

# 一、Redis介绍

## 1.1 锋迷商城项目问题

- **数据库访问压力**: 为了降低对数据库的访问压力, 当多个用户请求相同的数据时, 我们可以将第一次从数据库查询到的数据进行缓存(存储在内存中), 以减少对数据库的访问次数
- **首页数据的加载效率**: 将大量的且不经常改变的数据缓存在内存中, 可以大幅度提高访问速度
- **集群部署下的商品超卖**: 分布式事务
- **用户登录**: 分布式会话

## 1.2 Redis介绍

### 1.2.1 Redis的产生背景

- 2008年 萨尔瓦多——开发一个进行网站实时统计软件项目(LLOOGG), 项目的实时统计功能需要频繁的进行数据库的读写(对数据库的读写要求很高—数千次/s), MySQL满足不了项目的需求, 萨尔瓦多就使用C语言自定义了一个数据存储系统—Redis。后来萨尔瓦多不满足仅仅在LLOOGG这个项目中使用redis, 就对redis进行产品化并进行开源, 以便让更多的人能够使用。

### 1.2.2 Redis使用

Redis就是一个用C语言开发的、基于内存结构进行**键值对**数据存储的、高性能的、非关系型NoSQL数据库

### 1.2.3 Redis支持的数据类型

redis是基于键值对进行数据存储的, 但是value可以是多种数据类型:

- string 字符串
- hash 映射
- list 列表 (队列)
- set 集合
- zset 无序集合

### 1.2.4 Redis特点

- 基于内存存储, 数据读写效率很高
- Redis本身支持持久化
- Redis虽然基于key-value存储, 但是支持多种数据类型
- Redis支持集群、支持主从模式

## 1.3 Redis应用场景

- 缓存: 在绝大多数的互联网项目中, 为了提供数据的访问速度、降低数据库的访问压力, 我们可以使用redis作为缓存来实现
- 点赞、排行榜、计数器等功能: 对数据实时读写要求比较高, 但是对数据库的一致性要求并不是太高的功能场景
- 分布式锁: 基于redis的操作特性可以实现分布式锁功能

- 分布式会话：在分布式系统中可以使用redis实现 **session** (共享缓存)
- 消息中间件：可以使用redis实现应用之间的通信

## 1.4 Redis的优缺点

### 1.4.1 优点

- redis是基于内存结构，性能极高（读 110000次/秒，写 81000次/秒）
- redis基于键值对存储，但是支持多种数据类型
- redis的所有操作都是原子性，可以通过lua脚本将多个操作合并为一个院子操作（Redis的事务）
- reids是基于单线程操作，但是其多路复用实现了高性能读写

### 1.4.2 缺点

- 缓存数据与数据库数据必须通过两次写操作才能保持数据的一致性
- 使用缓存会存在缓存穿透、缓存击穿及缓存雪崩等问题，需要处理
- redis可以作为数据库使用进行数据的持久存储，存在丢失数据的风险

## 二、Redis安装及配置

### 2.1 Redis安装

基于linux环境安装redis

#### 2.1.1 下载Redis

```
 wget http://download.redis.io/releases/redis-5.0.5.tar.gz
```



#### 2.1.2 安装redis

- 安装gcc

```
 yum -y install gcc
```



- 解压redis安装包

```
 tar -zxvf redis-5.0.5.tar.gz
```



- 解压之后进入到redis-5.0.5目录

```
 cd redis-5.0.5
```



- 编译

```
 make MALLOC=libc
```



- 安装

```
 make install
```



- 启动redis

```
## 当我们完成redis安装之后，就可以执行redis相关的指令  
redis-server ## 启动redis服务  
redis-server &
```



- 打开客户端

```
redis-cli ## 启动redis操作客户端(命令行客户端)
```



## 2.2 Redis配置

- 使用 `redis-server` 指令启动redis服务的时候，可以在指令后添加redis配置文件的路径，以设置redis是以何种配置进行启动

```
redis-server redis-6380.conf & ## redis以redis-6380.conf文件中的配置来启动
```



- 如果不指定配置文件的名字，则按照redis的默认配置启动（默认配置≠redis.conf）
- 我们可以通过创建redis根目录下 `redis.conf` 来创建多个配置文件，启动多个redis服务

```
redis-server redis-6380.conf &  
redis-server redis-6381.conf &
```



### 常用redis配置

```
## 设置redis实例（服务）为守护模式，默认值为no，可以设置为yes  
daemonize no
```



```
## 设置当前redis实例启动之后保存进程id的文件路径  
pidfile /var/run/redis_6379.pid
```

```
## 设置redis实例的启动端口（默认6379）  
port 6380
```

```
## 设置当前redis实例是否开启保护模式  
protected-mode yes
```

```
## 设置允许访问当前redis实例的ip地址列表  
bind 127.0.0.1
```

```
## 设置连接密码  
requirepass 123456
```

```
## 设置redis实例中数据库的个数（默认16个，编号0-15）  
databases 16
```

```
## 设置最大并发数量  
maxclients
```

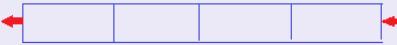
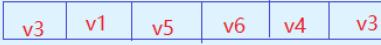
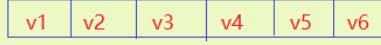
```
## 设置客户端和redis建立连接的最大空闲时间，设置为0表示不限制
```

```
timeout 0
```

## 三、Redis基本使用

### 3.1 Redis存储的数据结构

Redis是以键值对形式进行数据存储的，但是value支持多种数据类型

"key1"	"zhangsan"	字符串 string						
"key2"	<table border="1"><tr><td>field1</td><td>value1</td></tr><tr><td>field2</td><td>value2</td></tr><tr><td>field3</td><td>value3</td></tr></table>	field1	value1	field2	value2	field3	value3	映射 hash
field1	value1							
field2	value2							
field3	value3							
"key3"		队列 list						
"key4"		无序 两端取值、遍历 集合 set						
"key5"		有序集合 zset						

### 3.2 string常用指令

```
## 设置值/修改值 如果key存在则进行修改  
set key value
```



```
## 取值  
get key
```

```
## 批量添加  
mset k1 v1 [k2 v2 k3 v3 ...]
```

```
## 批量取值  
mget k1 [k2 k3...]
```

```
## 自增和自减  
incr key ## 在key对应的value上自增 +1  
decr key ## 在key对应的value上自减 -1  
incrby key v ## 在key对应的value上+v  
decrby key v ## 在key对应的value上-v
```

```
## 添加键值对，并设置过期时间(TTL)  
setex key time(seconds) value
```

```
## 设置值，如果key不存在则成功添加，如果key存在则添加失败（不做修改操作）  
setnx key value
```

```

## 在指定的key对应value拼接字符串
append key value

## 获取key对应的字符串的长度
strlen key

```

### 3.3 hash常用指令

```

## 向key对应的hash中添加键值对
hset key field value

## 从key对应的hash获取field对应的值
hget key field

## 向key对应的hash结构中批量添加键值对
hmset key f1 v1 [f2 v2 ...]

## 从key对应的hash中批量获取值
hmget key f1 [f2 f3 ...]

## 在key对应的hash中的field对应value上加v
hincrby key field v

## 获取key对应的hash中所有的键值对
hgetall key

## 获取key对应的hash中所有的field
hkeys key

## 获取key对应的hash中所有的value
hvals key

## 检查key对应的hash中是否有指定的field
hexists key field

## 获取key对应的hash中键值对的个数
hlen key

## 向key对应的hash结构中添加f-v, 如果field在hash中已经存在, 则添加失败
hsetnx key field value

```



### 3.4 list常用指令

"key3"		先进先出	对于list数据结构, 支持两侧存取数据 同侧存取: 实现了堆栈结构, 后进先出 异侧存取: 实现了队列结构, 先进先出
--------	---	------	--

```

## 存储数据
lpush key value # 在key对应的列表的左侧添加数据value
rpush key value # 在key对应的列表的右侧添加数据value

```



```

## 获取数据
lpop key # 从key对应的列表的左侧取一个值
rpop key # 从key对应的列表的右侧取一个值

## 修改数据
lset key index value #修改key对应的列表的索引位置的数据（索引从左往右，从0开始）

## 查看key对应的列表中索引从start开始到stop结束的所有值
lrange key start stop

## 查看key对应的列表中index索引对应的值
lindex key index

## 获取key对应的列表中的元素个数
llen key

## 从key对应的列表中截取key在[start, stop]范围的值，不在此范围的数据一律被清除掉
ltrim key start stop

## 从k1右侧取出一个数据放到k2的左侧
rpoplpush k1 k2

```

### 3.5 set常用指令



```

## 存储元素：在key对应的集合中添加元素，可以添加1个，也可以同时添加多个元素
sadd key v1 [v2 v3 v4...]

## 遍历key对应的集合中的所有元素
smembers key

## 随机从key对于的集合中获取一个值（出栈）
spop key

## 交集
sinter key1 key2

## 并集
sunion key1 key2

## 差集
sdiff key1 key2

## 从key对应的集合中移出指定的value
srem key value

## 检查key对应的集合中是否有指定的value
sismember key value

```

## 3.6 zset常用指令

zset 有序不可重复集合 z



```
## 存储数据(score存储位置必须是数值, 可以是float类型的任意数字; member元素不允许重复)
zadd key score member [score member...]
```

```
## 查看key对应的有序集合中索引[start,stop]数据--按照score值由小到大 (start 和 stop指的不是score, 而是元素在有序集合中的索引)
```

```
zrange key start stop
```

```
##查看member元素在key对应的有序集合中的索引
```

```
zscore key member
```

```
## 获取key对应的zset中的元素个数
```

```
zcard key
```

```
## 获取key对应的zset中, score在[min,max]范围内的member个数
```

```
zcount key min max
```

```
## 从key对应的zset中移除指定的member
```

```
zrem key6 member
```

```
## 查看key对应的有序集合中索引[start,stop]数据--按照score值由大到小
```

```
zrevrange key start stop
```

## 3.7 key相关指令



```
## 查看redis中满足pattern规则的所有的key (keys *)
```

```
keys pattern
```

```
## 查看指定的key谁否存在
```

```
exists key
```

```
## 删除指定的key-value对
```

```
del key
```

```
## 获取当前key的存活时间(如果没有设置过期返回-1, 设置过期并且已经过期返回-2)
```

```
ttl key
```

```
## 设置键值对过期时间
```

```
expire key seconds
```

```
pexpire key milliseconds
```

```
## 取消键值对过期时间
```

```
persist key
```

### 3.8 db常用指令

redis的键值对是存储在数据库中的——db

redis中默认有16个db，编号 0-15



```
## 切换数据库
select index

## 将键值对从当前db移动到目标db
move key index

## 清空当前数据库数据
flushdb

## 清所有数据库的k-v
flushall

## 查看当前db中k-v个数
dbsize

## 获取最后一次持久化操作时间
lastsave
```

## 四、Redis的持久化 [重点]

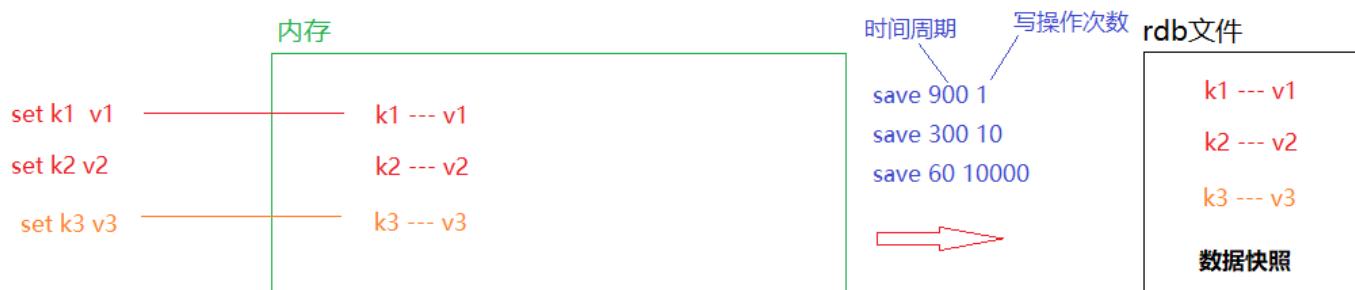
Redis是基于内存操作，但作为一个数据库也具备数据的持久化能力；但是为了实现高效的读写操作，并不会即时进行数据的持久化，而是按照一定的规则进行持久化操作的——持久化策略

Redis提供了2中持久化策略：

- RDB (Redis DataBase)
- AOF(Append Only File)

### 4.1 RDB

在满足特定的redis操作条件时，将内存中的数据以数据快照的形式存储到rdb文件中



- 原理：

RDB是redis默认的持久化策略，当redis中的写操作达到指定的次数、同时距离上一次持久化达到指定的时间就会将redis内存中的数据生成数据快照，保存在指定的rdb文件中。

- 默认触发持久化条件：

- 900s 1次：当操作次数达到1次，900s就会进行持久化
- 300s 10次：当操作次数达到10次，300s就会进行持久化
- 60s 10000次：当操作次数达到10000次，60s就会进行持久化

- 我们可以通过修改redis.conf文件，来设置RDB策略的触发条件：

```
## rdb持久化开关
rdbcompression yes

## 配置redis的持久化策略
save 900 1
save 300 10
save 60 10000

## 指定rdb数据存储的文件
dbfilename dump.rdb
```



- RED持久化细节分析：

### 缺点

- 如果redis出现故障，存在数据丢失的风险，丢失上一次持久化之后的操作数据；
- RDB采用的是数据快照形式进行持久化，不适合实时性持久化；
- 如果数据量巨大，在RDB持久化过程中生成数据快照的子进程执行时间过长，会导致redis卡顿，因此save时间周期设置不宜过短；

### 优点

- 但是在数据量较小的情况下，执行速度比较快；
- 由于RDB是以数据快照的形式进行保存的，我们可以通过拷贝rdb文件轻松实现redis数据移植

## 4.2 AOF

Apeend Only File，当达到设定触发条件时，将redis执行的写操作指令存储在aof文件中，Redis默认未开启aof持久化



- 原理：

Redis将每一个成功的写操作写入到aof文件中，当redis重启的时候就执行aof文件中的指令以恢复数据

- 配置：

```
## 开启AOF
appendonly yes

## 设置触发条件（三选一）
appendfsync always      ## 只要进行成功的写操作，就执行aof
appendfsync everysec    ## 每秒进行一次aof
appendfsync no          ## 让redis执行决定aof

## 设置aof文件路径
appendfilename "appendonly.aof"
```



- AOF细节分析：

- 也可以通过拷贝aof文件进行redis数据移植
- aof存储的指令，而且会对指令进行整理；而RDB直接生成数据快照，在数据量不大时RDB比较快
- aof是对指令文件进行增量更新，更适合实时性持久化
- redis官方建议同时开启2中持久化策略，如果同时存在aof文件和rdb文件的情况下aof优先

## 五、Java应用连接Redis

### 5.1 设置redis允许远程连接

Java应用连接Redis，首先要将我们的Redis设置允许远程连接

- 修改redis-6379.conf

```
## 关闭保护模式
protected-mode no

## 将bind注释掉（如果不注释，默认为 bind 127.0.0.1 只能本机访问）
# bind 127.0.0.1

## 密码可以设置（也可以不设置）
# requirepass 123456
```



- 重启redis

```
redis-server redis-6379.conf
```



- 阿里云安全组设置放行6379端口

手动添加	快速添加	全部编辑	输入端口或者授权对象进行搜索	授权策略	优先级 ①	协议类型	端口范围 ①	授权对象 ①	描述	创建时间	操作
<input type="checkbox"/>	<input checked="" type="radio"/>	允许	1	自定义 TCP			目的: 6379/6379	源: 0.0.0.0/0	redis	2021年5月18日11:35:55	<a href="#">编辑</a> <a href="#">复制</a> <a href="#">删除</a>

## 5.1 在普通Maven工程连接Redis

使用jedis客户端连接

### 5.1.1 添加Jedis依赖

```
<!-- https://mvnrepository.com/artifact/redis.clients/jedis -->
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.9.0</version>
</dependency>
```



### 5.1.2 使用案例

```
public static void main(String[] args) {
    Product product = new Product("101", "娃哈哈AD钙奶", 2.5);

    //1.连接redis
    Jedis jedis = new Jedis("47.96.11.185", 6379);
    //2.操作
    String s = jedis.set(product.getId(), new Gson().toJson(product));
    System.out.println(s);
    //3.关闭连接
    jedis.close();

}
```



### 5.1.3 redis远程可视化客户端

- Redis desktop manager

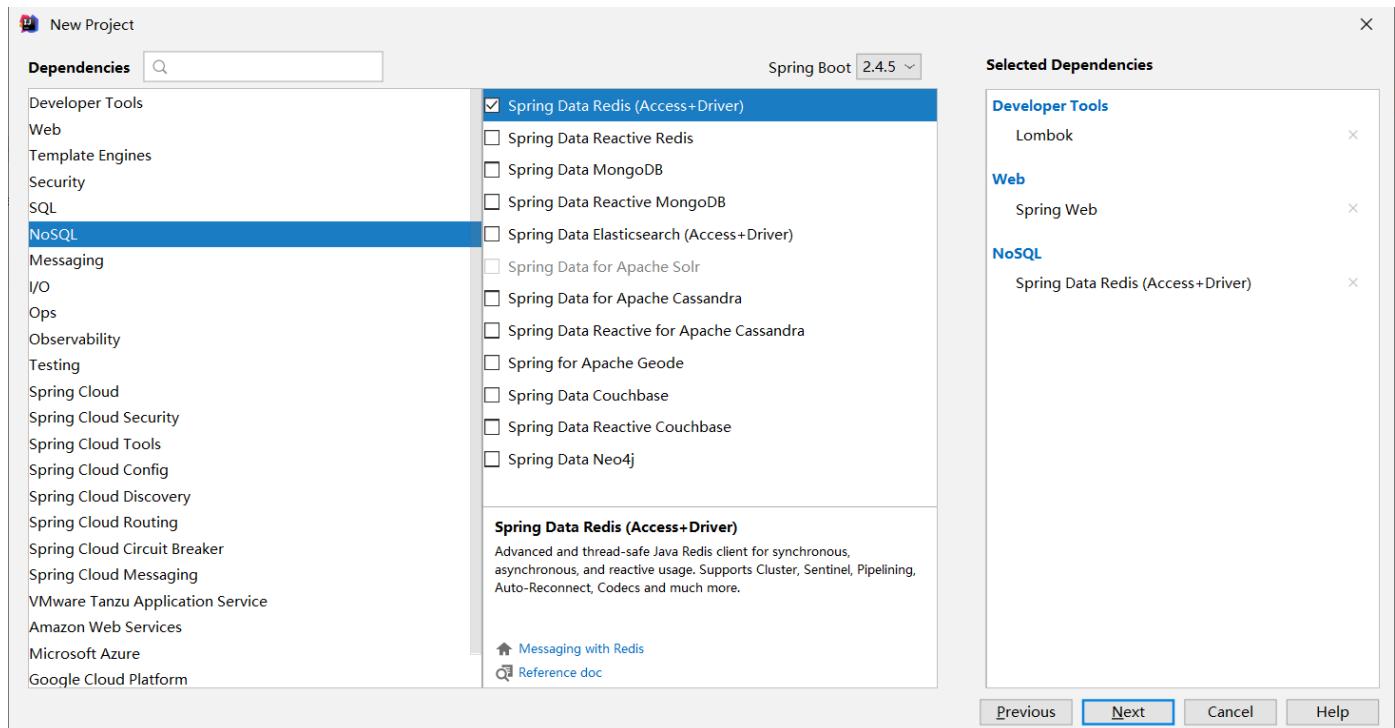
## 5.2 在SpringBoot工程连接Redis

**Spring Data Redis**, part of the larger Spring Data family, provides easy configuration and access to Redis from Spring applications. It offers both low-level and high-level abstractions for interacting with the store, freeing the user from infrastructural concerns.

Spring Data Redis依赖中，提供了用于连接redis的客户端：

- RedisTemplate
- StringRedisTemplate

## 5.2.1 创建springBoot应用



## 5.2.2 配置redis

application.yml文件配置redis的连接信息

```
spring:  
  redis:  
    host: 47.96.11.185  
    port: 6379  
    database: 0  
    password: 123456
```



## 5.2.3 使用redis客户端连接redis

直接在service中注入 `RedisTemplate` 或者 `StringRedisTemplate` ,就可以使用此对象完成redis操作

## 5.3 Spring Data Redis

### 5.3.1 不同数据结构的添加操作

```
//1.string  
//添加数据 set key value  
stringRedisTemplate.boundValueOps(product.getId()).set(jsonstr);  
  
//2.hash  
stringRedisTemplate.boundHashOps("products").put(product.getId(), jsonstr);  
  
//3.list  
stringRedisTemplate.boundListOps("list").leftPush("ccc");
```



```
//4.set
stringRedisTemplate.boundSetOps("s1").add("v2");

//5.zset
stringRedisTemplate.boundZSetOps("z1").add("v1", 1.2);
```

### 5.3.2 string类型的操作方法



```
//添加数据 set key value
stringRedisTemplate.boundValueOps(product.getId()).set(jsonstr);

//添加数据时指定过期时间 setex key 300 value
stringRedisTemplate.boundValueOps("103").set(jsonstr, 300);

//设置指定key的过期时间 expire key 20
stringRedisTemplate.boundValueOps("103").expire(20, TimeUnit.SECONDS);

//添加数据 setnx key value
Boolean absent = stringRedisTemplate.boundValueOps("103").setIfAbsent(jsonstr);
```

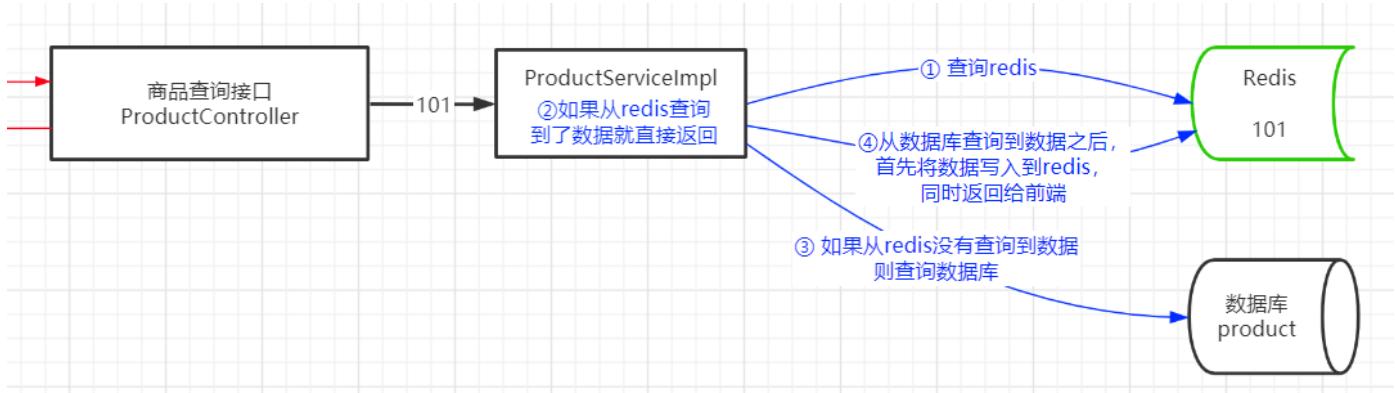
### 5.3.3 不同数据类型的取值操作



```
//string
String o = stringRedisTemplate.boundValueOps("103").get();
//hash
Object v = stringRedisTemplate.boundHashOps("products").get("101");
//list
String s1 = stringRedisTemplate.boundListOps("list").leftPop();
String s2 = stringRedisTemplate.boundListOps("list").rightPop();
String s3 = stringRedisTemplate.boundListOps("list").index(1);
//set
Set<String> vs = stringRedisTemplate.boundSetOps("s1").members();
//zset
Set<String> vs2 = stringRedisTemplate.boundZSetOps("z1").range(0, 5);
```

## 六、使用redis缓存数据库数据

### 6.1 redis作为缓存的使用流程



## 6.2 在使用redis缓存商品详情(锋迷商城)

### 6.2.1 在service子工程添加Spring data redis依赖

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



### 6.2.2 在application.yml配置redis数据源

```
spring:
  datasource:
    druid:
      driver-class-name: com.mysql.jdbc.Driver
      ## 如果后端项目服务器和数据库服务器不在同一台主机，则需要修改localhost为数据库服务器ip地址
      url: jdbc:mysql://localhost:3306/fmmall2?characterEncoding=utf-8
      username: root
      password: admin123
    redis:
      port: 6379
      host: 47.96.11.185
      database: 0
      password: 123456
```



### 6.3.3 在ProductServiceImpl中修改业务代码

products	1	{"productId":"101","productName":"111","productPrice":4.0}
products	1	[{} , {} , {} ]
products	1	[0,0,0]

```

@Service
public class ProductServiceImpl implements ProductService {

    @Autowired
    private ProductMapper productMapper;
    @Autowired
    private ProductImgMapper productImgMapper;
    @Autowired
    private ProductSkuMapper productSkuMapper;
    @Autowired
    private ProductParamsMapper productParamsMapper;

    @Autowired
    private StringRedisTemplate stringRedisTemplate;
    private ObjectMapper objectMapper = new ObjectMapper();

    @Transactional(propagation = Propagation.SUPPORTS)
    public ResultVO getProductBasicInfo(String productId) {
        try{
            // ①根据商品id查询redis
            String productInfo = (String) stringRedisTemplate.boundHashOps( key: "products").get(productId);

            // ②如果redis中查询到了商品信息，则直接返回给控制器
            if(productInfo != null){
                Product product = objectMapper.readValue(productInfo, Product.class);
                //从redis中查询此商品的图片
                String imgsStr = (String) stringRedisTemplate.boundHashOps( key: "productImg").get(productId);
                JavaType javaType1 = objectMapper.getTypeFactory().constructParametricType(ArrayList.class, ProductImg.class);
                List<ProductImg> productImg = objectMapper.readValue(imgsStr, javaType1);
                //从redis中查询此商品的套餐
                String skusStr = (String) stringRedisTemplate.boundHashOps( key: "productSkus").get(productId);
                JavaType javaType2 = objectMapper.getTypeFactory().constructParametricType(ArrayList.class, ProductSku.class);
                List<ProductSku> productSkus = objectMapper.readValue(skusStr, javaType2);
                //封装商品、图片及套餐
                HashMap<String, Object> basicInfo = new HashMap<>();
                basicInfo.put("product", product);
                basicInfo.put("productImg", productImg);
                basicInfo.put("productSkus", productSkus);
                return new ResultVO(ResStatus.OK, msg: "success", basicInfo);
            }else{
                //③如果redis中没有查询到商品信息，则查询数据库
                //商品基本信息
                Example example = new Example(Product.class);
                Example.Criteria criteria = example.createCriteria();
                criteria.andEqualTo( property: "productId", productId);
                criteria.andEqualTo( property: "productStatus", value: 1); //状态为1表示上架商品
                List<Product> products = productMapper.selectByExample(example);
                if(products.size() > 0 ){
                    //④将从数据库查询的数据写入到redis
                    Product product = products.get(0);
                    String jsonStr = objectMapper.writeValueAsString(product);
                    stringRedisTemplate.boundHashOps( key: "products").put(productId, jsonStr); 将商品信息存入到redis

                    //根据商品id查询商品图片
                    Example example1 = new Example(ProductImg.class);
                    Example.Criteria criteria1 = example1.createCriteria();
                    criteria1.andEqualTo( property: "itemId", productId);
                    List<ProductImg> productImg = productImgMapper.selectByExample(example1); 将商品图片存入到redis
                    stringRedisTemplate.boundHashOps( key: "productImg").put(productId, objectMapper.writeValueAsString(productImg) );

                    //根据商品id查询商品套餐
                    Example example2 = new Example(ProductSku.class);
                    Example.Criteria criteria2 = example2.createCriteria();
                    criteria2.andEqualTo( property: "productId", productId);
                    criteria2.andEqualTo( property: "status", value: 1);
                    List<ProductSku> productSkus = productSkuMapper.selectByExample(example2); 将商品套餐存入到redis
                    stringRedisTemplate.boundHashOps( key: "productSkus").put(productId, objectMapper.writeValueAsString(productSkus));
                }
            }
        }
    }
}

```

```
        HashMap<String, Object> basicInfo = new HashMap<>();
        basicInfo.put("product", products.get(0));
        basicInfo.put("productImg", productImg);
        basicInfo.put("productSkus", productSkus);
        return new ResultVO(ResStatus.OK, msg: "success", basicInfo);
    }
} catch (Exception e){
}
return new ResultVO(ResStatus.NO, msg: "fail", data: null);
}
```

## 七、使用Redis做缓存使用存在的问题 [重点]

使用redis做为缓存在高并发场景下有可能出现缓存击穿、缓存穿透、缓存雪崩等问题

### 7.1 缓存击穿

缓存击穿：大量的并发请求同时访问同一个在redis中不存在的数据，就会导致大量的请求绕过redis同时并发访问数据库，对数据库造成了高并发访问压力。

- 使用 双重检测锁 解决 缓存击穿 问题

```
@Service
public class IndexImgServiceImpl implements IndexImgService {

    @Autowired
    private IndexImgMapper indexImgMapper;
    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    private ObjectMapper objectMapper = new ObjectMapper();

    public ResultVO listIndexImgs() {
        List<IndexImg> indexImgs = null;
        try {
            //1000个并发请求，请求轮播图
            String imgsStr = stringRedisTemplate.boundValueOps("indexImgs").get();

            //1000个请求查询到redis中的数据都是null
            if (imgsStr != null) {
                // 从redis中获取到了轮播图信息
                JavaType javaType = objectMapper.getTypeFactory()
                    .constructParametricType(ArrayList.class, IndexImg.class);
                indexImgs = objectMapper.readValue(imgsStr, javaType);
            } else {
                // 1000个请求都会进入else
                // (service类在spring容器中是单例的,
                // 1000个并发会启动1000个线程来处理，但是公用一个service实例)
                synchronized (this){
```



```

        // 第二次查询redis
        String s = stringRedisTemplate.boundValueOps("indexImg").get();
        if(s == null){
            // 这1000个请求中，只有第一个请求再次查询redis时依然为null
            indexImg = indexImgMapper.listIndexImg();
            System.out.println("-----查询数据库");
            stringRedisTemplate.boundValueOps("indexImg")
                .set(objectMapper.writeValueAsString(indexImg));
            stringRedisTemplate.boundValueOps("indexImg")
                .expire(1, TimeUnit.DAYS);
        }else{
            JavaType javaType = objectMapper.getTypeFactory()
                .constructParametricType(ArrayList.class, IndexImg.class);
            indexImg = objectMapper.readValue(s, javaType);
        }
    }

} catch (JsonProcessingException e) {
    e.printStackTrace();
}

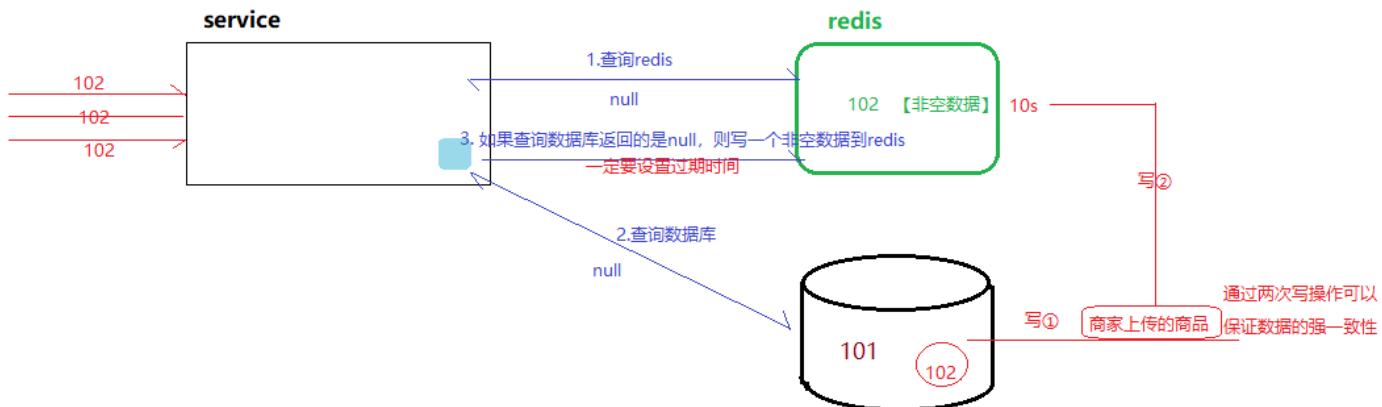
// 返回数据
if(indexImg != null){
    return new ResultVO(ResStatus.OK, "success", indexImg);
} else{
    return new ResultVO(ResStatus.NO, "fail", null);
}
}
}

```

## 7.2 缓存穿透

缓存穿透：大量的请求一个数据库中不存在的数据，首先在redis中无法命中，最终所有的请求都会访问数据库，同样会导致数据库承受巨大的访问压力。

- 解决方案：当从数据库查询到一个null时，写一个非空的数据到redis，并设置过期时间



```
indexImgs = indexImgMapper.listIndexImgs();
if(indexImgs != null) {
    String s = objectMapper.writeValueAsString(indexImgs);
    stringRedisTemplate.boundValueOps("indexImgs").set(s);
    stringRedisTemplate.boundValueOps("indexImgs").expire(1, TimeUnit.DAYS);
} else{
    //当从数据库查询数据为null时，保存一个非空数据到redis，并设置过期时间
    stringRedisTemplate.boundValueOps("indexImgs").set("[ ]");
    stringRedisTemplate.boundValueOps("indexImgs").expire(10, TimeUnit.SECONDS);
}
```

## 7.3 缓存雪崩

缓存雪崩：缓存大量的数据集中过期，导致请求这些数据的大量的并发请求会同时访问数据库

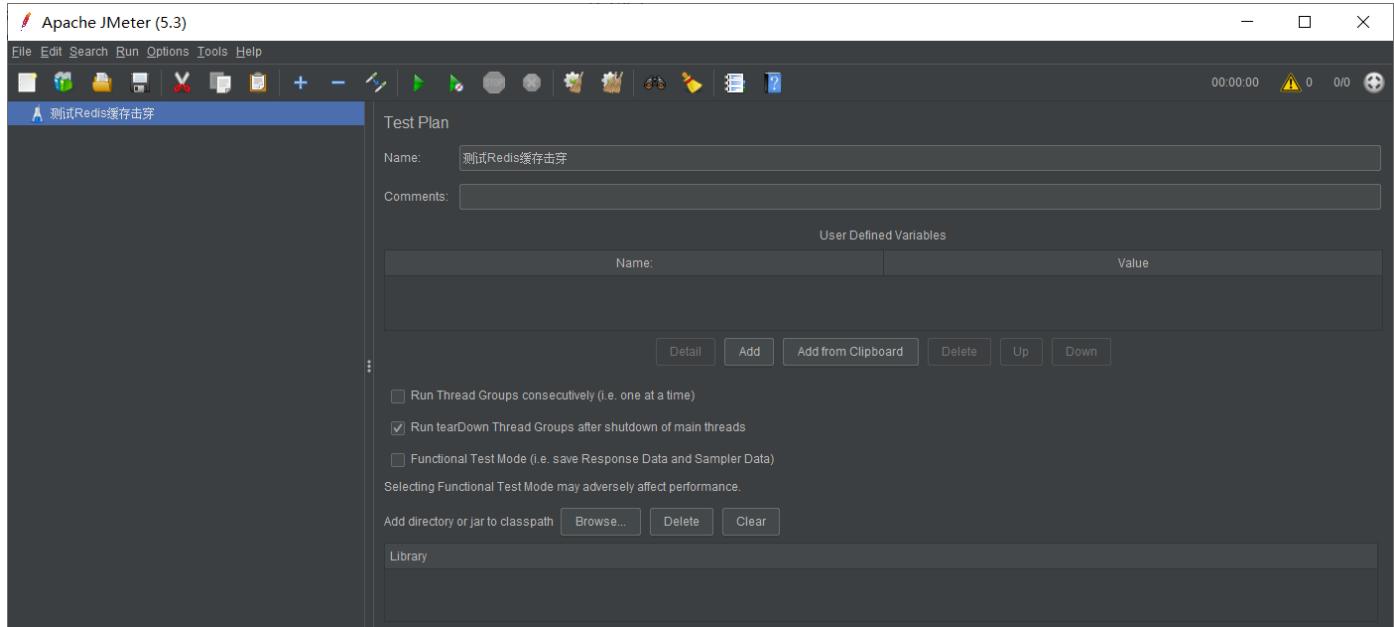
解决方案：

- 将缓存中的数据设置成不同的过期时间
- 在访问洪峰到达前缓存热点数据，过期时间设置到流量最低的时段

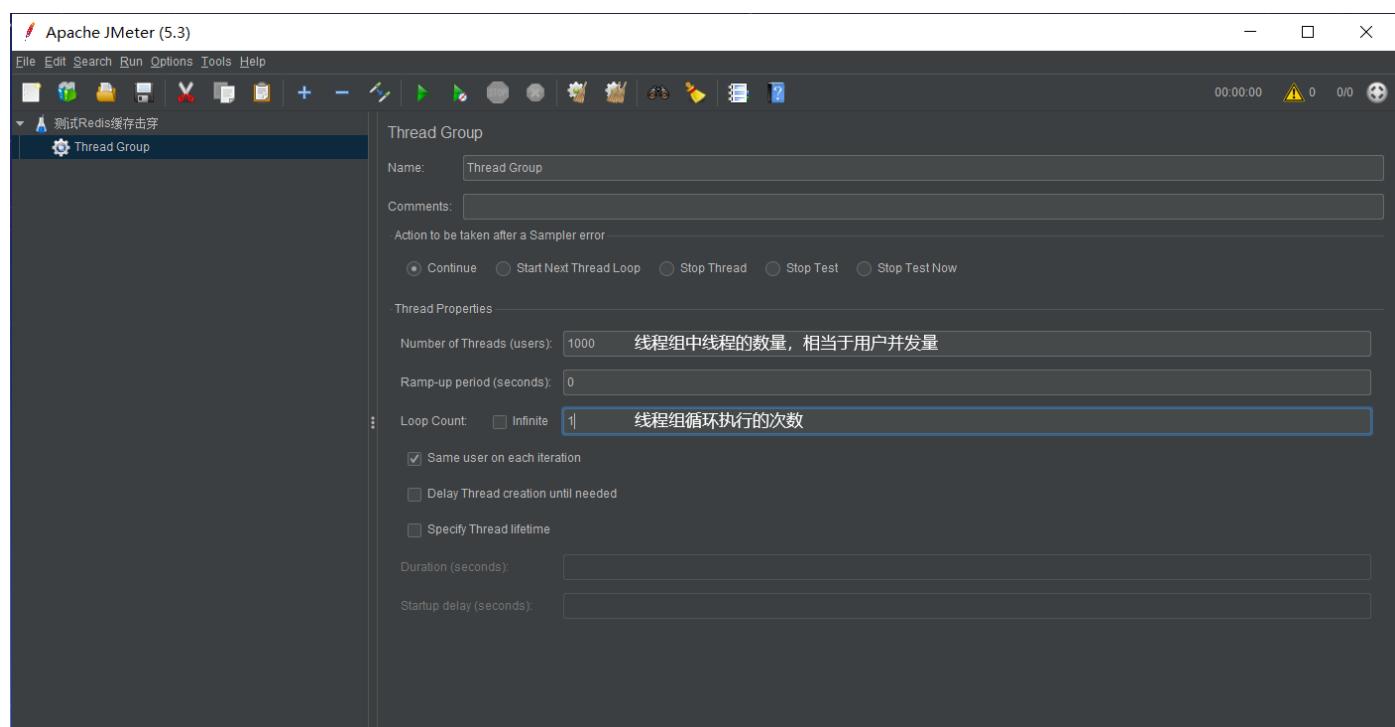
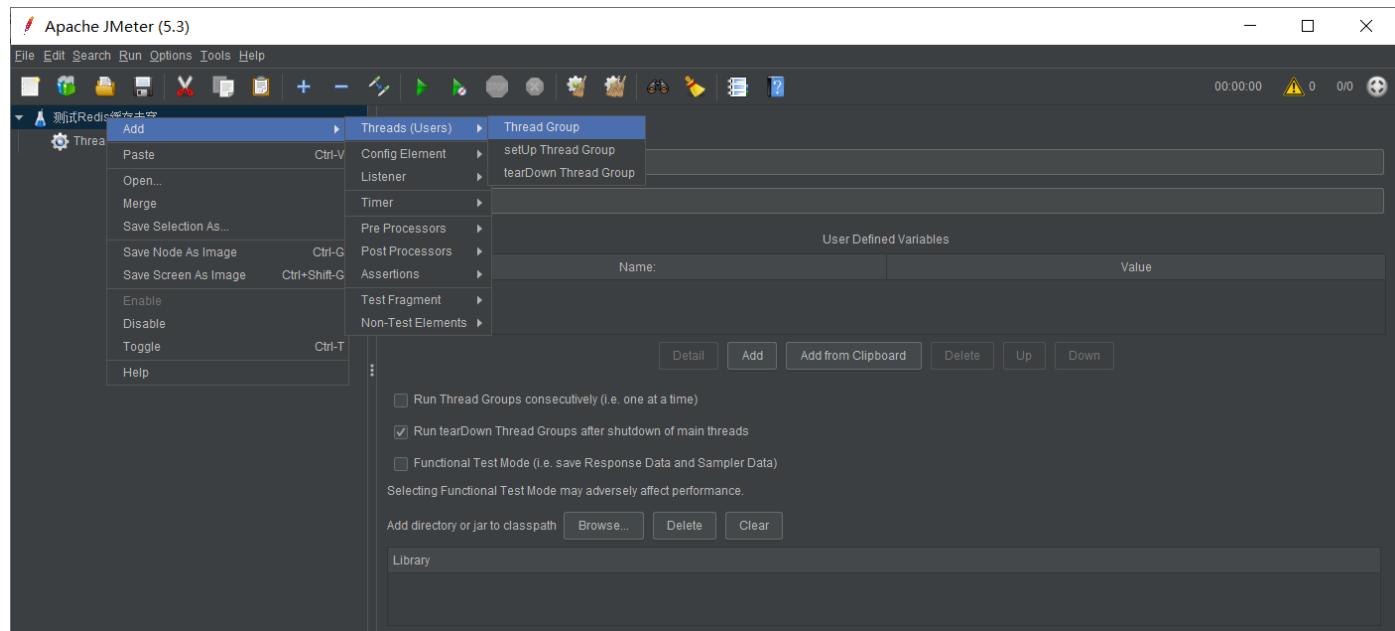
## 7.4 Jmeter测试

Jmeter是基于Java开发的一个测试工具，因此需要先安装JDK

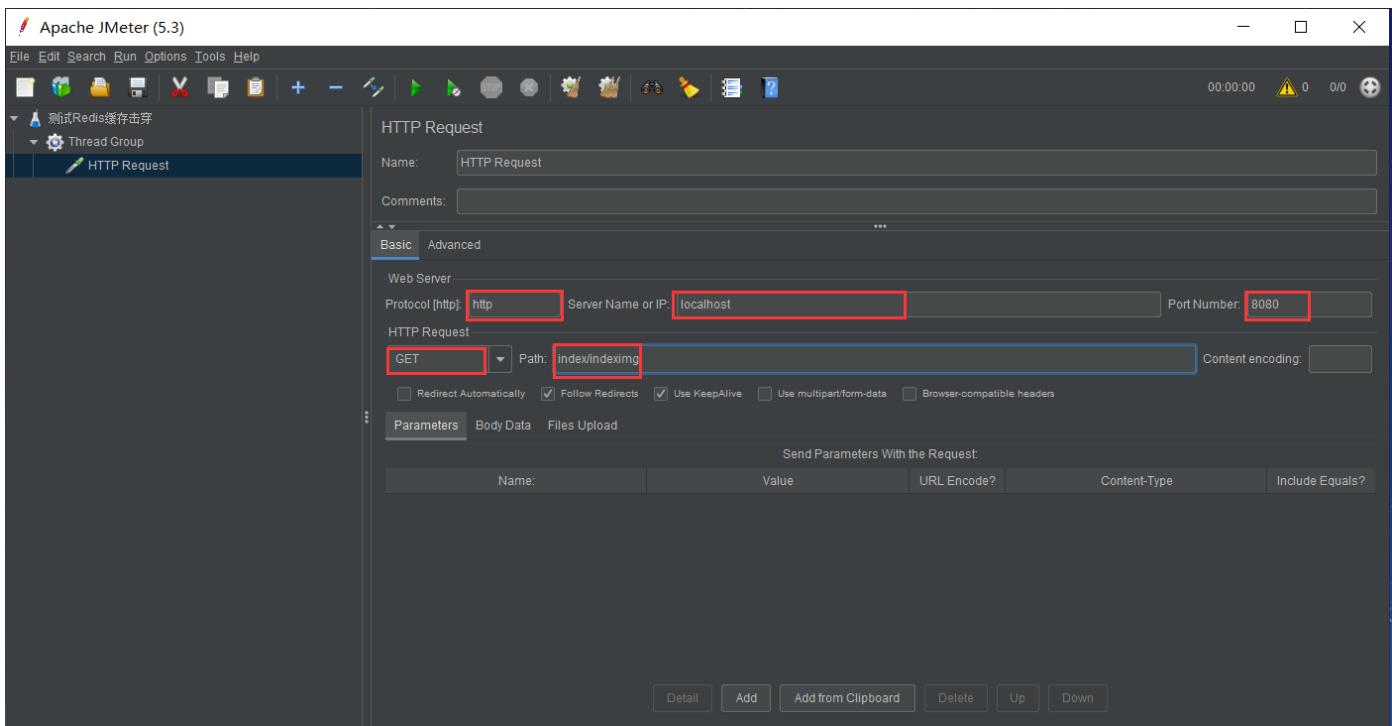
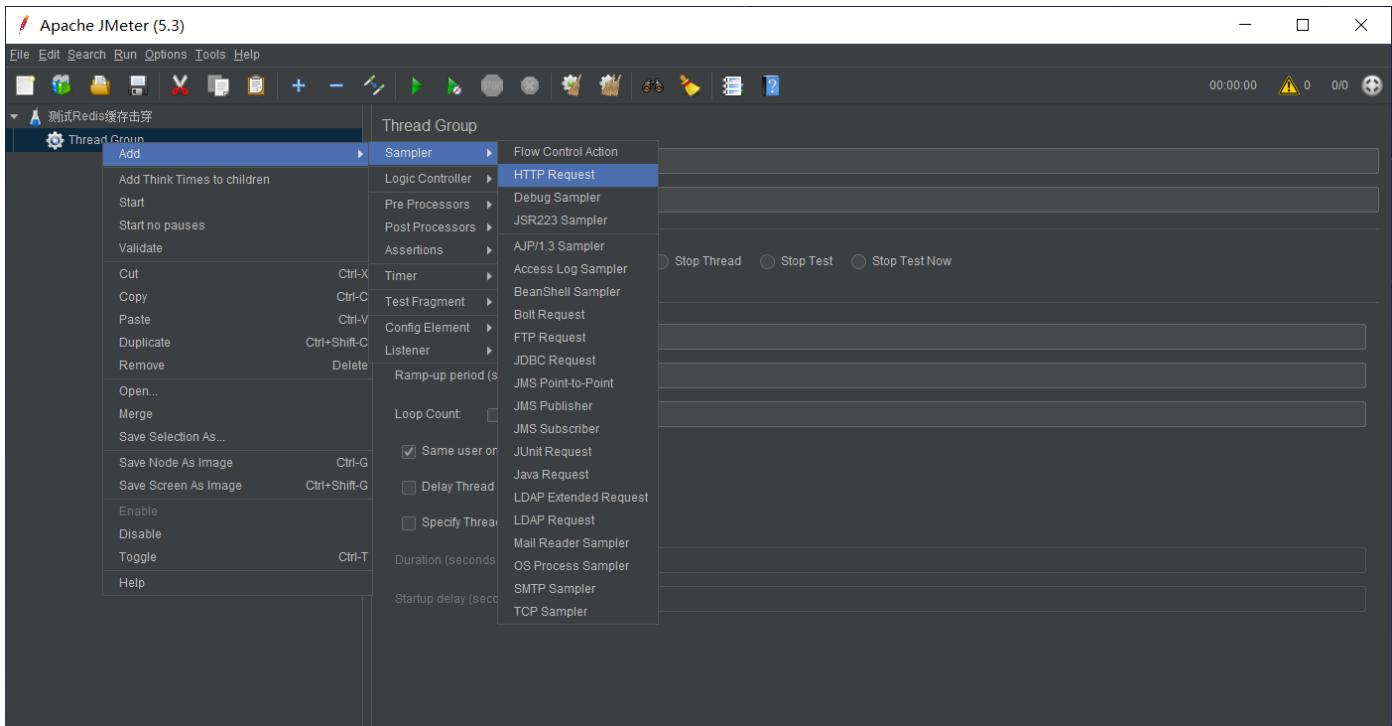
### 7.4.1 创建测试计划



## 7.4.2 创建线程组



## 7.4.3 设置HTTP请求



## 八、Redis高级应用

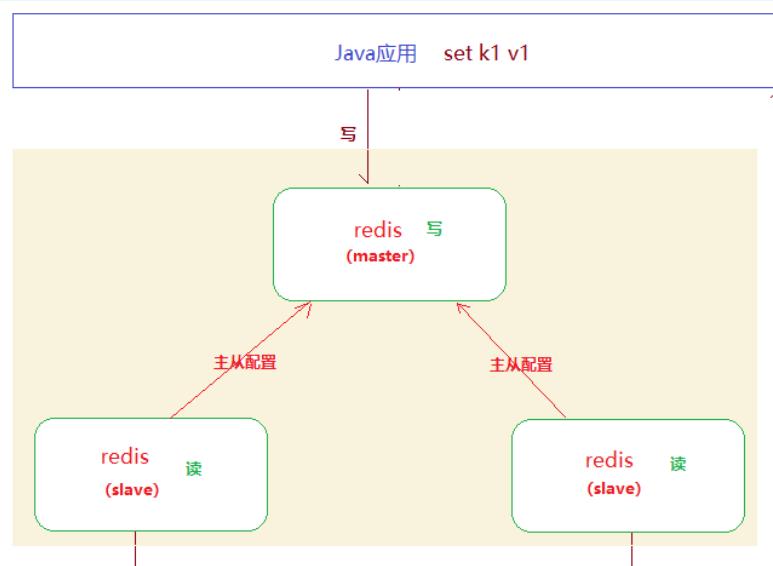
使用redis作为缓存数据库使用目的是为了提升数据加载速度、降低对数据库的访问压力，我们需要保证redis的可用性。

- 主从配置
- 哨兵模式
- 集群配置

## 8.1 主从配置

主从配置：在多个redis实例建立起主从关系，当 **主redis** 中的数据发生变化，**从redis** 中的数据也会同步变化。

- 通过主从配置可以实现redis数据的备份（**从redis** 就是对 **主redis** 的备份），保证数据的安全性；
- 通过主从配置可以实现redis的读写分离



### 主从配置示例

- 启动三个redis实例

```
## 在redis-5.0.5目录下创建 msconf 文件夹
[root@theo redis-5.0.5]# mkdir msconf

## 拷贝redis.conf文件 到 msconf文件夹 ---> redis-master.conf
[root@theo redis-5.0.5]# cat redis.conf |grep -v "#" | grep -v "^$" > msconf/redis-
master.conf

## 修改 redis-master.conf 端口及远程访问设置
[root@theo msconf]# vim redis-master.conf
```

```
# bind 127.0.0.1
protected-mode no
port 6380 取消保护模式，允许远程连接，如果要设置密码，则需要添加 requirepass 123456
tcp-backlog 511
timeout 0
tcp-keepalive 300
daemonize no
supervised no
pidfile /var/run/redis_6380.pid
loglevel notice
logfile ""
databases 16
always-show-logo yes
save 900 1
save 300 10
save 60 10000
stop-writes-on-bgsave-error yes
rdbcompression yes
rdbchecksum yes
dbfilename dump_6380.rdb
dir ./
replica-serve-stale-data yes
replica-read-only yes
repl-diskless-sync no
repl-diskless-sync-delay 5
repl-disable-tcp-nodelay no
replica-priority 100
lazyfree-lazy-eviction no
lazyfree-lazy-expire no
lazyfree-lazy-server-del no
replica-lazy-flush no
appendonly no
appendfilename "appendonly_6380.aof"
appendfsync everysec
no-appendfsync-on-rewrite no
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

```
## 将 redis-master.conf 拷贝两份分别为: redis-slave1.conf  redis-slave2.conf
[root@theo msconf]# sed 's/6380/6381/g' redis-master.conf > redis-slave1.conf
[root@theo msconf]# sed 's/6380/6382/g' redis-master.conf > redis-slave2.conf
```



```
## 修改redis-slave1.conf  redis-slave2.conf, 设置“跟从”---127.0.0.1 6380
[root@theo msconf]# vim redis-slave1.conf
[root@theo msconf]# vim redis-slave2.conf
```

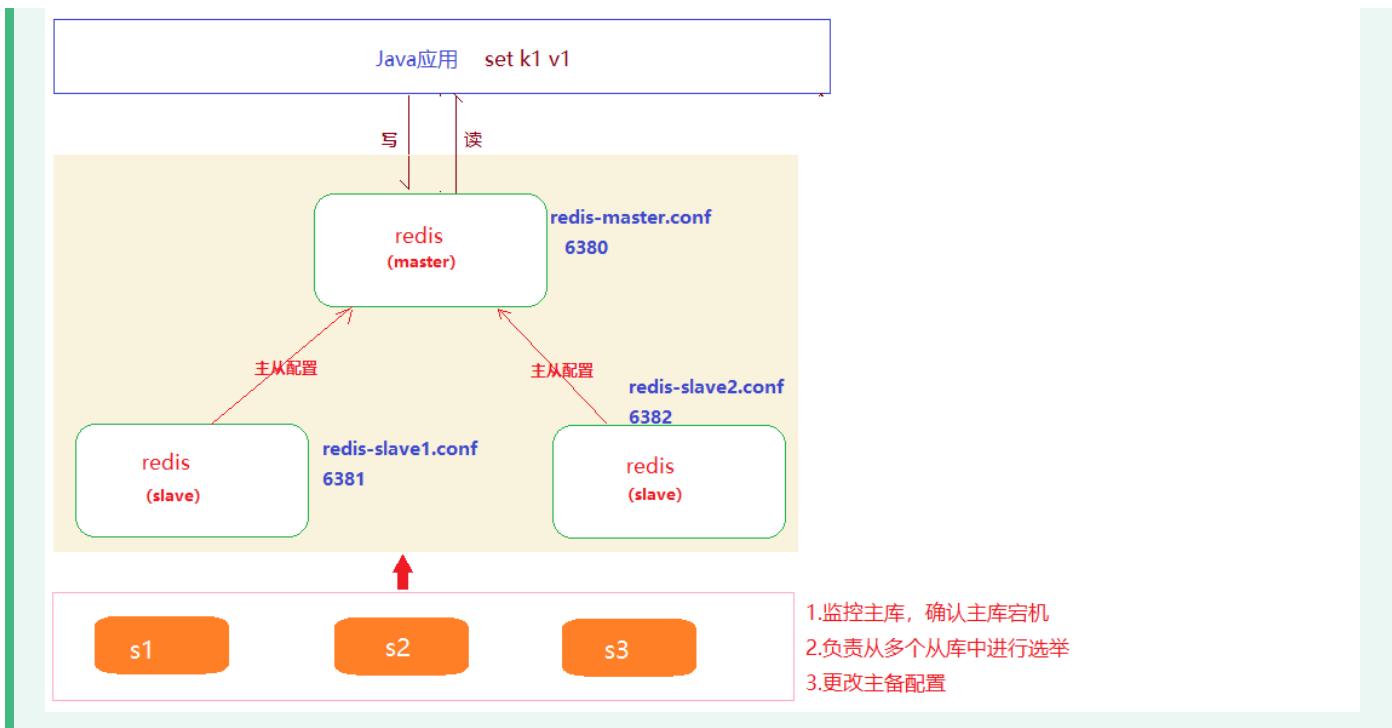
```
# bind 127.0.0.1
protected-mode no
port 6382
slaveof 127.0.0.1 6380
tcp-backlog 511
timeout 0
tcp-keepalive 300
daemonize no
```

```
## 启动三个redis实例
[root@theo msconf]# redis-server redis-master.conf &
[root@theo msconf]# redis-server redis-slave1.conf &
[root@theo msconf]# redis-server redis-slave2.conf &
```



## 8.2 哨兵模式

哨兵模式：用于监听主库，当确认主库宕机之后，从备库(从库)中选举一个转备为主



- 哨兵模式配置

```
##首先实现三个redis实例之间的主从配置（如上）
```



```
## 创建并启动三个哨兵
## 拷贝sentinel.conf文件三份: sentinel-26380.conf  sentinel-26382.conf  sentinel-26382.conf
```

```
## 创建sentinelconf目录
```

```
[root@theo redis-5.0.5]# mkdir sentinelconf
```

```
## 拷贝sentinel.conf文件到 sentinelconf目录: sentinel-26380.conf
```

```
[root@theo redis-5.0.5]# cat sentinel.conf | grep -v "#" | grep -v "^$" >
sentinelconf/sentinel-26380.conf
```

```
[root@theo redis-5.0.5]# cd sentinelconf/
```

```
[root@theo sentinelconf]# ll
```

```
total 4
```

```
-rw-r--r-- 1 root root 326 May 19 17:09 sentinel-26380.conf
```

```
## 编辑 sentinelconf/sentinel-26380.conf文件
```

```
[root@theo sentinelconf]# vim sentinel-26380.conf
```

```
port 26380
daemonize no
pidfile "/var/run/redis-sentinel-26380.pid"
logfile ""
dir "/tmp"
sentinel deny-scripts-reconfig yes
# 此处配置默认的主库的ip 和端口 最后的数字是哨兵数量的一半多一个
sentinel monitor mymaster 127.0.0.1 6380 2
sentinel config-epoch mymaster 1
sentinel leader-epoch mymaster 1
protected-mode no
```



```
[root@theo sentinelconf]# sed 's/26380/26381/g' sentinel-26380.conf > sentinel-26381.conf  
[root@theo sentinelconf]# sed 's/26380/26382/g' sentinel-26380.conf > sentinel-26382.conf
```



测试：

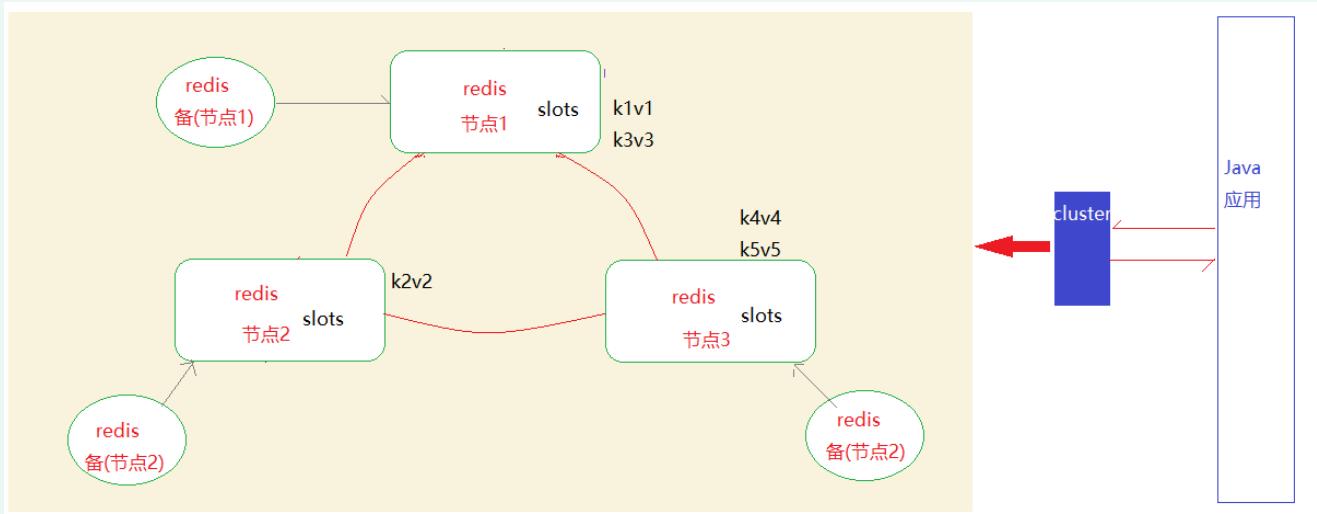
```
启动 主redis  
启动 备1redis  
启动 备2redis  
再依次启动三个哨兵：  
[root@theo sentinelconf]# redis-sentinel sentinel-26380.conf
```



## 8.3 集群配置

高可用：保证redis一直处于可用状态，即时出现了故障也有备用方案保证可用性

高并发：一个redis实例已经可以支持多达11w并发读操作或者8.1w并发写操作；但是如果对于有更高并发需求的应用来说，我们可以通过 **读写分离**、**集群配置** 来解决高并发问题



### Redis集群

- Redis集群中每个节点是对等的，无中心结构
- 数据按照slots分布式存储在不同的redis节点上，节点中的数据可共享，可以动态调整数据的分布
- 可扩展性强，可以动态增删节点，最多可扩展至1000+节点
- 集群每个节点通过主备（哨兵模式）可以保证其高可用性

#### 8.3.1 集群搭建

```
[root@theo ~]# cd /usr/local/redis-5.0.5  
[root@theo redis-5.0.5]# mkdir cluster-conf  
[root@theo redis-5.0.5]# cat redis.conf | grep -v "#" | grep -v "^$" > cluster-conf/redis-7001.conf  
[root@theo redis-5.0.5]# cd cluster-conf/  
[root@theo cluster-conf]# ls  
redis-7001.conf  
[root@theo cluster-conf]# vim redis-7001.conf
```



```

#bind 127.0.0.1 ← ① 注释bind, 关闭保护模式允许远程连接
protected-mode no ←
port 7001 ← ② 修改当前redis实例端口
cluster-enabled yes ←
cluster-config-file nodes 7001.conf ←
cluster-node-timeout 15000 ← ③ 集群配置
tcp-backlog 511
timeout 0
tcp-keepalive 300
daemonize yes
supervised no
pidfile /var/run/redis_7001.pid
loglevel notice
logfile ""
databases 16
always-show-logo yes
save 900 1
save 300 10
save 60 10000
stop-writes-on-bgsave-error yes
rdbcompression yes
rdbchecksum yes
dbfilename dump_7001.rdb
dir ./ ←
replica-serve-stale-data yes
replica-read-only yes
repl-diskless-sync no
repl-diskless-sync-delay 5
repl-disable-tcp-nodelay no
replica-priority 100
lazyfree-lazy-eviction no
lazyfree-lazy-expire no
lazyfree-lazy-server-del no
replica-lazy-flush no
appendonly no
appendfilename "appendonly_7001.aof"

```

- 拷贝6个文件，端口分别为7001-7006

```

[root@theo cluster-conf]# sed 's/7001/7002/g' redis-7001.conf > redis-7002.conf
[root@theo cluster-conf]# sed 's/7001/7003/g' redis-7001.conf > redis-7003.conf
[root@theo cluster-conf]# sed 's/7001/7004/g' redis-7001.conf > redis-7004.conf
[root@theo cluster-conf]# sed 's/7001/7005/g' redis-7001.conf > redis-7005.conf
[root@theo cluster-conf]# sed 's/7001/7006/g' redis-7001.conf > redis-7006.conf

```



- 启动6个redis实例

```

[root@theo cluster-conf]# redis-server redis-7001.conf &
[root@theo cluster-conf]# redis-server redis-7002.conf &
[root@theo cluster-conf]# redis-server redis-7003.conf &
[root@theo cluster-conf]# redis-server redis-7004.conf &
[root@theo cluster-conf]# redis-server redis-7005.conf &
[root@theo cluster-conf]# redis-server redis-7006.conf &

```



- 查看6个实例是否启动

```

[root@theo cluster-conf]# ps -ef|grep redis
root      4789      1  0 10:20 ?          00:00:00 redis-server *:7001 [cluster]
root      4794      1  0 10:20 ?          00:00:00 redis-server *:7002 [cluster]
root      4799      1  0 10:20 ?          00:00:00 redis-server *:7003 [cluster]
root      4806      1  0 10:21 ?          00:00:00 redis-server *:7004 [cluster]
root      4811      1  0 10:21 ?          00:00:00 redis-server *:7005 [cluster]
root      4816      1  0 10:21 ?          00:00:00 redis-server *:7006 [cluster]

```



- 启动集群

```
[root@theo cluster-conf]# redis-cli --cluster create 47.96.11.185:7001 47.96.11.185:7002 47.96.11.185:7003 47.96.11.185:7004 47.96.11.185:7005 47.96.11.185:7006 --cluster-replicas 1
```

```
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 47.96.11.185:7005 to 47.96.11.185:7001
Adding replica 47.96.11.185:7006 to 47.96.11.185:7002
Adding replica 47.96.11.185:7004 to 47.96.11.185:7003
>>> Trying to optimize slaves allocation for anti-affinity
[WARNING] Some slaves are in the same host as their master
M: 4678478aa66bb6d37b23944cf7db0ac07298538a4 47.96.11.185:7001
  slots:[0-5460] (5461 slots) master
M: e26eaf2ae6c52e3477f3d6569428e02ccf63668e 47.96.11.185:7002
  slots:[5461-10922] (5462 slots) master
M: 5752eb202dc70d07ffad3383aa381308b9041c61 47.96.11.185:7003
  slots:[10923-16383] (5461 slots) master
S: 4315316860a5c3c97d7e83cb66047c8e1f5fd72 47.96.11.185:7004
  replicates 4678478aa66bb6d37b23944cf7db0ac07298538a4
S: eb7173417f4392efba0710f6bfde7ef794e100ef 47.96.11.185:7005
  replicates e26eaf2ae6c52e3477f3d6569428e02ccf63668e
S: a5f35724529659cdd13019309bca5e163cd4d32d 47.96.11.185:7006
  replicates 5752eb202dc70d07ffad3383aa381308b9041c61
Can I set the above configuration? (type 'yes' to accept): [输入 yes]
```

- 连接集群：

```
[root@theo cluster-conf]# redis-cli -p 7001 -c
```



### 8.3.2 集群管理

- 如果集群启动失败：等待节点加入

- 云服务器检查安全组是否放行redis实例端口，以及+10000的端口
- Linux防火墙是否放行redis服务（关闭防火墙）
- Linux状态（top）----更换云主机操作系统
- redis配置文件错误

- 创建集群：

```
[root@theo cluster-conf]# redis-cli --cluster create 47.96.11.185:7001 47.96.11.185:7002 47.96.11.185:7003 47.96.11.185:7004 47.96.11.185:7005 47.96.11.185:7006 --cluster-replicas 1
```

- 查看集群状态

```
[root@theo cluster-conf]# redis-cli --cluster info 47.96.11.185:7001
47.96.11.185:7001 (4678478a...) -> 2 keys | 5461 slots | 1 slaves.
47.96.11.185:7002 (e26eaf2a...) -> 0 keys | 5462 slots | 1 slaves.
47.96.11.185:7003 (5752eb20...) -> 1 keys | 5461 slots | 1 slaves.
[OK] 3 keys in 3 masters.
0.00 keys per slot on average.
```



- 平衡节点的数据槽数

```
[root@theo cluster-conf]# redis-cli --cluster rebalance 47.96.11.185:7001
>>> Performing Cluster Check (using node 47.96.11.185:7001)
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
*** No rebalancing needed! All nodes are within the 2.00% threshold.
```

- 迁移节点槽

```
[root@theo cluster-conf]# redis-cli --cluster info 47.96.11.185:7001
47.96.11.185:7001 (4678478aa...) -> 2 keys | 5459 slots | 1 slaves.
47.96.11.185:7002 (e26eaf2a...) -> 0 keys | 5465 slots | 1 slaves.
47.96.11.185:7003 (5752eb20...) -> 1 keys | 5460 slots | 1 slaves.
[OK] 3 keys in 3 masters.
0.00 keys per slot on average.
[root@theo cluster-conf]# redis-cli --cluster reshard 47.96.11.185:7001 启动迁移
>>> Performing Cluster Check (using node 47.96.11.185:7001)
M: 4678478aa66b6d37b23944cf7db0ac07298538a4 47.96.11.185:7001
  slots:[2-5460] (5459 slots) master
  1 additional replica(s)
S: 4315316860a5c3c97d7e83cbe66047c8e1f5fd72 47.96.11.185:7004
  slots: (0 slots) slave
  replicates 4678478aa66b6d37b23944cf7db0ac07298538a4
S: a5f35724529659ccdd13019309bca5e163cd4d32d 47.96.11.185:7006
  slots: (0 slots) slave
  replicates 5752eb202dc70d07ffad3383aa381308b9041c61
M: e26eaf2ae6c52e3477f3d6569428e02ccf63668e 47.96.11.185:7002
  slots:[0-1],[5461-10923] (5465 slots) master
  1 additional replica(s)
M: 5752eb202dc70d07ffad3383aa381308b9041c61 47.96.11.185:7003
  slots:[10924-16383] (5460 slots) master
  1 additional replica(s)
S: eb7173417f4392efba0710f6bfde7ef794e100ef 47.96.11.185:7005
  slots: (0 slots) slave
  replicates e26eaf2ae6c52e3477f3d6569428e02ccf63668e
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
How many slots do you want to move (from 1 to 16384)? 5459 设置迁移的槽的数量
What is the receiving node ID? e26eaf2ae6c52e3477f3d6569428e02ccf63668e 目标节点的id
Please enter all the source node IDs.
  Type 'all' to use all the nodes as source nodes for the hash slots.
  Type 'done' once you entered all the source nodes IDs.
Source node #1: 4678478aa66b6d37b23944cf7db0ac07298538a4
Source node #2: done
```

- 删除节点

```
[root@theo cluster-conf]# redis-cli --cluster del-node 47.96.11.185:7001
4678478aa66b6d37b23944cf7db0ac07298538a4

>>> Removing node 4678478aa66b6d37b23944cf7db0ac07298538a4 from cluster 47.96.11.185:7001
>>> Sending CLUSTER FORGET messages to the cluster...
>>> SHUTDOWN the node.

[root@theo cluster-conf]# redis-cli --cluster info 47.96.11.185:7002
47.96.11.185:7002 (e26eaf2a...) -> 1 keys | 8192 slots | 2 slaves.
47.96.11.185:7003 (5752eb20...) -> 2 keys | 8192 slots | 1 slaves.
[OK] 3 keys in 2 masters.
0.00 keys per slot on average.
```

- 添加节点

```
[root@theo cluster-conf]# redis-cli --cluster add-node 47.96.11.185:7007 47.96.11.185:7002
```

### 8.3.3 SpringBoot应用连接集群

- 添加依赖

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



- 配置集群节点

```
spring:
  redis:
    cluster:
      nodes: 47.96.11.185:7001,47.96.11.185:7002,47.96.11.185:7003
      max-redirects: 3
```



- 操作集群

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = RedisDemo3Application.class)
class RedisDemo3ApplicationTests {

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    @Test
    void contextLoads() {
        //stringRedisTemplate.boundValueOps("key1").set("value1");
        String s = stringRedisTemplate.boundValueOps("key1").get();
        System.out.println(s);
    }

}
```



## 九、Redis淘汰策略

Redis是基于内存结构进行数据缓存的，当内存资源消耗完毕，想要有新的数据缓存进来，必然要从Redis的内存结构中释放一些数据。如何进行数据的释放呢？----Redis的淘汰策略

Redis提供的8中淘汰策略

```
# volatile-lru -> 从设置了过期时间的数据中淘汰最久未使用的数据.  
# allkeys-lru -> 从所有数据中淘汰最久未使用的数据.  
# volatile-lfu -> 从设置了过期时间的数据中淘汰使用最少的数据.  
# allkeys-lfu -> 从所有数据中淘汰使用最少的数据.  
# volatile-random -> 从设置了过期时间的数据中随机淘汰一批数据.  
# allkeys-random -> 从所有数据中随机淘汰一批数据.  
# volatile-ttl -> 淘汰过期时间最短的数据.  
# noeviction -> 不淘汰任何数据, 当内存不够时直接抛出异常.
```

- 理解两个算法名词：

**LRU** 最久最近未使用

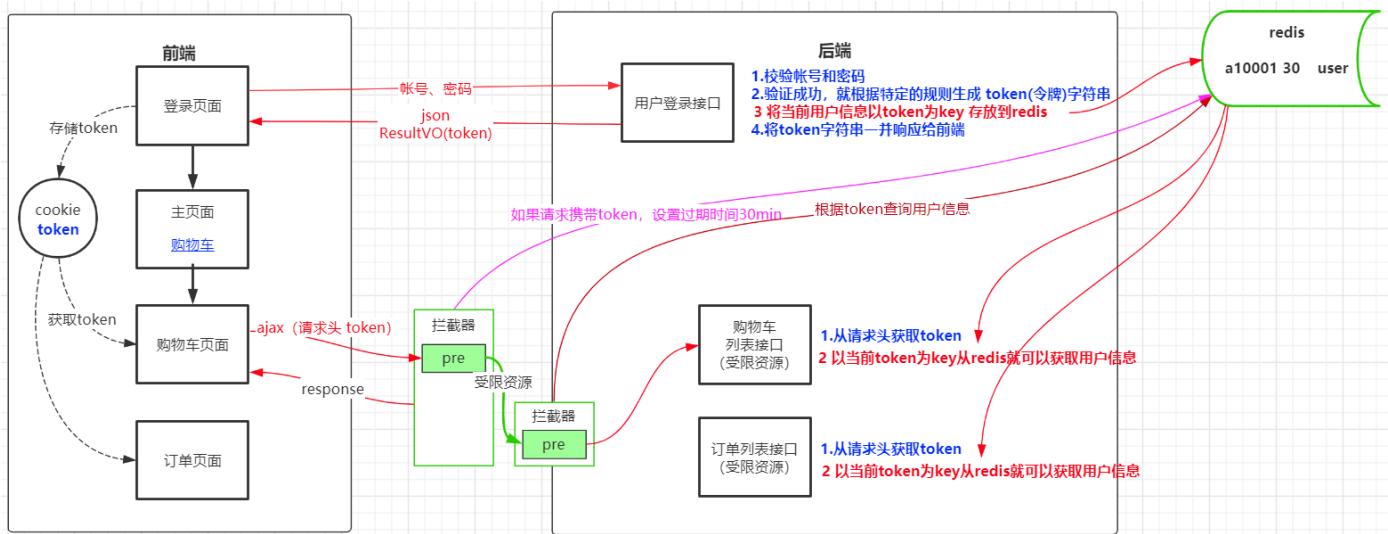
**LFU** 最近最少使用

## 十、Redis高频面试题

1. 在项目中redis的使用场景
  - 用于缓存首页数据
  - ...
2. Redis的持久化策略
3. Redis支持的数据类型
4. 如何保证redis的高可用
  - Redis支持持久化，同时开启rdb 和 aof，以保证数据的安全性（还是存在数据丢失风险的）
  - Redis支持主从配置，我们可通过配置哨兵，实现主备配置，保证可用性
  - Redis也支持集群，通过集群配置可以保证redis的高并发
5. 你刚才提到了redis集群，请问如何解决redis集群的脑裂问题？
6. redis中的数据可以设置过期时间，当数据过期之后有些key并没有及时清除，请问如何处理？

## 十一、使用Redis实现分布式会话

### 11.1 流程分析



## 11.2 在锋迷商城中使用redis实现分布式会话

### 11.2.1 修改登录接口

- 当登录成功以token为key将用户信息保存到redis

```
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UsersMapper usersMapper;
    @Autowired
    private StringRedisTemplate stringRedisTemplate;
    private ObjectMapper objectMapper = new ObjectMapper();

    @Transactional
    public ResultVO userResgit(String name, String pwd) {
        //...
    }

    @Override
    public ResultVO checkLogin(String name, String pwd) {
        Example example = new Example(Users.class);
        Example.Criteria criteria = example.createCriteria();
        criteria.andEqualTo("username", name);
        List<Users> users = usersMapper.selectByExample(example);

        if(users.size() == 0){
            return new ResultVO(ResStatus.NO, "登录失败, 用户名不存在! ", null);
        }else{
            String md5Pwd = MD5Utils.md5(pwd);
            if(md5Pwd.equals(users.get(0).getPassword())){
                //如果登录验证成功, 则需要生成令牌token (token就是按照特定规则生成的字符串)
                //使用jwt规则生成token字符串
                JwtBuilder builder = Jwts.builder();
                HashMap<String, Object> map = new HashMap<>();
                map.put("username", name);
                map.put("password", pwd);
                map.put("role", "user");
                map.put("exp", System.currentTimeMillis() + 30 * 60 * 1000);
                String token = builder.setClaims(map).compact();
                stringRedisTemplate.opsForValue().set("token", token);
                return new ResultVO(ResStatus.OK, "登录成功!", token);
            }
        }
    }
}
```

```
        map.put("key1", "value1");
        map.put("key2", "value2");

        String token = builder.setSubject(name)
            .setIssuedAt(new Date())
            .setId(users.get(0).getUserId() + "")
            .setClaims(map)
            .setExpiration(new Date(System.currentTimeMillis() + 24*60*60*1000))
            .signWith(SignatureAlgorithm.HS256, "QIANfeng6666")
            .compact();

        //当用户登录成功之后，以token为key 将用户信息保存到reids
        try {
            String userInfo = objectMapper.writeValueAsString(users.get(0));
            stringRedisTemplate.boundValueOps(token).set(userInfo, 30, TimeUnit.MINUTES);
        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
        return new ResultVO(ResStatus.OK, token, users.get(0));
    }else{
        return new ResultVO(ResStatus.NO, "登录失败，密码错误! ", null);
    }
}
```

11.2.2 在需要只用用户信息的位置，直接根据token从redis查询

```
@PostMapping("/add")
public ResultVO addShoppingCart(@RequestBody ShoppingCart cart,@Re
token) throws JsonProcessingException {
    ResultVO resultVO = shoppingCartService.addShoppingCart(cart);
    String s = stringRedisTemplate.boundValueOps(token).get();
    Users users = objectMapper.readValue(s, Users.class);
    System.out.println(users);
    return resultVO;
}
```



### 11.2.3 修改受限资源拦截器

```
@Component
public class CheckTokenInterceptor implements HandlerInterceptor {

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object
handler) throws Exception {
        String method = request.getMethod();
        if("OPTIONS".equalsIgnoreCase(method)){
            response.setStatus(204);
            return true;
        }
        String token = request.getHeader("token");
        if(token == null || token.length() < 1){
            response.setStatus(401);
            return false;
        }
        //从redis中取出token
        String value = stringRedisTemplate.opsForValue().get(token);
        if(value == null || !value.equals(token)){
            response.setStatus(401);
            return false;
        }
        return true;
    }
}
```

```

        return true;
    }
    String token = request.getHeader("token");
    if(token == null){
        ResultVO resultVO = new ResultVO(ResStatus.LOGIN_FAIL_NOT, "请先登录!", null);
        doResponse(response, resultVO);
    }else{
        //根据token从redis中获取用户信息
        String s = stringRedisTemplate.boundValueOps(token).get();
        if(s == null){
            //如果用户信息为空，表示用户未登录或者距离上一次访问超过30分钟
            ResultVO resultVO = new ResultVO(ResStatus.LOGIN_FAIL_NOT, "请先登录!", null);
            doResponse(response, resultVO);
        }else{
            //如果不为空，表示用户登录成功，续命
            stringRedisTemplate.boundValueOps(token).expire(30, TimeUnit.MINUTES);
            return true;
        }
    }
    return false;
}

private void doResponse(HttpServletRequest response, ResultVO resultVO) throws IOException
{
    response.setContentType("application/json");
    response.setCharacterEncoding("utf-8");
    PrintWriter out = response.getWriter();
    String s = new ObjectMapper().writeValueAsString(resultVO);
    out.print(s);
    out.flush();
    out.close();
}
}

```

#### 11.2.4 创建非受限资源拦截器

即使访问的是非受限资源，但是如果已经登录，只要与服务器有交互也要续命



```

@Component
public class SetTimeInterceptor implements HandlerInterceptor {

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object
handler) {
        String token = request.getHeader("token");
        if(token != null){
            String s = stringRedisTemplate.boundValueOps(token).get();

```

```
        if(s != null){
            stringRedisTemplate.boundValueOps(token).expire(30, TimeUnit.MINUTES);
        }
    }
    return true;
}
}
```

- 配置拦截器：

```
@Configuration
public class InterceptorConfig implements WebMvcConfigurer {

    @Autowired
    private CheckTokenInterceptor checkTokenInterceptor;
    @Autowired
    private SetTimeInterceptor setTimeInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(checkTokenInterceptor)
            .addPathPatterns("/shopcart/**")
            .addPathPatterns("/orders/**")
            .addPathPatterns("/useraddr/**")
            .addPathPatterns("/user/check");

        registry.addInterceptor(setTimeInterceptor).addPathPatterns("/**");
    }
}
```



前端访问注意事项： 只要前端有token，对接口的访问就要携带token