduler
        //暂时设置为 5秒 用来测试
        sessionManager.setSessionValidationInterval(3600000);
        //取消url 后面的 JSESSIONID
        sessionManager.setSessionIdUrlRewritingEnabled(false);
        return sessionManager;

    }

    /**
     * 并发登录控制
     * @return
     */
    @Bean
    public KickoutSessionControlFilter kickoutSessionControlFilter(){
        KickoutSessionControlFilter kickoutSessionControlFilter = new KickoutSessionControlFilter();
        //用于根据会话ID，获取会话进行踢出操作的；
        kickoutSessionControlFilter.setSessionManager(sessionManager());
        //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；
        kickoutSessionControlFilter.setRedisManager(redisManager());
        //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；
        kickoutSessionControlFilter.setKickoutAfter(false);
        //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；
        kickoutSessionControlFilter.setMaxSession(1);
        //被踢出后重定向到的地址；
        kickoutSessionControlFilter.setKickoutUrl("/login?kickout=1");
        return kickoutSessionControlFilter;
    }

    /**
     * 配置密码比较器
     * @return
     */
    @Bean("credentialsMatcher")
    public RetryLimitHashedCredentialsMatcher retryLimitHashedCredentialsMatcher(){
        RetryLimitHashedCredentialsMatcher retryLimitHashedCredentialsMatcher = new RetryLimitHashedCredentialsMatcher();
        retryLimitHashedCredentialsMatcher.setRedisManager(redisManager());

        //如果密码加密,可以打开下面配置
        //加密算法的名称
        //retryLimitHashedCredentialsMatcher.setHashAlgorithmName("MD5");
        //配置加密的次数
        //retryLimitHashedCredentialsMatcher.setHashIterations(1024);
        //是否存储为16进制
        //retryLimitHashedCredentialsMatcher.setStoredCredentialsHexEncoded(true);

        return retryLimitHashedCredentialsMatcher;
    }

}
```

ShiroRealm.java

```java
package com.springboot.test.shiro.config.shiro;

import com.springboot.test.shiro.modules.user.dao.PermissionMapper;
import com.springboot.test.shiro.modules.user.dao.RoleMapper;
import com.springboot.test.shiro.modules.user.dao.entity.Permission;
import com.springboot.test.shiro.modules.user.dao.entity.Role;
import com.springboot.test.shiro.modules.user.dao.UserMapper;
import com.springboot.test.shiro.modules.user.dao.entity.User;
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.authc.*;
import org.apache.shiro.authz.AuthorizationInfo;
import org.apache.shiro.authz.SimpleAuthorizationInfo;
import org.apache.shiro.realm.AuthorizingRealm;
import org.apache.shiro.subject.PrincipalCollection;
import org.springframework.beans.factory.annotation.Autowired;

import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;

/**
 * @author: wangsaichao
 * @date: 2018/5/10
 * @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
 * 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.
 */
public class ShiroRealm extends AuthorizingRealm {

    @Autowired
    private UserMapper userMapper;

    @Autowired
    private RoleMapper roleMapper;

    @Autowired
    private PermissionMapper permissionMapper;

    /**
     * 验证用户身份
     * @param authenticationToken
     * @return
     * @throws AuthenticationException
     */
    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) throws AuthenticationException {

        //获取用户名密码 第一种方式
        //String username = (String) authenticationToken.getPrincipal();
        //String password = new String((char[]) authenticationToken.getCredentials());

        //获取用户名 密码 第二种方式
        UsernamePasswordToken usernamePasswordToken = (UsernamePasswordToken) authenticationToken;
        String username = usernamePasswordToken.getUsername();
        String password = new String(usernamePasswordToken.getPassword());

        //从数据库查询用户信息
        User user = this.userMapper.findByUserName(username);

        //可以在这里直接对用户名校验,或者调用 CredentialsMatcher 校验
        if (user == null) {
            throw new UnknownAccountException("用户名或密码错误！");
        }
        //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
        //if (!password.equals(user.getPassword())) {
        //    throw new IncorrectCredentialsException("用户名或密码错误！");
        //}
        if ("1".equals(user.getState())) {
            throw new LockedAccountException("账号已被锁定,请联系管理员！");
        }

        //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
        //配置自定义权限登录器 参考博客:

        SimpleAuthenticationInfo info = new SimpleAuthenticationInfo(user, user.getPassword(), getName());
        return info;
    }

    /**
     * 授权用户权限
     * 授权的方法是在碰到<shiro:hasPermission name=''></shiro:hasPermission>标签的时候调用的
     * 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
     * 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.)
     *
     * shiro的权限授权是通过继承AuthorizingRealm抽象类，重载doGetAuthorizationInfo();
     * 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
     * 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。
     *
     * 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。
     * authorizationInfo.addRole(role.getRole()); authorizationInfo.addStringPermission(p.getPermission());
     *
     * 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
     * authorizationInfo.setRoles(roles); authorizationInfo.setStringPermissions(stringPermissions);
     *
     * 就是说如果在shiro配置文件中添加了filterChainDefinitionMap.put("/add", "perms[权限添加]");
     * 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
     *
     * 如果在shiro配置文件中添加了filterChainDefinitionMap.put("/add", "roles[100002]，perms[权限添加]");
     * 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
     * @param principalCollection
     * @return
     */
    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {

        System.out.println("查询权限方法调用了！！！");

        //获取用户
        User user = (User) SecurityUtils.getSubject().getPrincipal();

        //获取用户角色
        Set<Role> roles =this.roleMapper.findRolesByUserId(user.getUid());
        //添加角色
        SimpleAuthorizationInfo authorizationInfo =  new SimpleAuthorizationInfo();
        for (Role role : roles) {
            authorizationInfo.addRole(role.getRole());
        }

        //获取用户权限
        Set<Permission> permissions = this.permissionMapper.findPermissionsByRoleId(roles);
        //添加权限
        for (Permission permission:permissions) {
            authorizationInfo.addStringPermission(permission.getPermission());
        }

        return authorizationInfo;
    }

    /**
     * 重写方法,清除当前用户的的 授权缓存
     * @param principals
     */
    @Override
    public void clearCachedAuthorizationInfo(PrincipalCollection principals) {
        super.clearCachedAuthorizationInfo(principals);
    }

    /**
     * 重写方法，清除当前用户的 认证缓存
     * @param principals
     */
    @Override
    public void clearCachedAuthenticationInfo(PrincipalCollection principals) {
        super.clearCachedAuthenticationInfo(principals);
    }

    @Override
    public void clearCache(PrincipalCollection principals) {
        super.clearCache(principals);
    }

    /**
     * 自定义方法：清除所有 授权缓存
     */
    public void clearAllCachedAuthorizationInfo() {
        getAuthorizationCache().clear();
    }

    /**
     * 自定义方法：清除所有 认证缓存
     */
    public void clearAllCachedAuthenticationInfo() {
        getAuthenticationCache().clear();
    }

    /**
     * 自定义方法：清除所有的  认证缓存  和 授权缓存
     */
    public void clearAllCache() {
        clearAllCachedAuthenticationInfo();
        clearAllCachedAuthorizationInfo();
    }

}
```

KickoutSessionControlFilter.java(限制并发 登录人数)

```java
package com.springboot.test.shiro.config.shiro;

import java.io.Serializable;
import java.util.Deque;
import java.util.LinkedList;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;

import com.springboot.test.shiro.modules.user.dao.entity.User;
import org.apache.shiro.session.Session;
import org.apache.shiro.session.mgt.DefaultSessionKey;
import org.apache.shiro.session.mgt.SessionManager;
import org.apache.shiro.subject.Subject;
import org.apache.shiro.web.filter.AccessControlFilter;
import org.apache.shiro.web.util.WebUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.servlet.resource.ResourceUrlProvider;

/**
 * @author: WangSaiChao
 * @date: 2018/5/23
 * @description: shiro 自定义filter 实现 并发登录控制
 */
public class KickoutSessionControlFilter  extends AccessControlFilter{

    @Autowired
    private ResourceUrlProvider resourceUrlProvider;

    /** 踢出后到的地址 */
    private String kickoutUrl;

    /**  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 */
    private boolean kickoutAfter = false;

    /**  同一个帐号最大会话数 默认1 */
    private int maxSession = 1;
    private SessionManager sessionManager;

    private RedisManager redisManager;

    public static final String DEFAULT_KICKOUT_CACHE_KEY_PREFIX = "shiro:cache:kickout:";
    private String keyPrefix = DEFAULT_KICKOUT_CACHE_KEY_PREFIX;

    public void setKickoutUrl(String kickoutUrl) {
        this.kickoutUrl = kickoutUrl;
    }

    public void setKickoutAfter(boolean kickoutAfter) {
        this.kickoutAfter = kickoutAfter;
    }

    public void setMaxSession(int maxSession) {
        this.maxSession = maxSession;
    }

    public void setSessionManager(SessionManager sessionManager) {
        this.sessionManager = sessionManager;
    }

    public void setRedisManager(RedisManager redisManager) {
        this.redisManager = redisManager;
    }

    public String getKeyPrefix() {
        return keyPrefix;
    }

    public void setKeyPrefix(String keyPrefix) {
        this.keyPrefix = keyPrefix;
    }

    private String getRedisKickoutKey(String username) {
        return this.keyPrefix + username;
    }

    /**
     * 是否允许访问，返回true表示允许
     */
    @Override
    protected boolean isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) throws Exception {
        return false;
    }
    /**
```

```
     * 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）。
     */
    @Override
    protected boolean onAccessDenied(ServletRequest request, ServletResponse response) throws Exception {
        Subject subject = getSubject(request, response);
        if(!subject.isAuthenticated() && !subject.isRemembered()) {
            //如果没有登录，直接进行之后的流程
            return true;
        }

        //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
        HttpServletRequest httpServletRequest = (HttpServletRequest)request;
        String path = httpServletRequest.getServletPath();
        // 如果是静态文件，则返回true
        if (isStaticFile(path)){
            return true;
        }


        Session session = subject.getSession();
        //这里获取的User是实体 因为我在 自定义ShiroRealm中的doGetAuthenticationInfo方法中
        //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userN
        String username = ((User) subject.getPrincipal()).getUsername();
        Serializable sessionId = session.getId();

        // 初始化用户的队列放到缓存里
        Deque<Serializable> deque = (Deque<Serializable>) redisManager.get(getRedisKickoutKey(username));
        if(deque == null || deque.size()==0) {
            deque = new LinkedList<Serializable>();
        }

        //如果队列里没有此sessionId，且用户没有被踢出；放入队列
        if(!deque.contains(sessionId) && session.getAttribute("kickout") == null) {
            deque.push(sessionId);
        }

        //如果队列里的sessionId数超出最大会话数，开始踢人
        while(deque.size() > maxSession) {
            Serializable kickoutSessionId = null;
            if(kickoutAfter) { //如果踢出后者
                kickoutSessionId=deque.getFirst();
                kickoutSessionId = deque.removeFirst();
            } else { //否则踢出前者
                kickoutSessionId = deque.removeLast();
            }
            try {
                Session kickoutSession = sessionManager.getSession(new DefaultSessionKey(kickoutSessionId));
                if(kickoutSession != null) {
                    //设置会话的kickout属性表示踢出了
                    kickoutSession.setAttribute("kickout", true);
                }
            } catch (Exception e) {//ignore exception
                e.printStackTrace();
            }
        }

        redisManager.set(getRedisKickoutKey(username), deque);

        //如果被踢出了，直接退出，重定向到踢出后的地址
        if (session.getAttribute("kickout") != null) {
            //会话被踢出了
            try {
                subject.logout();
            } catch (Exception e) {
            }
            WebUtils.issueRedirect(request, response, kickoutUrl);
            return false;
        }
        return true;
    }

    private boolean isStaticFile(String path) {
        String staticUri = resourceUrlProvider.getForLookupPath(path);
        return staticUri != null;
    }

}
```

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

```java
package com.springboot.test.shiro.config.shiro;

import java.util.concurrent.atomic.AtomicInteger;

import com.springboot.test.shiro.modules.user.dao.UserMapper;
import com.springboot.test.shiro.modules.user.dao.entity.User;
import org.apache.log4j.Logger;
import org.apache.shiro.authc.AuthenticationInfo;
import org.apache.shiro.authc.AuthenticationToken;
import org.apache.shiro.authc.LockedAccountException;
import org.apache.shiro.authc.credential.SimpleCredentialsMatcher;
import org.apache.shiro.cache.Cache;
import org.apache.shiro.cache.CacheManager;
import org.springframework.beans.factory.annotation.Autowired;


/**
 * @author: WangSaiChao
 * @date: 2018/5/25
 * @description: 登陆次数限制
 */
public class RetryLimitHashedCredentialsMatcher extends SimpleCredentialsMatcher {

    private static final Logger logger = Logger.getLogger(RetryLimitHashedCredentialsMatcher.class);

    public static final String DEFAULT_RETRYLIMIT_CACHE_KEY_PREFIX = "shiro:cache:retrylimit:";
    private String keyPrefix = DEFAULT_RETRYLIMIT_CACHE_KEY_PREFIX;
    @Autowired
    private UserMapper userMapper;
    private RedisManager redisManager;

    public void setRedisManager(RedisManager redisManager) {
        this.redisManager = redisManager;
    }

    private String getRedisKickoutKey(String username) {
        return this.keyPrefix + username;
    }

    @Override
    public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {

        //获取用户名
        String username = (String)token.getPrincipal();
        //获取用户登录次数
        AtomicInteger retryCount = (AtomicInteger)redisManager.get(getRedisKickoutKey(username));
        if (retryCount == null) {
            //如果用户没有登陆过,登陆次数加1 并放入缓存
            retryCount = new AtomicInteger(0);
        }
        if (retryCount.incrementAndGet() > 5) {
            //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
            User user = userMapper.findByUserName(username);
            if (user != null && "0".equals(user.getState())){
                //数据库字段 默认为 0  就是正常状态 所以 要改为1
                //修改数据库的状态字段为锁定
                user.setState("1");
                userMapper.update(user);
            }
            logger.info("锁定用户" + user.getUsername());
            //抛出用户锁定异常
            throw new LockedAccountException();
        }
        //判断用户账号和密码是否正确
        boolean matches = super.doCredentialsMatch(token, info);
        if (matches) {
            //如果正确,从缓存中将用户登录计数 清除
            redisManager.del(getRedisKickoutKey(username));
        }{
            redisManager.set(getRedisKickoutKey(username), retryCount);
        }
        return matches;
    }

    /**
     * 根据用户名 解锁用户
     * @param username
     * @return
     */
    public void unlockAccount(String username){
        User user = userMapper.findByUserName(username);
        if (user != null){
            //修改数据库的状态字段为锁定
            user.setState("0");
            userMapper.update(user);
            redisManager.del(getRedisKickoutKey(username));
        }
    }

}
```

ShiroSessionListener.java(session 监听)

```java
package com.springboot.test.shiro.config.shiro;

import com.springboot.test.shiro.Application;
import com.springboot.test.shiro.modules.user.dao.entity.User;
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.session.Session;
import org.apache.shiro.session.SessionListener;
import org.springframework.beans.factory.annotation.Autowired;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.http.HttpSessionAttributeListener;
import javax.servlet.http.HttpSessionBindingEvent;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * @author: wangsaichao
 * @date: 2018/5/15
 * @description: 配置session监听器,
 */
public class ShiroSessionListener implements SessionListener{

    /**
     * 统计在线人数
     * juc包下线程安全自增
     */
    private final AtomicInteger sessionCount = new AtomicInteger(0);

    /**
     * 会话创建时触发
     * @param session
     */
    @Override
    public void onStart(Session session) {
        //会话创建，在线人数加一
        sessionCount.incrementAndGet();
    }

    /**
     * 退出会话时触发
     * @param session
     */
    @Override
    public void onStop(Session session) {
        //会话退出,在线人数减一
        sessionCount.decrementAndGet();
    }

    /**
     * 会话过期时触发
     * @param session
     */
    @Override
    public void onExpiration(Session session) {
        //会话过期,在线人数减一
        sessionCount.decrementAndGet();
    }

    /**
     * 获取在线人数使用
     * @return
     */
    public AtomicInteger getSessionCount() {
        return sessionCount;
    }
}
```

上面的类中有一些依赖类,并没有贴出来,该些类是为了解决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: