

好友关注

一、关注和取消关注

1、分析

在探店图文的详情页面中，可以关注发布笔记的作者：



功能对应的接口地址：

尝试关注用户

1 | put请求 `http://localhost:8080/api/follow/{id}/true`

是否关注用户

1 | get请求
`http://localhost:8080/api/follow/or/not/{id}`

对应数据库表：tb_follow

实现思路：

需求：基于该表数据结构，实现两个接口：

- 关注和取关接口

- 判断是否关注的接口

关注是User之间的关系，是博主与粉丝的关系，数据库中有一张tb_follow表来标示：

```
1 CREATE TABLE `tb_follow` (  
2   `id` bigint NOT NULL AUTO_INCREMENT COMMENT '主  
   键',  
3   `user_id` bigint unsigned NOT NULL COMMENT '用户  
   id',  
4   `follow_user_id` bigint unsigned NOT NULL  
   COMMENT '关联的用户id',  
5   `create_time` timestamp NOT NULL DEFAULT  
   CURRENT_TIMESTAMP COMMENT '创建时间',  
6   PRIMARY KEY (`id`) USING BTREE  
7 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4  
   COLLATE=utf8mb4_general_ci ROW_FORMAT=COMPACT;
```

注意: 这里需要把主键修改为自增长，简化开发。

2、代码实现

Controloller层

FollowController类实现接口

```

1  @Resource
2      private IFollowService followService;
3
4      //关注和取消关注
5      @PutMapping("/{id}/{isFollow}")
6      public Result follow(@PathVariable("id") Long
followUserId, @PathVariable("isFollow") Boolean
isFollow) {
7          return followService.follow(followUserId,
isFollow);
8      }
9      //是否关注
10     @GetMapping("/or/not/{id}")
11     public Result isFollow(@PathVariable("id")
Long followUserId) {
12         return
followService.isFollow(followUserId);
13     }

```

FollowServiceImpl类

```

1  @Override
2      public Result follow(Long followUserId,
Boolean isFollow) {
3      // 1.获取登录用户
4      Long userId =
UserHolder.getUser().getId();
5      // 1.判断到底是关注还是取关
6      if (isFollow) {
7          // 2.关注，新增数据
8          Follow follow = new Follow();
9          follow.setUserId(userId);
10         follow.setFollowUserId(followUserId);
11         save(follow);
12
13         } else {

```

```

14         // 3.取关, 删除 delete from tb_follow
    where user_id = ? and follow_user_id = ?
15         remove(new QueryWrapper<Follow>()
16             .eq("user_id",
    userId).eq("follow_user_id", followUserId));
17
18     }
19     return Result.ok();
20 }
21
22 @Override
23 public Result isFollow(Long followUserId) {
24     // 1.获取登录用户
25     Long userId =
    UserHolder.getUser().getId();
26     // 2.查询是否关注 select count(*) from
    tb_follow where user_id = ? and follow_user_id =
    ?
27     Integer count = query().eq("user_id",
    userId).eq("follow_user_id",
    followUserId).count();
28     // 3.判断
29     return Result.ok(count > 0);
30 }

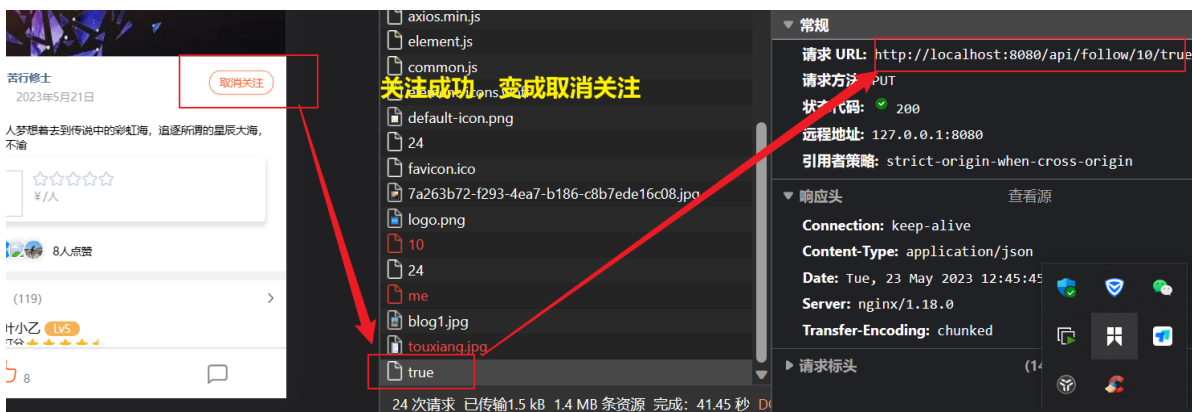
```

3、结果：

查询是否关注



关注成功

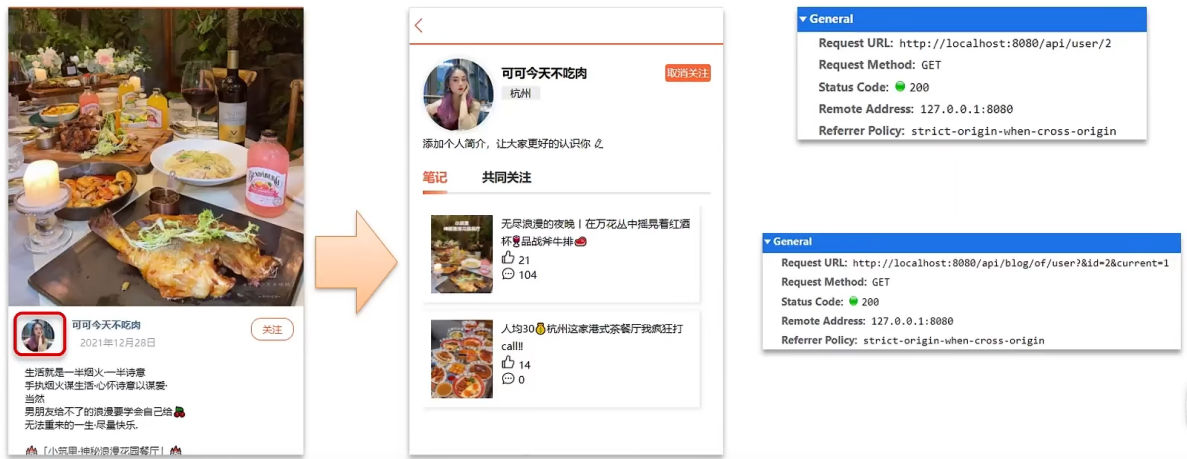


取消关注



二、共同关注

如果我和对方关注了相同的人就会出现共同关注好友显示



想要去看共同关注的好友，需要首先进入到这个页面，这个页面会发起两个请求

1、查询用户

1) 去查询用户的详情

根据id查询用户UserController

```

1  @GetMapping("/{id}")
2  public Result queryUserById(@PathVariable("id")
   Long userId){
3      // 查询详情
4      User user = userService.getById(userId);
5      if (user == null) {
6          return Result.ok();
7      }
8      UserDTO userDTO =
        BeanUtil.copyProperties(user, UserDTO.class);
9      // 返回
10     return Result.ok(userDTO);
11 }

```

2) 去查询用户的笔记

BlogController 根据id查询博主的探店笔记

```

1 @GetMapping("/of/user")
2 public Result queryBlogByUserId(
3     @RequestParam(value = "current",
4     defaultValue = "1") Integer current,
5     @RequestParam("id") Long id) {
6     // 根据用户查询
7     Page<Blog> page = blogService.query()
8         .eq("user_id", id).page(new Page<>
9         (current, SystemConstants.MAX_PAGE_SIZE));
10    // 获取当前页数据
11    List<Blog> records = page.getRecords();
12    return Result.ok(records);
13 }

```

以上两个功能和共同关注没有什么关系，大家可以自行将笔记中的代码拷贝到idea中就可以实现这两个功能了，我们的重点在于共同关注功能。

2、实现共同关注功能

1) 思路分析

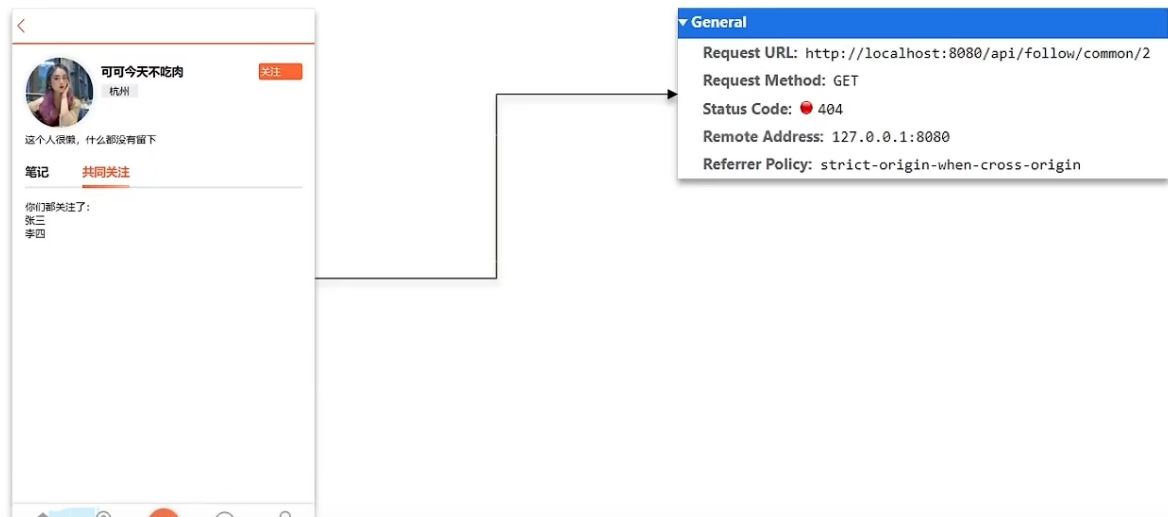
对应接口：

```
1 http://localhost:8080/api/follow/common/{userId}
```

接下来我们来看看共同关注如何实现：

需求：利用Redis中恰当的数据结构，实现共同关注功能。在博主个人页面展示出当前用户与博主的共同关注呢。

当然是使用我们之前学习过的set集合咯，在set集合中，有交集并集补集的api，我们可以把两人的关注的人分别放入到一个set集合中，然后再通过api去查看这两个set集合中的交集数据。



我们先来改造当前的关注列表

改造原因是因为我们需要在用户关注了某位用户后，需要将数据放入到set集合中，方便后续进行共同关注，同时当取消关注时，也需要从set集合中进行删除

FollowServiceImpl

2) 修改follow方法

```
1 @Override
2     public Result follow(Long followUserId,
3     boolean isFollow) {
4         // 1. 获取登录用户
5         Long userId =
6         UserHolder.getUser().getId();
7         String key = "follows:" + userId;
8         // 1. 判断到底是关注还是取关
9         if (isFollow) {
10             // 2. 关注，新增数据
11             Follow follow = new Follow();
12             follow.setUserId(userId);
13             follow.setFollowUserId(followUserId);
14             boolean isSuccess = save(follow);
15             if (isSuccess) {
16                 // 把关注用户的id，放入redis的set集合
17                 sadd userId followerUserId
18             }
19         }
20     }
```



```

15     stringRedisTemplate.opsForSet().add(key,
16         followUserId.toString());
17     } else {
18         // 3.取关，删除 delete from tb_follow
19         // where user_id = ? and follow_user_id = ?
20         boolean isSuccess = remove(new
21             QueryWrapper<Follow>()
22                 .eq("user_id",
23                     followUserId).eq("follow_user_id", followUserId));
24         if (isSuccess) {
25             // 把关注用户的id从Redis集合中移除
26
27             stringRedisTemplate.opsForSet().remove(key,
28                 followUserId.toString());
29         }
30     }
31     return Result.ok();
32 }

```

3) 具体的关注代码:

FollowController

```

1 //共同关注
2 @GetMapping("/common/{id}")
3 public Result
4 followCommons(@PathVariable("id") Long id){
5     return followService.followCommons(id);
6 }

```

4) 实现followCommons方法

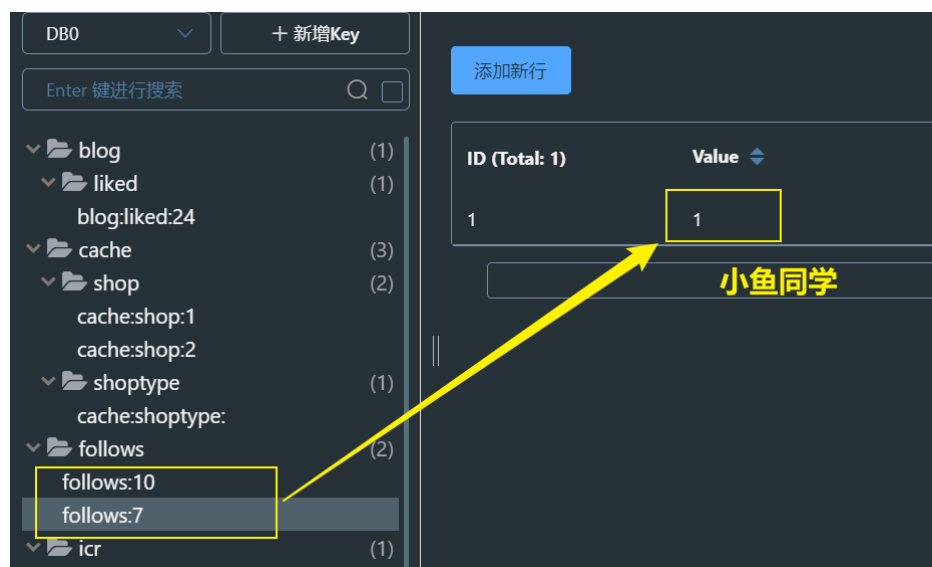
实现接口和接口方法

FollowServiceImpl

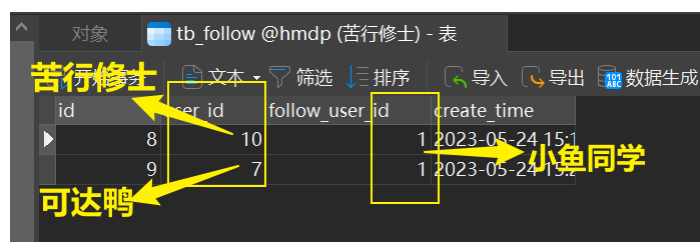
```
1 @Resource
2     private IFollowService userService;
3
4 @Override
5 public Result followCommons(Long id) {
6     // 1.获取当前用户
7     Long userId = UserHolder.getUser().getId();
8     String key = "follows:" + userId;
9     // 2.求交集
10    String key2 = "follows:" + id;
11    Set<String> intersect =
stringRedisTemplate.opsForSet().intersect(key,
key2);
12    if (intersect == null || intersect.isEmpty())
{
13        // 无交集
14        return
Result.ok(Collections.emptyList());
15    }
16    // 3.解析id集合
17    List<Long> ids =
intersect.stream().map(Long::valueOf).collect(Col
lectors.toList());
18    // 4.查询用户
19    List<UserDTO> users =
userService.listByIds(ids)
20        .stream()
21        .map(user ->
BeanUtil.copyProperties(user, UserDTO.class))
22        .collect(Collectors.toList());
23    return Result.ok(users);
24 }
```

5) 结果

从缓存中可以看到苦行修士和可达鸭共同关注了小鱼同学



数据库中也没问题



但是页面似乎还是有问题，不知道发的什么神经！！！ 🤔 🤔 🤔

三、关注推送

1、Feed流实现方案

当我们关注了用户后，这个用户发了动态，那么我们应该把这些数据推送给用户，这个需求，其实我们又把他叫做Feed流，关注推送也叫做Feed流，直译为投喂。为用户持续的提供“沉浸式”的体验，通过无限下拉刷新获取新的信息。

对于传统的模式的内容解锁：我们是需要用户去通过搜索引擎或者是其他方式去解锁想要看的内容



对于新型的Feed流的效果：不需要我们用户再去推送信息，而是系统分析用户到底想要什么，然后直接把内容推送给用户，从而使用户能够更加的节约时间，不用主动去寻找。



Feed流的实现有两种模式：

Feed流产品有两种常见模式：

Timeline：不做内容筛选，简单的按照内容发布时间排序，常用于好友或关注。例如朋友圈

- 优点：信息全面，不会有缺失。并且实现也相对简单
- 缺点：信息噪音较多，用户不一定感兴趣，内容获取效率低

智能排序：利用智能算法屏蔽掉违规的、用户不感兴趣的内容。推送用户感兴趣信息来吸引用户

- 优点：投喂用户感兴趣信息，用户粘度很高，容易沉迷
- 缺点：如果算法不精准，可能起到反作用

我们本次针对好友的操作，采用的就是Timeline的方式，只需要拿到我们关注用户的信息，然后按照时间排序即可

，因此采用Timeline的模式。该模式的实现方案有三种：

- 拉模式
- 推模式
- 推拉结合

拉模式：也叫做读扩散

该模式的核心含义就是：当张三和李四和王五发了消息后，都会保存在自己的邮箱中，假设赵六要读取信息，那么他会从读取他自己的收件箱，此时系统会从他关注的人群中，把他关注人的信息全部都进行拉取，然后在进行排序

优点：比较节约空间，因为赵六在读信息时，并没有重复读取，而且读取完之后可以把他的收件箱进行清楚。

缺点：比较延迟，当用户读取数据时才去关注的人里边去读取数据，假设用户关注了大量的用户，那么此时就会拉取海量的内容，对服务器压力巨大。

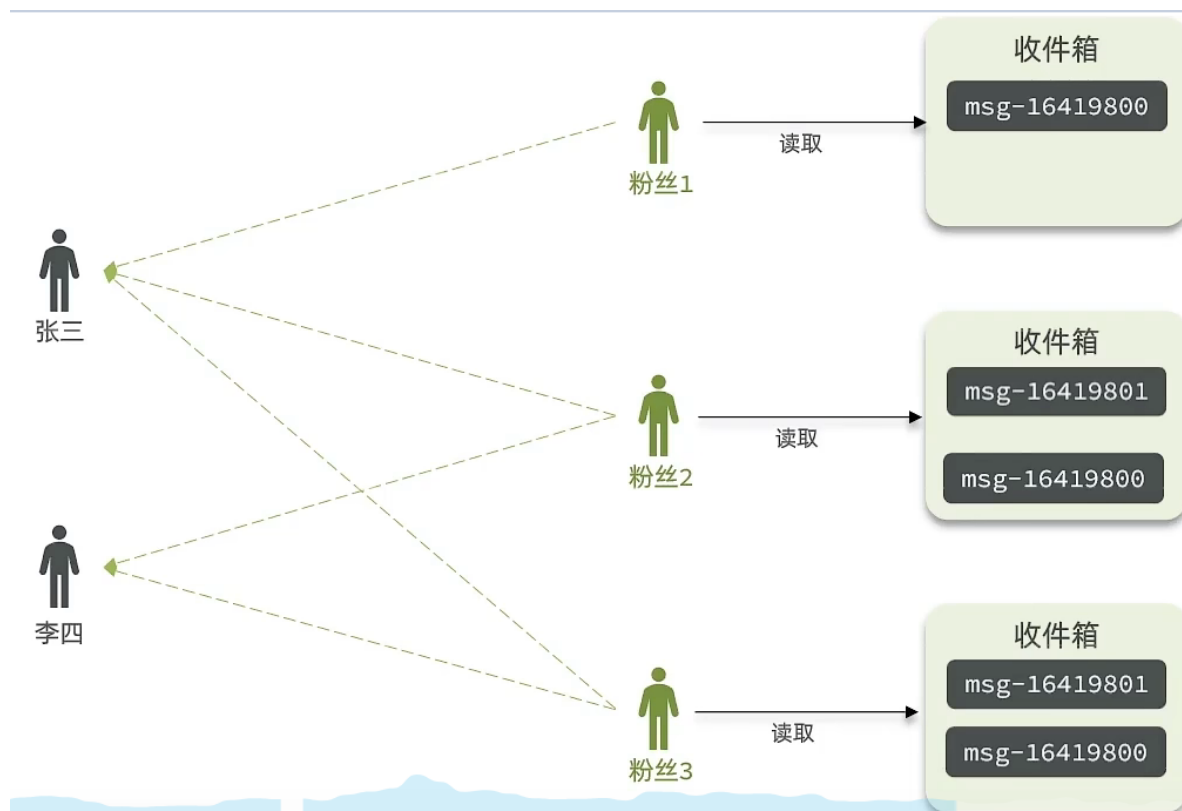


推模式：也叫做写扩散。

推模式是没有写邮箱的，当张三写了一个内容，此时会主动的把张三写的内容发送到他的粉丝收件箱中去，假设此时李四再来读取，就不用再去临时拉取了

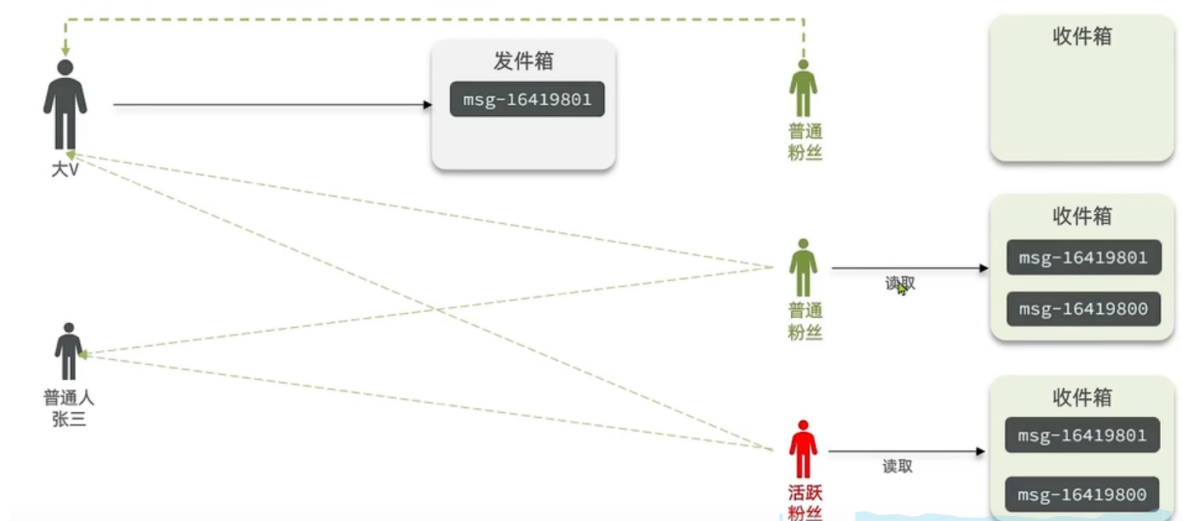
优点：时效快，不用临时拉取

缺点：内存压力大，假设一个大V写信息，很多人关注他，就会写很多分数据到粉丝那边去



推拉结合模式：也叫做读写混合，兼具推和拉两种模式的优点。

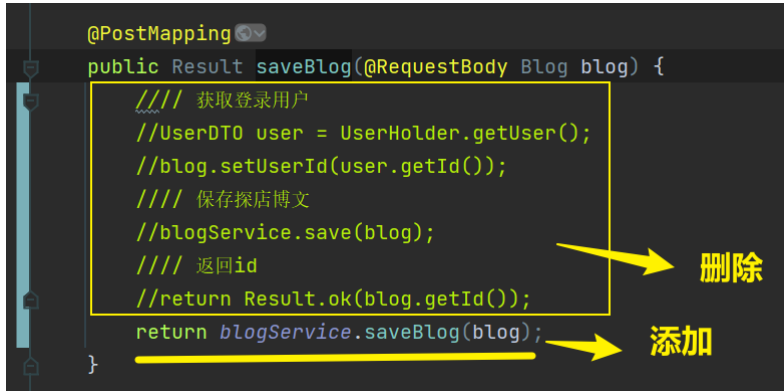
推拉模式是一个折中的方案，站在发件人这一段，如果是个普通的人，那么我们采用写扩散的方式，直接把数据写入到他的粉丝中去，因为普通的人他的粉丝关注量比较小，所以这样做没有压力，如果是大V，那么他是直接将数据先写入到一份到发件箱里边去，然后再直接写一份到活跃粉丝收件箱里边去，现在站在收件人这端来看，如果是活跃粉丝，那么大V和普通的人发的都会直接写入到自己收件箱里边来，而如果是普通的粉丝，由于他们上线不是很频繁，所以等他们上线时，再从发件箱里边去拉信息。



2、推送到粉丝收件箱

1) 修改BlogController

把saveBlog方法的逻辑转到BlogServiceImpl中



```
@PostMapping
public Result saveBlog(@RequestBody Blog blog) {
    /// 获取登录用户
    //UserDTO user = UserHolder.getUser();
    //blog.setUserId(user.getId());
    /// 保存探店博文
    //blogService.save(blog);
    /// 返回id
    //return Result.ok(blog.getId());
    return blogService.saveBlog(blog);
}
```

创建方法saveBlog

2) service层实现saveBlog方法

BlogServiceImpl

```
1  @Override
2      public Result saveBlog(Blog blog) {
3          // 1、获取登录用户
4          UserDTO user = UserHolder.getUser();
5          blog.setUserId(user.getId());
6          // 2、保存探店博文
7          boolean isSuccess = save(blog);
8          if (isSuccess) {
9              return Result.fail("新增笔记失
10             败!!!");
11             }
12             //3、查询笔记作者的所以粉丝 select * from
13             tb_follow where follow_user_id=?
14             List<Follow> follows =
15             followService.query()
16                 .eq("follow_user_id",
17                     user.getId()).list();
18             //4、推送笔记id给所有的粉丝
```

```

15         for (!Follow follow : follows) {
16             //4.1、获取粉丝id
17             Long userId = follow.getUserId();
18             //4.2、推送
19             String key = "feed:" + userId;
20             stringRedisTemplate.opsForZSet()
21
22             .add(key, blog.getId().toString(), System.currentTimeMillis());
23         }
24         // 3、返回id
25         return Result.ok(blog.getId());
26     }

```

3) 测试结果

登录小鱼同学账号先新增笔记，小鱼同学新增的笔记会出现在关注她的粉丝可达鸭中

所以缓存中会出现小鱼同学的信息在id=可达鸭

①登录账户新增笔记



数据库新增成功

3	10	1	杭州周末好去/imgs/blogs杭州周末好去处 50就可以骑马啦	1
4	10	1	杭州周末好去/imgs/blogs杭州周末好去处 50就可以骑马啦	1
23	10	10	知识海洋 /imgs/blogs亏本大买卖	1
24	10	10	我们的约定 /imgs/blogs曾有一个梦想着去到传说中的彩虹海, 过	8
25	10	1	亚洲帝王火锅/imgs/blogs全场八折优惠, 走过路过千万不要错过!!	0
27	10	1	豪华大套餐 /imgs/blogs新老用户一律八折优惠, 吃不完还可打包	0

tb_blog表

新增笔记id=27

②查看缓存

数据无误

DB0	+ 新增Key	添加新行
Enter 键进行搜索		
cache type:		
feed	(2)	
feed:10		
feed:7		
follows	(2)	
follows:10		

ID (Total: 1)	Score	Member
1	1684935520028	27

对应笔记id=27

3、实现滚动分页查询收件箱

需求：在个人主页的“关注”卡片中，查询并展示推送的Blog信息：

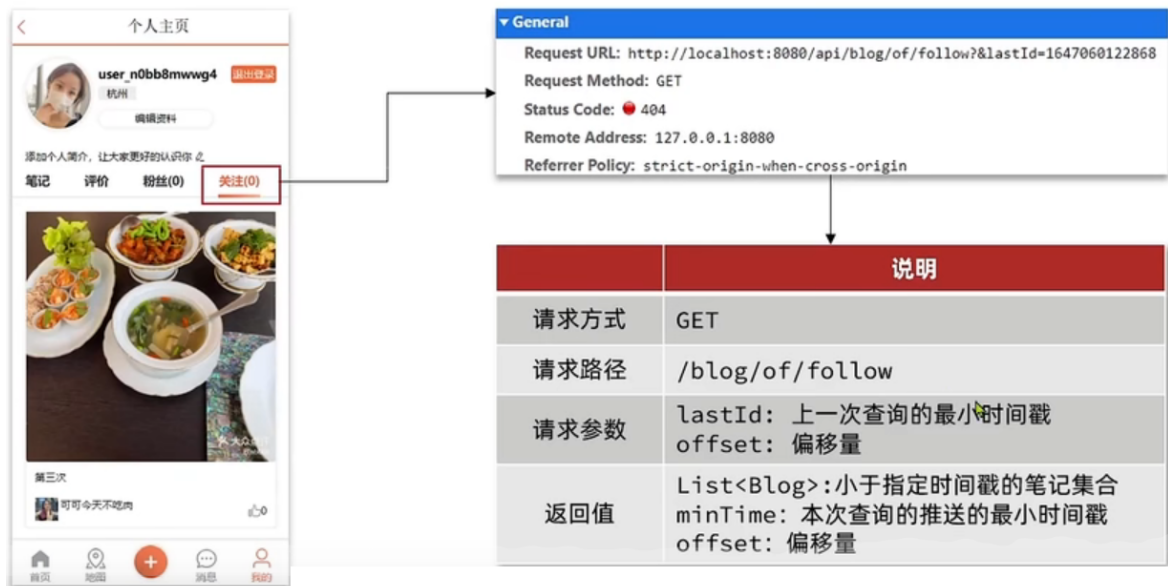
1) 思路

具体操作如下：

- 1、每次查询完成后，我们要分析出查询出数据的最小时间戳，这个值会作为下一次查询的条件
- 2、我们需要找到与上一次查询相同的查询个数作为偏移量，下次查询时，跳过这些查询过的数据，拿到我们需要的数据

综上：我们的请求参数中就需要携带 lastId：上一次查询的最小时间戳 和偏移量这两个参数。

这两个参数第一次会由前端来指定，以后的查询就根据后台结果作为条件，再次传递到后台。



2) 原理

3) 代码实现

① 定义出来具体的返回值实体类

```
1 @Data
2 public class ScrollResult {
3     private List<?> list;
4     private Long minTime;
5     private Integer offset;
6 }
```

② BlogController

注意: RequestParam 表示接受url地址栏传参的注解, 当方法上参数的名称和url地址栏不相同, 可以通过RequestParam 来进行指定

创建方法queryBlogOfFollow

```
1 @GetMapping("/of/follow")
2 public Result queryBlogOfFollow(
3     @RequestParam("lastId") Long max,
4     @RequestParam(value = "offset", defaultValue =
5         "0") Integer offset){
6     return blogService.queryBlogOfFollow(max,
7         offset);
8 }
```

③ BlogServiceImpl

实现逻辑：

- 1.获取当前用户
- 2.查询收件箱 ZREVRANGEBYSCORE key Max Min LIMIT
offset count
- 3.非空判断
- 4.解析数据：blogId、minTime（时间戳）、offset
 - 4.1.获取id
 - 4.2.获取分数(时间戳)
- 5.根据id查询blog
 - 5.1.查询blog有关的用户
 - 5.2.查询blog是否被点赞
- 6.封装并返回

实现方法queryBlogOfFollow

```
1 @Override
```

```
2 public Result queryBlogOfFollow(Long max, Integer
offset) {
3     // 1.获取当前用户
4     Long userId = UserHolder.getUser().getId();
5     // 2.查询收件箱 ZREVRANGEBYSCORE key Max Min
LIMIT offset count
6     String key = FEED_KEY + userId;
7     Set<ZSetOperations.TypedTuple<String>>
typedTuples = stringRedisTemplate.opsForZSet()
8         .reverseRangeByScoreWithScores(key, 0,
max, offset, 2);
9     // 3.非空判断
10    if (typedTuples == null ||
typedTuples.isEmpty()) {
11        return Result.ok();
12    }
13    // 4.解析数据: blogId、minTime (时间戳)、offset
14    List<Long> ids = new ArrayList<>
(typedTuples.size());
15    long minTime = 0; // 2
16    int os = 1; // 2
17    for (ZSetOperations.TypedTuple<String> tuple
: typedTuples) { // 5 4 4 2 2
18        // 4.1.获取id
19        ids.add(Long.valueOf(tuple.getValue()));
20        // 4.2.获取分数(时间戳)
21        long time = tuple.getScore().longValue();
22        if(time == minTime){
23            os++;
24        }else{
25            minTime = time;
26            os = 1;
27        }
28    }
29    os = minTime == max ? os : os + offset;
30    // 5.根据id查询blog
31    String idStr = StrUtil.join(",", ids);
```

```
32     List<Blog> blogs = query().in("id",
33     ids).last("ORDER BY FIELD(id," + idStr +
34     ")").list();
35
36     for (Blog blog : blogs) {
37         // 5.1. 查询blog有关的用户
38         queryBlogUser(blog);
39         // 5.2. 查询blog是否被点赞
40         isBlogLiked(blog);
41     }
42
43     // 6. 封装并返回
44     ScrollResult r = new ScrollResult();
45     r.setList(blogs);
46     r.setOffset(os);
47     r.setMinTime(minTime);
48     return Result.ok(r);
49 }
```

4) 页面效果