没有学历拼技术

没有技术拼态度

一 Redis命令

`Redis 命令用于在 redis 服务上执行操作。要在 redis 服务上执行命令需要一个 redis 客户端。` `Redis 客户端在我们之前下载的的 redis 的安装包中。`

Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)等

常用命令key管理

```
keys  * : 返回满足的所有键 ,可以模糊匹配 比如 keys  abc* 代表 abc 开头的 key
exists key ：是否存在指定的key，存在返回1，不存在返回0
expire key second：设置某个key的过期时间 时间为秒
del key:删除某个key
ttl key：查看剩余时间,当key不存在时，返回 -2;存在但没有设置剩余生存时间时，返回 -1,否则，以秒为单位，返回 key 的剩余生存时间。
persist key：取消过去时间
PEXPIRE key milliseconds 修改key 的过期时间为毫秒
select :  选择数据库  数据库为0-15（默认一共16个数据库） s
设计成多个数据库实际上是为了数据库安全和备份
move key dbindex ： 将当前数据中的key转移到其他数据库
randomkey：随机返回一个key
rename key key2：重命名key
echo：打印命令
dbsize：查看数据库的key数量
info：查看数据库信息
config get * 实时传储收到的请求，返回相关的配置
flushdb ：清空当前数据库
flushall ：清空所有数据库
```

DEL key

```
该命令用于在 key 存在时删除 key。
```

EXISTS key

```
检查给定 key 是否存在。
```

EXPIRE key seconds

为给定 key 设置过期时间（以秒计）。

PEXPIRE key milliseconds

设置 key 的过期时间以毫秒计。

TTL key

以秒为单位，返回给定 key 的剩余生存时间(TTL, time to live)

PTTL key

以毫秒为单位返回 key 的剩余的过期时间。

KEYS pattern

```
查找所有符合给定模式( pattern)的 key 。
keys 通配符       获取所有与pattern匹配的key,返回所有与该匹配
  通配符：
          *   代表所有
          ?  表示代表一个字符
```

RENAME key newkey

修改Key的名称

MOVE key db

将当前数据库的 key 移动到给定的数据库 db 当中

TYPE key

返回 key 所储存的值的类型

应用场景

EXPIRE key seconds

1、限时的优惠活动信息 2、网站数据缓存（对于一些需要定时更新的数据，例如：积分排行榜） 3、手机验证码 4、限制网站访客访问频率（例如：1分钟最多访问10次）

Key的命名建议

redis单个key允许存入512M大小

- 1.key不要太长，尽量不要超过1024字节，这不仅消耗内存，而且会降低查找的效率；
- 2.key也不要太短，太短的话，key的可读性会降低；
- 3.在一个项目中，key最好使用统一的命名模式，例如user:123:password;
- 4.key名称区分大小写

二 Redis数据类型

String类型

简介

string类型是Redis最基本的数据类型，一个键最大能存储512MB。

string 数据结构是简单的key-value类型，value其不仅是string，也可以是数字，是包含很多种类型的特殊类型，

string类型是二进制安全的。意思是redis的string可以包含任何数据。 比如序列化的对象进行存储，比如一张图片进行二进制存储，比如一个简单的字符串,数值等等。

String命令

```
赋值语法:
  SET KEY_NAME   VALUE: (说明：多次设置name会覆盖) (Redis SET 命令用于设置给定 key 的值。如果 key 已经存储值， SET 就覆写旧值，且无视类型)

命令：
  SETNX key1 value:(not exist) 如果key1不存在，则设值 并返回1。如果key1存在，则不设值并返回0；(解决分布式锁 方案之一,只有在 key 不存在时设置 key 的值。Setnx（SET if Not eXists） 命令在指定的 key 不存在时，为 key 设置指定的值)

  SETEX key1 10 lx :(expired) 设置key1的值为lx，过期时间为10秒，10秒后key1清除（key也清除）

  SETRANGE string range value: 替换字符串



取值语法:
 GET KEY_NAME :Redis GET命令用于获取指定 key 的值。如果 key 不存在，返回 nil 。如果key 储存的值不是字符串类型，返回一个错误。
 GETRANGE key start end :用于获取存储在指定 key 中字符串的子字符串。字符串的截取范围由 start 和 end 两个偏移量决定(包括 start 和 end 在内)

GETBIT key offset :对 key 所储存的字符串值，获取指定偏移量上的位(bit)

GETSET语法：  GETSET  KEY_NAME  VALUE :Getset 命令用于设置指定 key 的值，并返回 key 的旧值,当 key 不存在时，返回 nil
```

```
STRLEN key :返回 key 所储存的字符串值的长度


删值语法：
DEL KEY_Name :删除指定的KEY，如果存在，返回值数字类型。

批量写：MSET k1 v1 k2 v2 ... 一次性写入多个值
批量读：MGET k1 k2 k3

GETSET name value :一次性设值和读取（返回旧值，写上新值）

自增/自减：
INCR KEY_Name :Incr 命令将 key 中储存的数字值增1。如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 INCR 操作
 自增：INCRBY KEY_Name :增量值 Incrby 命令将 key 中储存的数字加上指定的增量值
 自减：DECR KEY_NAME       或    DECYBY KEY_NAME 减值 :DECR 命令将 key 中储存的数字减1

 :(注意这些 key 对应的必须是数字类型字符串,否则会出错,)

字符串拼接：APPEND  KEY_NAME VALUE
 :Append 命令用于为指定的 key 追加至末尾，如果不存在，为其赋值

字符串长度 :STRLEN key
```

应用场景

- 1、**String通常用于保存单个字符串或JSON字符串数据**
- 2、**因String是二进制安全的，所以你完全可以把一个图片文件的内容作为字符串来存储**
- 3、**计数器（常规key-value缓存应用。常规计数: 微博数, 粉丝数）**

> INCR等指令本身就具有原子操作的特性，所以我们完全可以利用redis的INCR、INCRBY、DECR、DECRBY等指令来实现原子计数的效果。假如，在某种场景下有3个客户端同时读取了mynum的值（值为2），然后对其同时进行了加1的操作，那么，最后mynum的值一定是5。 不少网站都利用redis的这个特性来实现业务上的统计计数需求。

Hash类型

简介

```
Hash类型是String类型的field和value的映射表，或者说是一个String集合。hash特别适合用于存储对象，相比较而言，将一个对象类型存储在Hash类型要存储在String类型里占用更少的内存空间，并对整个对象的存取。
可以看成具有KEY和VALUE的MAP容器，该类型非常适合于存储值对象的信息，
如：uname,upass,age等。该类型的数据仅占用很少的磁盘空间（相比于JSON）。
Redis 中每个 hash 可以存储 2的32次方  - 1 键值对（40多亿）
```

Hash命令

常用命令

```
赋值语法：
 HSET KEY  FIELD VALUE     :为指定的KEY，设定FILD/VALUE
 HMSET KEY FIELD VALUE [FIELD1,VALUE1]…… :同时将多个 field-value (域-值)对设置到哈希表 key 中。

取值语法：
  HGET KEY FIELD    :获取存储在HASH中的值，根据FIELD得到VALUE
  HMGET KEY field[field1]       :获取key所有给定字段的值
  HGETALL KEY       :返回HASH表中所有的字段和值

HKEYS KEY   :获取所有哈希表中的字段
HLEN KEY    :获取哈希表中字段的数量

删除语法：
   HDEL KEY field1[field2]     :删除一个或多个HASH表字段

其它语法：
HSETNX key field value :只有在字段 field 不存在时，设置哈希表字段的值

HINCRBY key field increment :为哈希表 key 中的指定字段的整数值加上增量 increment 。

HINCRBYFLOAT key field increment  :为哈希表 key 中的指定字段的浮点数值加上增量 increment 。

HEXISTS key field  :查看哈希表 key 中，指定的字段是否存在
```

应用场景

Hash的应用场景：（存储一个用户信息对象数据）1、 常用于存储一个对象 2、 为什么不用string存储一个对象？

```
    hash是最接近关系数据库结构的数据类型，可以将数据库一条记录或程序中一个对象转换成hashmap存放在redis中。
用户ID为查找的key，存储的value用户对象包含姓名，年龄，生日等信息，如果用普通的key/value结构来存储，主要有以下2种存储方式：
    第一种方式将用户ID作为查找key,把其他信息封装成一个对象以序列化的方式存储，这种方式的缺点是，增加了序列化/反序列化的开销，并且在需要修改其中一项信息时，需要把整个对象取回，并且修改操作需要对并发进行保护，引入CAS等复杂问题。
    第二种方法是这个用户信息对象有多少成员就存成多少个key-value对儿，用用户ID+对应属性的名称作为唯一标识来取得对应属性的值，虽然省去了序列化开销和并发问题，但是用户ID为重复存储，如果存在大量这样的数据，内存浪费还是非常可观的。
```

总结： Redis提供的Hash很好的解决了这个问题，Redis的Hash实际是内部存储的Value为一个HashMap，并提供了直接存取这个Map成员的接口

三 常用的redis客户端介绍以及对比

> Jedis api 在线网址：http://tool.oschina.net/uploads/apidocs/redis/clients/jedis/Jedis.html

> redisson 官网地址：https://redisson.org/
>
> redisson git项目地址：https://github.com/redisson/redisson
>
> lettuce 官网地址：https://lettuce.io/
>
> lettuce git项目地址：https://github.com/lettuce-io/lettuce-core
>
> 首先，在spring boot2之后，对redis连接的支持，默认就采用了lettuce。这就一定程度说明了lettuce 和Jedis的优劣。

概念：

> Jedis：是老牌的Redis的Java实现客户端，提供了比较全面的Redis命令的支持，
> Redisson：实现了分布式和可扩展的Java数据结构。
> Lettuce：高级Redis客户端，用于线程安全同步，异步和响应使用，支持集群，Sentinel，管道和编码器。

优点：

> Jedis：比较全面的提供了Redis的操作特性
> Redisson：促使使用者对Redis的关注分离，提供很多分布式相关操作服务，例如，分布式锁，分布式集合，可通过Redis支持延迟队列
> Lettuce：基于Netty框架的事件驱动的通信层，其方法调用是异步的。Lettuce的API是线程安全的，所以可以操作单个Lettuce连接来完成各种操作

可伸缩：

> Jedis：使用阻塞的I/O，且其方法调用都是同步的，程序流需要等到sockets处理完I/O才能执行，不支持异步。Jedis客户端实例不是线程安全的，所以需要通过连接池来使用Jedis。
> Redisson：基于Netty框架的事件驱动的通信层，其方法调用是异步的。Redisson的API是线程安全的，所以可以操作单个Redisson连接来完成各种操作
> Lettuce：基于Netty框架的事件驱动的通信层，其方法调用是异步的。Lettuce的API是线程安全的，所以可以操作单个Lettuce连接来完成各种操作
> lettuce能够支持redis4，需要java8及以上。
> lettuce是基于netty实现的与redis进行同步和异步的通信。

lettuce和jedis比较：

> jedis使直接连接redis server,如果在多线程环境下是非线程安全的，这个时候只有使用连接池，为每个jedis实例增加物理连接 ；
> lettuce的连接是基于Netty的，连接实例（StatefulRedisConnection）可以在多个线程间并发访问，StatefulRedisConnection是线程安全的，所以一个连接实例可以满足多线程环境下的并发访问，当然这也是可伸缩的设计，一个连接实例不够的情况也可以按需增加连接实例。
> Redisson实现了分布式和可扩展的Java数据结构，和Jedis相比，功能较为简单，不支持字符串操作，不支持排序、事务、管道、分区等Redis特性。Redisson的宗旨是促进使用者对Redis的关注分离，从而让使用者能够将精力更集中地放在处理业务逻辑上。

总结：

> 优先使用Lettuce，如果需要分布式锁，分布式集合等分布式的高级特性，添加Redisson结合使用，因为Redisson本身对字符串的操作支持很差。
> 在一些高并发的场景中，比如秒杀，抢票，抢购这些场景，都存在对核心资源，商品库存的争夺，控制不好，库存数量可能被减少到负数，出现超卖的情况，或者 产生唯一的一个递增ID，由于web应用部署在多个机器上，简单的同步加锁是无法实现的，给数据库加锁的话，对于高并发，1000/s的并发，数据库可能由行锁变成表锁，性能下降会厉害。那相对而言，redis的分布式锁，相对而言，是个很好的选择，redis官方推荐使用的Redisson就提供了分布式锁和相关服务。

在官方网站列一些Java客户端访问，有：Jedis/Redisson/Jredis/JDBC-Redis等，其中官方推荐使用Jedis和Redisson。常用**Jedis**。

四 SpringBoot整合Jedis

简介

我们在使用springboot搭建微服务的时候，在很多时候还是需要redis的高速缓存来缓存一些数据，存储一些高频率访问的数据，如果直接使用redis的话又比较麻烦，在这里，我们使用jedis来实现redis缓存来达到高效缓存的目的

引入Jedis依赖

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>
```

> 因为 SpringBoot 内默认引用了jedis版本。
>
> 所以我们直接引入jedis 依赖 无需在配置 jedis的版本号了。

application.yml

例如 在application.yml 中配置 如下信息：

```
spring:
  redis:
    port: 6379
    password: guoweixin
    host: 192.168.20.135
    jedis:
      pool:
        max-idle: 6    #最大空闲数
        max-active: 10 #最大连接数
        min-idle: 2    #最小空闲数
    timeout: 2000   #连接超时
```

编写Config

创建类：com.qfjy.config.jedis.JedisConfig

```
package com.qfjy.config.jedis;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPoolConfig;

/**
 * @ClassName JedisConfig
 * @Description TODO
 * @Author guoweixin
 * @Version 1.0
 */
@Configuration
public class JedisConfig {

    private Logger logger = LoggerFactory.getLogger(JedisConfig.class);

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

    @Value("${spring.redis.port}")
    private int port;

    @Value("${spring.redis.password}")
    private String password;
    @Value("${spring.redis.timeout}")
    private int timeout;

    @Value("${spring.redis.jedis.pool.max-active}")
    private int maxActive;

    @Value("${spring.redis.jedis.pool.max-idle}")
    private int maxIdle;

    @Value("${spring.redis.jedis.pool.min-idle}")
    private int minIdle;

    @Bean
    public JedisPool  jedisPool(){
        JedisPoolConfig jedisPoolConfig=new JedisPoolConfig();
        jedisPoolConfig.setMaxIdle(maxIdle);
        jedisPoolConfig.setMinIdle(minIdle);
        jedisPoolConfig.setMaxTotal(maxActive);
```

```
        JedisPool jedisPool=new JedisPool(jedisPoolConfig,host,port,timeout,password);

        logger.info("JedisPool连接成功:"+host+"\t"+port);

        return jedisPool;
    }
}
```

测试配置

```
@SpringBootTest
public class JedisApplicationTests {
    @Autowired
    private JedisPool jedisPool;

    @Test
 public   void contextLoads() {
        System.out.println(jedisPool);
        //在连接池中得到Jedis连接
        Jedis jedis=jedisPool.getResource();
        jedis.set("haha","你好");
        jedis.set("name","guoweixin");
        //关闭当前连接
        jedis.close();

    }
```

封装工具类

JedisUtil

```
package com.qfjy.config.jedis;
import org.springframework.beans.factory.annotation.Autowired;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
/**
 * @ClassName JedisUtils
 * @Description TODO
 * @Author guoweixin
 * @Version 1.0
 */
@Component
public class JedisUtils {
    @Autowired
    private JedisPool jedisPool;

    /**
     * 获取Jedis资源
     */
    public Jedis getJedis(){
```

```
        return jedisPool.getResource();
    }
    /**
     * 释放Jedis连接
     */
    public void close(Jedis jedis){
        if(jedis!=null){
            jedis.close();
        }
    }
    ......................

}
```

测试

Jedis操作String类型

业务JedisServiceImpl类

```
/**
 * @ClassName JedisServiceImpl
 * @Description TODO
 * @Author guoweixin
 * @Version 1.0
 */
@Service
@Log  //日志的处理
public class JedisServiceImpl {

    @Autowired
    private JedisUtils jedisUtils;
    /**
     * 测试String
     * 根据key 查询value值
     */
    public String  getString(String key){
        Jedis jedis=jedisUtils.getJedis();
        String val=null;
        if(!jedis.exists(key)){
            val="千锋南京";
            log.info(key+"在MYSQL数据库中进行查询的结果是："+val);
            jedis.set(key,val);
            log.info(key+"存入Redis中。值是："+val);

        }else{
             val=jedis.get(key);
            log.info(key+"是在Redis中查询的数据。值是："+val);
        }
        jedisUtils.close(jedis); //释放资源
        return val;
    }
```

```
}
```

单元测试

```
@SpringBootTest
public class JedisTests {
    @Autowired
    private JedisServiceImpl jedisService;

    @Test
    void t1(){
       String val= jedisService.getString("name");
       System.out.println(val);
    }
}
```

Jedis操作Hash类型

业务JedisServiceImpl类

```
@Service
@Log
public class JedisServiceImpl {

    @Autowired
    private JedisUtils jedisUtils;
/**
     * 测试 jedis 操作hash类型
     * 根据用户ID查询用户信息
     * 先判断Redis中是否存在，
     * 如果不存在，数据库中查询。并存到Redis中
     * 如果存在，直接查询Redis 并返回
     */
    public User selectBy(String id){
        String key="user:id"; //根据规则生成相同规范的key
        User user=new User();;
        Jedis jedis=jedisUtils.getJedis();
        if(!jedis.exists(key)){
            //数据库中查询，并进行存
            user.setId(id);
            user.setName("千锋南京JAVA");
            user.setAge(20);
            log.info("数据库中查询的用户信息是："+user);
            Map<String,String> map=new  HashMap();
            map.put("id",user.getId());
            map.put("name",user.getName());
            jedis.hset(key,map);
            log.info(key+"成功存入Redis:"+user);
        }
        else{
            Map<String,String> map= jedis.hgetAll(key);
            user.setId(map.get("id"));
```

```
            user.setName(map.get("name"));
            log.info(key+"Redis中查询出来的是:"+map);
        }
        jedisUtils.close(jedis);

        return user;
    }
}
```

单元测试

```
@SpringBootTest
public class JedisTests {
    @Autowired
    private JedisServiceImpl jedisService;
    @Test
    void hash(){
       User user= jedisService.selectBy("1001");
        System.out.println(user);
    }
}
```

五 SpringBoot2.x中redis使用(lettuce)

java代码操作Redis，需要使用Jedis，也就是redis支持java的第三方类库
注意:Jedis2.7以上的版本才支持集群操作

maven配置

新建SpringBoot2.0.3的WEB工程，在MAVEN的pom.xml文件中加入如下依赖

```
<dependencies>
       <!--默认是lettuce客户端-->
       <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-redis</artifactId>
        </dependency>

        <!-- redis依赖commons-pool 这个依赖一定要添加 -->
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-pool2</artifactId>
        </dependency>
        <!-- 测试库依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

hibernate-->JPA-->SpringData

配置文件配置

application.yml

```
spring:
  redis:
    port: 6379
    password: guoweixin
    host: 192.168.20.135
    lettuce:
      pool:
        max-active: 8 # 连接池最大连接数（使用负值表示没有限制）
        max-idle: 8 # 连接池中的最大空闲连接
        min-idle: 0 # 连接池中的最小空闲连接
        max-wait: 1000 # 连接池最大阻塞等待时间（使用负值表示没有限制）
      shutdown-timeout: 100    # 关闭超时时间
```

redis配置类

JdbcTemplate-->JDBC 进一步封装。

RedisTemplate-->redis进行了进一步封装（lettuce)

简介

> 编写缓存配置类RedisConfig用于调优缓存默认配置，RedisTemplate<String, Object>的类型兼容性更高
>
> 大家可以看到在redisTemplate()这个方法中用**JacksonJsonRedisSerializer**更换掉了Redis默认的序列化方式：**JdkSerializationRedisSerializer**
>
> spring-data-redis中序列化类有以下几个：
>
> GenericToStringSerializer：可以将任何对象泛化为字符创并序列化 Jackson2JsonRedisSerializer：序列化Object对象为json字符创（与JacksonJsonRedisSerializer相同） JdkSerializationRedisSerializer：序列化java对象 StringRedisSerializer：简单的字符串序列化
>
> JdkSerializationRedisSerializer序列化被序列化对象必须实现Serializable接口，被序列化除属性内容还有其他内容，长度长且不易阅读,默认就是采用这种序列化方式
>
> 存储内容如下：
>
> "\xac\xed\x00\x05sr\x00!com.oreilly.springdata.redis.User\xb1\x1c
> \n\xcd\xed%\xd8\x02\x00\x02I\x00\x03ageL\x00\buserNamet\x00\x12Ljava/lang/String;xp\x00\x00\x00\
> x14t\x00\x05user1"

> JacksonJsonRedisSerializer序列化,被序列化对象不需要实现Serializable接口，被序列化的结果清晰，容易阅读，而且存储字节少，速度快
>
> 存储内容如下：
>
> "{"userName":"guoweixin","age":20}"
>
> StringRedisSerializer序列化
>
> 一般如果key、value都是string字符串的话，就是用这个就可以了

RedisConfig类

```
package com.qfjy.config;

import com.fasterxml.jackson.annotation.JsonAutoDetect;
import com.fasterxml.jackson.annotation.PropertyAccessor;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.CachingConfigurerSupport;
import org.springframework.cache.interceptor.KeyGenerator;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.cache.RedisCacheConfiguration;
import org.springframework.data.redis.cache.RedisCacheManager;
import org.springframework.data.redis.cache.RedisCacheWriter;
import org.springframework.data.redis.connection.lettuce.LettuceConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.Jackson2JsonRedisSerializer;
import org.springframework.data.redis.serializer.StringRedisSerializer;

import java.lang.reflect.Method;

/**
 * @ClassName RedisConfig
 * @Description TODO
 * @Author guoweixin
 * @Version 1.0
 */
@Configuration
public class RedisConfig extends CachingConfigurerSupport {

    /**
     * 自定义缓存key的生成策略。默认的生成策略是看不懂的(乱码内容) 通过Spring 的依赖注入特性进行自定义的配置注入并且此类是一个配置类可以更多程度的自定义配置
     *
     * @return
     */
    @Bean
    @Override
    public KeyGenerator keyGenerator() {
        return new KeyGenerator() {
```

```
            @Override
            public Object generate(Object target, Method method, Object... params) {
                StringBuilder sb = new StringBuilder();
                sb.append(target.getClass().getName());
                sb.append(method.getName());
                for (Object obj : params) {
                    sb.append(obj.toString());
                }
                return sb.toString();
            }
        };
    }

    /**
     * 缓存配置管理器
     */
    @Bean
    public CacheManager cacheManager(LettuceConnectionFactory factory) {
        //以锁写入的方式创建RedisCacheWriter对象
        RedisCacheWriter writer = RedisCacheWriter.lockingRedisCacheWriter(factory);
        //创建默认缓存配置对象
        RedisCacheConfiguration config = RedisCacheConfiguration.defaultCacheConfig();
        RedisCacheManager cacheManager = new RedisCacheManager(writer, config);
        return cacheManager;
    }

    @Bean
    public RedisTemplate<String,Object> redisTemplate(LettuceConnectionFactory factory){
        RedisTemplate<String,Object> template = new RedisTemplate <>();
        template.setConnectionFactory(factory);

        Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new
Jackson2JsonRedisSerializer(Object.class);
        ObjectMapper om = new ObjectMapper();
        om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
        om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
        jackson2JsonRedisSerializer.setObjectMapper(om);

        StringRedisSerializer stringRedisSerializer = new StringRedisSerializer();

        // 在使用注解@Bean返回RedisTemplate的时候，同时配置hashKey与hashValue的序列化方式。
        // key采用String的序列化方式
        template.setKeySerializer(stringRedisSerializer);
        // value序列化方式采用jackson
        template.setValueSerializer(jackson2JsonRedisSerializer);

        // hash的key也采用String的序列化方式
        template.setHashKeySerializer(stringRedisSerializer);
        // hash的value序列化方式采用jackson
        template.setHashValueSerializer(jackson2JsonRedisSerializer);
        template.afterPropertiesSet();
        return template;
    }
```

```
}
```

代码示例

RedisServiceImpl

测试String类型

```
@Service
@Log
public class RedisServiceImpl {
    @Autowired
    private RedisTemplate<String,Object> redisTemplate;


    /**
     * 普通缓存放入
     * @param key 键
     * @return true成功 false失败
     */
    public String getString(String key) {
        if(redisTemplate.hasKey(key)) {
            log.info("Redis中查询");
           return (String) redisTemplate.opsForValue().get(key);
        }else{
            String val="guoweixin";
            redisTemplate.opsForValue().set(key, val);
            log.info("数据库中查询的");
            return  val;
        }

    }

    /**
     * 普通缓存放入
     * @param key 键
     * @param value 值
     * @param expireTime 超时时间(秒)
     * @return true成功 false失败
     */
    public Boolean set(String key, Object value, int expireTime) {
        try {
            redisTemplate.opsForValue().set(key, value, expireTime, TimeUnit.SECONDS);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }
}
```

测试Hash类型

```
@Service
@Log
public class RedisServiceImpl {
    @Autowired
    private RedisTemplate<String,Object> redisTemplate;

    @Resource(name="redisTemplate")
    private HashOperations<String,String,User> hash;
     /**
     * 判断key是否存在，如果存在 在Redis中查询
     * 如果不存在，在MYSQL中查询，并将结果得到，添加到Redis Hash中
     * @param id
     * @return
     */
      public User selectUserById1(String id){
        if(hash.hasKey("user",id)){
            log.info("Redis中查询对象");
            return  hash.get("user",id);
        }else{
            User u=new User();
            u.setId(id);
            u.setName("guoweixin");
            u.setAge(22);
            log.info("mysql中查询对象");
            hash.put("user",id,u);
            return u;
        }
    }
}
```

hash类型代码示例

```
package com.qfjy.redis.demo.service.impl;

import com.qfjy.redis.demo.service.HashCacheService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.Cursor;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.ScanOptions;
import org.springframework.stereotype.Service;
import org.springframework.util.CollectionUtils;

import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.TimeUnit;
```

```java
@Service("hashCacheService")
public class HashCacheServiceImpl implements HashCacheService {
    private final static Logger log = LoggerFactory.getLogger(HashCacheServiceImpl.class);

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    /**
     * 获取MAP中的某个值
     * @param key  键
     * @param item 项
     * @return 值
     */
    public Object hget(String key, String item) {
        return redisTemplate.opsForHash().get(key, item);
    }

    /**
     * 获取hashKey对应的所有键值
     * @param key 键
     * @return 对应的多个键值
     */
    public Map <Object, Object> hmget(String key) {
        return redisTemplate.opsForHash().entries(key);
    }

    /**
     * 以map集合的形式添加键值对
     * @param key 键
     * @param map 对应多个键值
     * @return true 成功 false 失败
     */
    public boolean hmset(String key, Map <String, Object> map) {
        try {
            redisTemplate.opsForHash().putAll(key, map);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * HashSet 并设置时间
     * @param key  键
     * @param map  对应多个键值
     * @param time 时间(秒)
     * @return true成功 false失败
     */
    public boolean hmset(String key, Map <String, Object> map, long time) {
        try {
            redisTemplate.opsForHash().putAll(key, map);
            if (time > 0) {
```

```
                expire(key, time);
            }
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 向一张hash表中放入数据,如果不存在将创建
     * @param key   键
     * @param item  项
     * @param value 值
     * @return true 成功 false失败
     */
    public boolean hset(String key, String item, Object value) {
        try {
            redisTemplate.opsForHash().put(key, item, value);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 向一张hash表中放入数据,如果不存在将创建
     * @param key   键
     * @param item  项
     * @param value 值
     * @param time  时间(秒) 注意:如果已存在的hash表有时间,这里将会替换原有的时间
     * @return true 成功 false失败
     */
    public boolean hset(String key, String item, Object value, long time) {
        try {
            redisTemplate.opsForHash().put(key, item, value);
            if (time > 0) {
                expire(key, time);
            }
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 删除hash表中的值
     * @param key  键 不能为null
     * @param item 项 可以使多个 不能为null
     */
    public void hdel(String key, Object... item) {
```

```
        redisTemplate.opsForHash().delete(key, item);
    }

    /**
     * 判断hash表中是否有该项的值
     * @param key  键 不能为null
     * @param item 项 不能为null
     * @return true 存在 false不存在
     */
    public boolean hHasKey(String key, String item) {
        return redisTemplate.opsForHash().hasKey(key, item);
    }

    /**
     * hash递增 如果不存在,就会创建一个 并把新增后的值返回
     * @param key  键
     * @param item 项
     * @param by   要增加几(大于0)
     * @return
     */
    public long hincr(String key, String item, long by) {
        return redisTemplate.opsForHash().increment(key, item, by);
    }

    /**
     * hash递减
     * @param key  键
     * @param item 项
     * @param by   要减少记(小于0)
     * @return
     */
    public long hdecr(String key, String item, long by) {
        return redisTemplate.opsForHash().increment(key, item, -by);
    }

    /**
     * 获取指定变量中的hashMap值。
     * @param key
     * @return 返回LIST对象
     */
    @Override
    public List<Object> values(String key) {
        return redisTemplate.opsForHash().values(key);
    }

    /**
     *  获取变量中的键。
     * @param key
     * @return 返回SET集合
     */
    @Override
    public Set<Object> keys(String key) {
        return  redisTemplate.opsForHash().keys(key);
```

```
    }

    /**
     * 获取变量的长度。
     * @param key 键
     * @return  返回长度
     */
    @Override
    public long size(String key) {
        return redisTemplate.opsForHash().size(key);
    }

    /**
     * 以集合的方式获取变量中的值。
     * @param key
     * @param list
     * @return 返回LIST集合值
     */
    @Override
    public List multiGet(String key, List list) {
        return redisTemplate.opsForHash().multiGet(key,list);
    }

    /**
     *  如果变量值存在，在变量中可以添加不存在的的键值对
     *  如果变量不存在，则新增一个变量，同时将键值对添加到该变量。
     * @param key
     * @param hashKey
     * @param value
     */
    @Override
    public void putIfAbsent(String key, String hashKey, Object value) {
        redisTemplate.opsForHash().putIfAbsent(key,hashKey,value);
    }

    /**
     * 匹配获取键值对，ScanOptions.NONE为获取全部键对，
ScanOptions.scanOptions().match("map1").build()
     * 匹配获取键位map1的键值对,不能模糊匹配。
     * @param key
     * @param options
     * @return
     */
    @Override
    public Cursor<Map.Entry<Object, Object>> scan(String key, ScanOptions options) {
        return  redisTemplate.opsForHash().scan(key,options);
    }

    /**
     * 删除变量中的键值对，可以传入多个参数，删除多个键值对。
     * @param key 键
     * @param hashKeys MAP中的KEY
     */
```

```
    @Override
    public void delete(String key, String... hashKeys) {
        redisTemplate.opsForHash().delete(key,hashKeys);
    }

    public boolean expire(String key, long seconds) {
        return redisTemplate.expire(key,seconds,TimeUnit.SECONDS);
    }

    /**
     * 删除
     * @param keys
     */
    @Override
    public void del(String... keys) {
        if (keys != null && keys.length > 0) {
            if (keys.length == 1) {
                redisTemplate.delete(keys[0]);
            } else {
                redisTemplate.delete(CollectionUtils.arrayToList(keys));
            }
        }
    }

    @Override
    public long getExpire(String key) {
        return 0;
    }
}
```

六 作业

手机验证功能

需求描述：

用户在客户端输入手机号，点击发送后随机生成4位数字码。有效期为20秒。
输入验证码，点击验证，返回成功或者失败。且每个手机号在5分钟内只能验证3次。并给相应信息提示

限制登录功能

需求描述：

用户在2分钟内，仅允许输入错误密码5次。
如果超过次数，限制其登录1小时。（要求每登录失败时，都要给相应提式）

代码参考：

需求详细分析逻辑：

```
/**
1判断当前登录的用户是否被限制登录
1.1如果没有被限制
（执行登录功能）
2判断是否登录成功
    2.1 登录成功-->（清除输入密码错误次数信息）
2.2登录不成功
    3记录登录错误次数
（判断Redis中的登录次数KEY是否存在）user:loginCount:fail:用户名
      3.1如果不存在
是第一次登录失败次数为1    user:loginCount:fail:用户名进行赋值，同时设置失效期
          3.2如果存在
查询登录失败次数的key结果
if(结果<4)
  user:loginCount:fail:+1
else{//4
  限制登录KEY存在，同时设置限制登录时间锁定1小时。
}
1.2如果被限制
做相应提示
*/
```

业务Service层：

```
/**
 * 1判断当前登录的用户是否被限制登录
 * 查询当前KEY是否存在，如果存在，就被限制 注意：需要给用户做提示：您当前的用户已被限制，还剩多长时间
 * 如果不存在，就不被限制。
 */
@Override
public Map<String,Object> loginUserLock(String uname) {
    String key=User.getLoginTimeLockKey(uname);
    Map<String,Object> map=new HashMap<String,Object>();
    if(redisTemplate.hasKey(key)){
        long lockTime=redisTemplate.getExpire(key,TimeUnit.MINUTES);//以分钟为单位进行返回
        //如果存在
        map.put("flag",true);
        map.put("lockTime", lockTime);//还剩多长时间（小时单位锁定：给用户返回分钟）

    }else{
        map.put("flag", false);
    }
    return map;
}
```

```java
/**
 * 登录不成功相应操作
 */
@Override
public String loginValdate(String uname) {
    //记录登录错误次数key
    String key=User.getLoginCountFailKey(uname);
    int num=5;//登录错误的次数
    if(!redisTemplate.hasKey(key)){//如果不存在
        //是第一次登录失败次数为1 赋值为1和设置失效期2分钟 user:loginCount:fail:用户名进行赋值，同时设置失效期
        //注意：redisTemplate中在赋值时不能直接赋值并设置失效期（会设置失败）
        redisTemplate.opsForValue().set(key,"1");//先赋值
        redisTemplate.expire(key, 2,TimeUnit.MINUTES);//再设置失效期2分钟
        return "登录失败，在2分钟内还允许输入错误"+(num-1)+"次，";
    }
    else{//如果存在
        //查询登录失败次数的key结果
        long loginFailCount=Long.parseLong(redisTemplate.opsForValue().get(key));
        if(loginFailCount<(num-1)){//代表如果当前登录失败次数<4 意思：还有资格继续进行登录
            //user:loginCount:fail:+1 登录次数+1
            redisTemplate.opsForValue().increment(key, 1);//对指定KEY 增加指定数据
            long seconds=redisTemplate.getExpire(key, TimeUnit.SECONDS);//返回的是秒
            return uname+"登录失败，在"+seconds+" 秒内还允许输入错误"+(num-loginFailCount-1)+"次";
        }else{ //超过指定登录次数
            //限制登录KEY存在，同时设置限制登录时间锁定1小时。

            //限制登录KEY存在，同时设置限制登录时间锁定1小时。
            redisTemplate.opsForValue().set(User.getLoginTimeLockKey(uname),"1");
            redisTemplate.expire(User.getLoginTimeLockKey(uname), 1,TimeUnit.HOURS);
            return "因登录失败次数超过限制"+num+"次，已对其限制登录1小时";
        }
    }
}
```

Controller层

erviceImpl.java | UserController.java

```java
@RequestMapping(produces = {"application/json;charset=UTF-8"},value="login")
public String login(@RequestParam(name="username") final String username,
                    @RequestParam(name="password",required=false)final String password,
                    @RequestParam(name="valcode",required=false)final String valcode){

    /** * 1验证码比较*/
    /**
     * 2执行登录功能
     * 2.1先判断当前用户是被限制登录
     */
    //2.1先判断当前用户是被限制登录
    Map<String,Object> map=userService.loginUserLock(username);
    if((boolean) map.get("flag")){//被限制登录
        return "登录失败，因"+username+"用户超过了限制登录次数，已被禁止登录。还剩："+map.get("lockTime")+"分钟";
    }else{//如果没有被限制登录
        // （执行登录功能）
        User user=userService.login(username, password);

        //判断是否登录成功
        if(user!=null){ //登录成功
            //清空对应的所有KEY
            return "/succ.jsp";
        }else{//登录不成功
            String result=userService.loginValdate(username);
            return result;
```