# 一、缓存问题

## 1、被动缓存

示例：

|  |
| --- |
| **public class** UserActiveFlagLocalCacheImpl **implements** UserActiveFlagLocalCache {  **private static final** String ***KEY\_USER\_FLAG*** = **"user:flag"**;  **private** Cache<String, Set<String>> **localCache**;   @Inject  **private** UserFlagCache **userFlagCache**;   @Inject  **public** UserActiveFlagLocalCacheImpl() {  **localCache** = CacheBuilder.*newBuilder*()  .maximumSize(10000)  .concurrencyLevel(4)  .expireAfterWrite(60, TimeUnit.***MINUTES***)  .build();  }   @Override  **public** Set<String> getUserIds() {  **try** {  Set<String> userids = **localCache**.getIfPresent(***KEY\_USER\_FLAG***);  **if** (userids == **null** || userids.size() == 0) {  **synchronized** (UserActiveFlagLocalCacheImpl.**class**) {  **if** (userids == **null** || userids.size() == 0) {  Set<String> smembers = **userFlagCache**.smembers();  **if** (smembers == **null**) {  smembers = **new** HashSet<>();  }  userids = smembers;  **localCache**.put(***KEY\_USER\_FLAG***, smembers);  }  }  }**return** userids;  }**catch** (Exception e){  }  **return new** HashSet<>();  } } |

1. **localCache** 是一个可以自动过期的缓存对象。

在外层使用getUserIds 方法获取用户id。其中先从本地缓存**localCache**中获取。如果本地缓存没有，则再从数据源（此处是**userFlagCache**）获取数据到本地，并将数据缓存到**localCache中**。

## 2、主动缓存（后台压力大，采用主动更新缓存的方式。）

示例：

|  |
| --- |
| */\*\* 从原创用户总粉榜+原创用户增粉榜+原创用户内容榜  \*/* **public class** OriginRankCacheImpl **implements** OriginRankCache {   **private static** Logger *LOGGER* = LoggerFactory.*getLogger*(OriginRankCacheImpl.**class**);   **private static final int *CACHE\_VALID\_TIME*** = 5;  */\*\* 重试次数 \*/* **private static final int *RETRY\_TIMES*** = 3;  **private static long** *updateTime* = 0;  */\*\* \*一小时对应的毫秒数 \*/* **private static long** *UPDATETIME\_GAP* = 60 \* 60 \* 1000;   **private** PuriService.Iface **puriService**;   **private** RankService **rankService**;   */\*\*\* 缓存用户榜的key和用户列表 \*/* **private static** Map<String, List<UserRankingDetail>> *userCacheMap*;   **private static** ScheduledExecutorService *executorService* = Executors.*newScheduledThreadPool*(1);   */\*\* 从BeLab获取所有榜单数据，加入本地缓存。返回上榜时间 \*/* @Inject  **public OriginRankCacheImpl**(PuriService.Iface puriService, RankService rankService) **throws** TException {  **this**.**puriService** = puriService;  **this**.**rankService** = rankService;  **initLocalCache();**  }  **private void** initLocalCache() {  *userCacheMap* = Maps.*newHashMap*();  **boolean** succ = **false**;   ***//重试RETRY\_TIMES次*****for** (**int** i = 0; i < ***RETRY\_TIMES***; i++) {  succ = getLocalCache();  **if** (succ) {  **break**;  }  }    ***//如果从远端获取失败，给locolCache 填充emptyList，并报警*** **if** (!succ) {  initEmptyCache();  **for** (**int** i = 0; i < 200; i++) {  FalconHelper.*countBeeLabRank*();  *LOGGER*.error(**"initLocalCache fail"**);  }  }**else** {  *LOGGER*.warn(**"initLocalCache success"**);  }    ***//启动线程池，定时主动拉取数据  executorService*.scheduleWithFixedDelay(new Runnable() {  @Override  public void run() {  *//距离上次成功更新时间大于 60 分钟，则请求* if (System.*currentTimeMillis*() - *updateTime* >= *UPDATETIME\_GAP*) {  boolean localCache = getLocalCache();  if (!localCache) {  for (int i = 0; i < 200; i++) {  FalconHelper.*countBeeLabRank*();  *LOGGER*.error("initLocalCache schedule fail");  }  }else {  *LOGGER*.warn("initLocalCache schedule success");  }  }   }  }, *CACHE\_VALID\_TIME*, *CACHE\_VALID\_TIME*, TimeUnit.*MINUTES*);  }**   */\*\*  \* 从远端拉取数据。成功返回true;失败或异常返回false  \* \*/* **private boolean** getLocalCache() {   OriginRankJsonResult originRankJsonResult = **null**;  originRankJsonResult = **puriService**.getOriginRankContent(FalconLogService.*getTraceID*());  **if** (originRankJsonResult.getCode() != 0 || StringUtils.*isBlank*(originRankJsonResult.getOriginRankData())) {  **return false**;  }  */\*把从远端拉取的数据放到userCacheMap缓存中 \*/*  CurrentRankingListVO currentRankingListVO = GsonUtil.*parse*(originRankJsonResult.getOriginRankData(), **new** TypeToken<CurrentRankingListVO>() {});  *userCacheMap*.put(key, sortUserRankingDetail(userList));  **return true**;  }   */\*  \* 生成空列表  \* \*/* **public void** initEmptyCache() {  **for** (RankCategoryEnum categoryEnum : RankCategoryEnum.*values*()) {  **for** (RankEnum rankEnum : RankEnum.*values*()) {  String key = RankUtils.*getRankNameCodeAndCategoryCodeKey*(rankEnum.getCode(), categoryEnum.getCode());  *//如果是视频榜* **if** (rankEnum == RankEnum.***RANKING\_OF\_VIDEO\_NEW\_LIKE***) {  *videoDetailCacheMap*.put(key, **new** ArrayList<VideoRankingDetail>());  } **else** {  *userCacheMap*.put(key, **new** ArrayList<UserRankingDetail>());  }  }  }  *//缓存更新时间  updateTime* = 0;  } |

**主动缓存的关键点**：

（1）主动缓存的数据一般不能为空。即缓存中的数据禁止为空。为了保证缓存一定有数据，在缓存对象初始化的时候，一般定义循环从数据源获取数据并加入到本地缓存中。

|  |
| --- |
| *//重试RETRY\_TIMES次* **for** (**int** i = 0; i < ***RETRY\_TIMES***; i++) {  succ = getLocalCache();  **if** (succ) {  **break**;  } } |

一旦有一次获取成功，那么就将远端数据put到本地缓存。

如果达到上限重复次数还没从数据源拉取到数据，那么就为本地缓存初始化内容为空的对象并报警。

|  |
| --- |
| *//如果从远端获取失败，给locolCache 填充emptyList，并报警* **if** (!succ) {  initEmptyCache();  **for** (**int** i = 0; i < 200; i++) {  FalconHelper.*countBeeLabRank*();  *LOGGER*.error(**"initLocalCache fail"**);  } }**else** {  *LOGGER*.warn(**"initLocalCache success"**); } |

（2）本地缓存禁止为null。如果本地缓存没有数据，应为本地缓存添加空对象，而不是直接将null返回。

## 3、从本地缓存对象到服务返回给客户端对象之间的转化关系

（1）puri-api项目的分层概念：

|  |  |  |
| --- | --- | --- |
| 层级 | Restlet | 数据模型命名 |
| 接口层 | SR（Controller） | VO |
| 业务处理层 | Service | DTO |
| 数据层 | Mapper | DO: 数据库表记录模型  CO：缓存结构模型  MO：Mango 记录模型 |

如果在业务处理时，为了实现分层隔离，每一层使用各层的数据模型。且上层不能直接将下层的模型数据直接返回。这是因为：

比如（数据层）本地缓存对象的model为

|  |
| --- |
| public class UserRankingDetail {  private int ranking; //排名  private String userId; //用户id  private int followStatus; //关注状态 0：未关注 1：已关注, -1：自己  } |

（接口层）客户端需要返回的对象model为：

|  |
| --- |
| public class UserRankingDetailVO {  private int ranking; //排名  private String userId; //用户id  private int followStatus; //关注状态 0：未关注 1：已关注, -1：自己  } |

其中：followStatus 是需要时时获取设置的。

如果直接将缓存对象UserRankingDetail返回给客户端，会存在两个问题：

1. 假设followStatus状态是时时获取的。上一次用户A读取缓存后，由于存在关注关系，将缓存中的某个用户的followStatus 设置成1。下一次未登录用户B读取缓存，不存在关注关系，此时应该将followStatus 设置成0。即每次读取缓存都需要清空状态标识。
2. 存在并发问题。由于followStatus状态每次请求都需要修改，当qps非常高时，多个请求同时访问用一个本地缓存，可能存在写写、写读、读写竞争问题。

正确处理方式：

定义（数据层）本地缓存对象的model为

|  |
| --- |
| public class UserRankingDetail {  private int ranking; //排名  private String userId; //用户id  } |

定义（接口层）客户端需要返回的对象model为：

|  |  |
| --- | --- |
| public class UserRankingDetailVO {  private int ranking; //排名  private String userId; //用户id  private int followStatus; //关注状态 0：未关注 1：已关注, -1：自己  } |  |

从本地缓存中获取到UserRankingDetail 后，new 一个新的UserRankingDetailVO 对象。将缓存UserRankingDetail 对象中的字段全部复制给UserRankingDetailVO 对象。再将实时获取的followStatus 复制给新对象。

1. 由于缓存中没有关注状态字段，因此缓存不需要更新这个字段。即不会修改缓存数据。
2. 由于每个请求都会重新new 对象，即使是高并发的请求，一方面对缓存的高并发读取没有问题。另一方面，写请求之间各写各的对象，不存在竞争。

## 4、缓存存对象中不实现业务，只提供set与get接口用户存取数据。

如果不这么做会有什么后果？**容易产生循环引用**

在puri-api项目中97373f626f05379823f3b3f2252981adc2936aa6版本前，本地缓存类，在创建对象OriginRankCacheImpl时，存在业务逻辑（获取用户的挂件排名）。循环引用关系如下图所示：

文本, 信件

描述已自动生成

文本

描述已自动生成

这里需要注意的是：

1. 如果在构造函数里面引用对象，可能产生循环依赖。但是，如果没有循环依赖，引用对象一定不为空。样例如下：

文本

低可信度描述已自动生成

在RankService的构造函数里面引用feedService，在不存在循环引用的情况下，在getFromOriginRankCache里面引用feedService时，feedService一定不为null。

1. 如果在外部引用对象，那么在构造函数里面引用该对象时，该对象可能为空。

图形用户界面, 文本, 应用程序, 信件

描述已自动生成

在OriginRankRecommendUserSR构造函数外引用对象rankService。如果需要在构造函数里面使用rankService对象，那么rankService对象可能为空。

# 二、请求问题

## 1、发起远端请求的参数问题

在网页端输入：http://console.micsql.pt.intl.miui.com/beelab/bi/staging

/rankinglist/v1/historyData?date=20210125

可以查询到历史记录。使以下方法发送请求时返回空：

|  |
| --- |
| String requestUrl = **beelabDomain** + **"/rankinglist/v1/historyData?date="** + historyTime;  Request.Builder requestBuilder = Request.*newBuilder*(requestUrl);  response = **httpService**.get(requestBuilder.build()); |

其中：**beelabDomain =** <http://console.micsql.pt.intl.miui.com/beelab/bi/staging>

historyTime = 20210125

**原因：**

**图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成**

虽然url中拼接了header参数 date = historyTime。但是在Request.newBuilder时，后面的参数被自动截取。即requestBuilder = http://console.micsql.pt.intl.miui.com/beelab/bi/staging

/rankinglist/v1/historyData

因此返回空数据

**解决：**

|  |
| --- |
| String requestUrl = **beelabDomain** + **"/rankinglist/v1/historyData"**; requestBuilder = Request.*newBuilder*(requestUrl); **requestBuilder.addParam("date", historyTime);** |

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

浏览器端直接输入全链接可以访问到数据，是因为浏览器自动将？之后的参数填充到header。

## 2、thrift模型传输问题

场景：puri-api 使用GsonUtils.parse方法解析thrift对象时，如果这个对象是一个thrift对象，那么该对象的非string字段的值会在解析时因解析失败从而丢失。



**解决：**

直接传输thrift对象。无需转Json后再传输

## 3、cms cookie问题

在puri-cms工程中，被@LoginRequired修饰的controller，在用postman请求时会被LoginRequiredInterceptor拦截。出现异常：

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

解决：

1. 找到对应环境的某个请求的header.如找到沙盒环境站内信列表的cookie，赋值sessionid：

SESSIONID=d3859a27-9aea-43fb-a9f6-441177f46b75

图形用户界面, 应用程序, 表格

描述已自动生成

1. 在postman中添加cookie

图形用户界面, 文本, 应用程序

描述已自动生成

SESSIONID=d3859a27-9aea-43fb-a9f6表格

低可信度描述已自动生成-441177f46b7图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 应用程序, 网站

描述已自动生成

图形用户界面, 应用程序

描述已自动生成

# 三、数据库操作问题

## 1、数据库插入数据时，需要添加事务保证操作的一致性。

如果使用 @Scheduled 定时启动数据添加操作。那么

@Scheduled 与 @Transactional 不能放在一个方法山。如果放在一起，那么@Transactional会失效。

正确写法参考:puri-mt OriginRankService

@Scheduled 放到接口层， @Transactional 放到service层

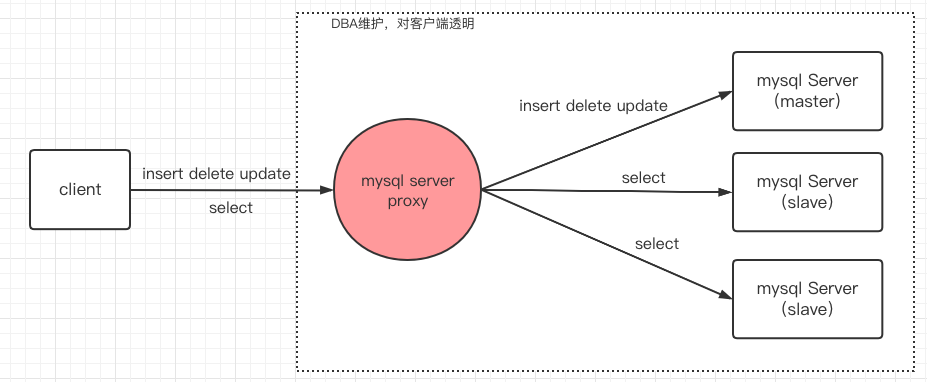
## 2、数据库代理问题。

(1)一般自己搭建mysql服务器后，执行流程如下图所示：

图示

低可信度描述已自动生成

当数据量比较大是，会搭建mysql代理与mysql集群进行分流处理。



一般来说，都是写操作分配到主库，读操作分配到从库。

也有例外场景。比如现有需求：备份puri\_notice\_user表中pushTime早于90天的记录到puri\_notice\_user\_bak表中，然后删除puri\_notice\_user表中的记录。由于主从同步有延迟且不可控，为了保证一定能够读到有效的数据，此时需要读取主库。在这种情况下，可以由mysql server proxy制定规则，如在select后添加”/\*master\*/ ”标识，由mysql server proxy强制将读请求发送给主库。

在小米就是如此约定。<https://wiki.n.miui.com/display/~guolicheng/miproxy>

**读取主库**样例如下：

|  |
| --- |
| <**select id="listNoticeUserByNoticeIdList" resultMap="noticeUserDO"**>  SELECT **/\*master\*/**  <**include refid="Base\_Column\_List"**/>  From puri\_notice\_user  WHERE  id > #{lastId} and `notice\_id` in  <**foreach collection='noticeIdList' item='item' index='index' open="(" separator="," close=")"**>  #{item}  </**foreach**>  limit 500 </**select**> |

## 3、如何查看mybatis日志

Mybatis框架集成了debug级别的日志，当sql语句执行的时候打印出来。如查看puri-mt 站内信分页查询的sql

文本

描述已自动生成

（**listNoticeUserByPageId 代表查询的id**）

tail -f mt-application.log| grep '**listNoticeUserByPageId**'

文本

描述已自动生成

可以看到对于每一个sql执行都有三个debug日志：

1. 红色框中的是sql语句，其中的？代表占位符，也就是入参。如上述第一个sql语句就是

|  |
| --- |
| SELECT n.id as id, n.`type` as type, n.sender\_id as sender\_id, n.extra as extra, unix\_timestamp(n.`push\_time`) as push\_time, n.exclude\_id as exclude\_id FROM puri\_notice\_user nu join puri\_notice n on n.id = nu.notice\_id WHERE (nu.receiver\_id = ? or nu.receiver\_id="") AND n.push\_time <= ? AND (n.end\_time > ? or n.end\_time is NULL) AND ( n.smaller\_version <= ? or n.smaller\_version is NULL ) AND ( n.larger\_version >= ? or n.larger\_version is NULL ) AND n.online\_status = 1 ORDER BY n.push\_time desc ,n.id DESC LIMIT ?; |

1. 紫色框中代表实际的入参

|  |
| --- |
| e53d1da20d27db19(String),  2021-04-06 10:13:30.707 (Timestamp),  2021-04-06 10:13:30.707(Timestamp),  20210401(Long),  20210401(Long),  10(Integer) |

根据（1）（2）可以得到本次的查询：

|  |
| --- |
| SELECT n.id as id, n.`type` as type, n.sender\_id as sender\_id, n.extra as extra, unix\_timestamp(n.`push\_time`) as push\_time, n.exclude\_id as exclude\_id FROM puri\_notice\_user nu join puri\_notice n on n.id = nu.notice\_id WHERE (nu.receiver\_id = **‘e53d1da20d27db19’** or nu.receiver\_id="") AND n.push\_time <= **‘2021-04-06 10:13:30.707’** AND (n.end\_time > **‘2021-04-06 10:13:30.707’** or n.end\_time is NULL) AND ( n.smaller\_version <= **20210401** or n.smaller\_version is NULL ) AND ( n.larger\_version >= **20210401** or n.larger\_version is NULL ) AND n.online\_status = 1 ORDER BY n.push\_time desc ,n.id DESC LIMIT **10**; |

1. 绿色框中代表本次查询的记录数： Total=2，表示结果有两条记录

## 4、如何使用乐观锁

在puri-cms 站内信更新消息时，有sql语句如下：

|  |
| --- |
| <**update id="update"**>  UPDATE puri\_notice  SET `exclude\_id` = #{excludeId},  `type`= #{type}  WHERE `id`= #{id} and **update\_time=#{updateTime}** limit 1 </**update**> |

在此处判断where条件时，加上了**update\_time=#{updateTime}。这**里的update\_time就是使用到了乐观锁。

为什么update\_time可以作为乐观锁？

（1）查看puri\_notice 的DDL。可以看见update\_time的属性为：

DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP COMMENT

ON UPDATE CURRENT\_TIMESTAMP COMMENT 表示当该记录存在写操作（更新|插入）时，该值就会自动更新为当前时间戳

图形用户界面, 应用程序

描述已自动生成

（2）假设用户A的操作时间线如下：

|  |  |
| --- | --- |
| 时间点 | 执行的操作 |
| 2021-04-08 10:00 | 用户A查询（得到update\_time = 10:00） |
| 2021-04-08 10:05 | 用户B执行更新操作(得到update\_time = 10:05) |
| 2021-04-08 10:10 | 用户A更新失败 |

这是因为：用户A上次查询的更新时间为10:00，而在10:10准备更新时，数据库的更新时间为10:05，因此更新失败。如果在用户A的两次操作之间没有人对该条记录修改，那么用户A才可以更新成功。

如果10:10时更新失败，用户A就知道他上次查询到的数据在此之前已经被修改过了，那么本次修改无效。正确的操作是，再次查询数据，拿到最新的数据后再执行更新操作。

## 5、对数据库的操作是否应该封装成jar包？

结论：不推荐

在puri-video-mt 项目中，对数据库表puri\_video\_tag 、puri\_video、video\_tag的操作封装成jar。业务上，需要对这些表的操作通过引用jar包的方式完成。

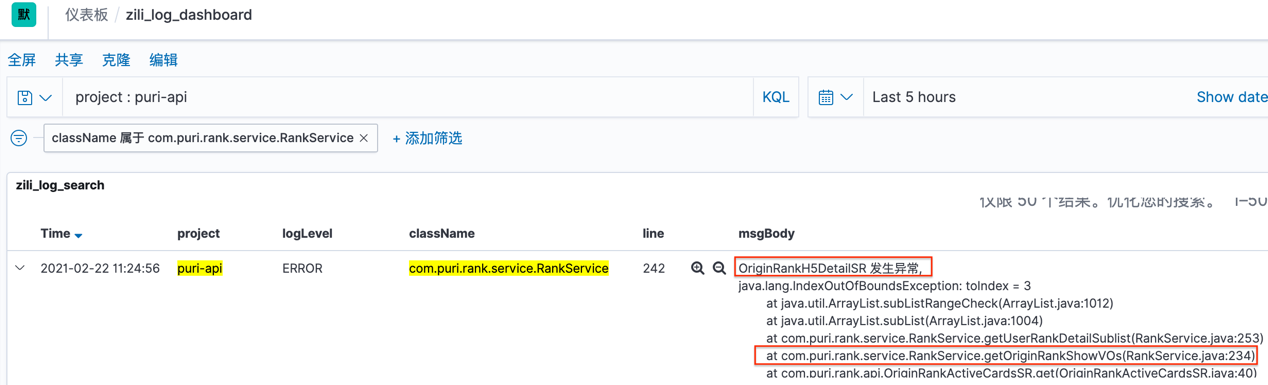
对数据库的操作的封装有两种方式：（1）jar包（2）微服务。两种方式的优缺点如下：

|  |  |  |
| --- | --- | --- |
|  | Jar包 | 微服务 |
| 优点 | 上线操作简单，直接升级包即可 | 1. 数据库的改变（包活缓存）与使用方解偶。数据库的任何改变不会影响使用方 2. 可以对恶意访问进行限流（ip标识恶意者） |
| 缺点 | 1. 如果数据库表表名、字段名或者表结构更换。数据库的更换操作与引用方的**包升级的顺序**应当如何？如果数据库先升级，当前数据库操作异常；如果包先升级，在数据库更新前服务不可用。如果某个引用方有10台机器，这10台机器一般无法同时升级。如果不能同时升级，后升级的机器也会存在数据库访问异常。 2. 如果数据库表表名、字段名或者表结构更换。应当如何添加缓存？如果缓存的redis 前缀与某个特定的东西绑定，比如zili项目的redis key与集群名绑定。点那个集群名发生变化时，key也自动发生变化。就会导致redis key改变，无法正常取到缓存数据。 3. 如果在数据库与业务之间添加一层redis缓存。当redis集群迁移时，所有引用方都需要升级包。因为包中的redis连接的相关配置已经发生了变化 4. 如果存在数据库的恶意访问，jar包无法区分恶意访问源，从而无法对恶意访问限流 | 升级需要重新部署服务 |

# 四、空指针问题

## 1.2021-02-22 排行榜客户端无数据。

查看es日志发现：“OriginRankH5DetailSR 发生异常, ”



**定性分析：**

登陆mb1-miui-global-zili001.awsind机器，查看详细日志：

cd /home/work/log/puri/api/

cat api-app.err.log| grep -A 5 'OriginRankH5DetailSR 发生异常,' | head -n 10

发现问题出现在RankService的234行，sublist数据截取时抛出数组越界异常。

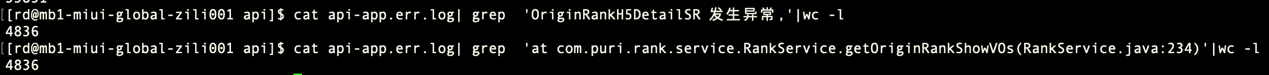
查看Beelab返回数据，发现beelab返回的增粉丝榜的lifestyle垂类的数据不足需要的最少数据。**虽然数据正常返回，但是由于数据不足导致越界**，从而抛出异常，客户端无法正常展示。

**定量分析：**

抛出的‘OriginRankH5DetailSR 发生异常,’是否都是由于RankService的234行引起的？需要进行定量分析

cat api-app.err.log| grep 'OriginRankH5DetailSR 发生异常,'|wc -l

cat api-app.err.log| grep 'at com.puri.rank.service.RankService.getOriginRankShowVOs(RankService.java:234)'|wc -l



发现：这两个出现的出现的行数相同。即说明：‘OriginRankH5DetailSR 发生异常,’全是因为RankService.java:234导致的。

**为什么这个问题排查了这么久？**

1、最开始出现‘OriginRankH5DetailSR 发生异常,’时是由于Beelab数据返回空。所以当再次出现该问题时，进行了错误的前提猜想：Beelab数据返回空

解决：查看异常日志时，应排除所有假想前提。

2、查看异常日志时，只定位了问题的出口：ArrayList 数组下标越界。定位入口时，认为入口是：

At com.puri.rank.service.RankService.getUserRankDetailSublist(RankService.java:253)

正确定位问题的入口在该行的下一行：

at com.puri.rank.service.RankService.getOriginRankShowVOs(RankService.java:234)

解决：查看异常日志时，应完整查看问题的所有调用链路。

# 五、es问题

## 1. es用户缺失

user表里有一个用户，但是es里面没有。这个是插入es的时候失败了。目前只能靠手动补数据。

**解决方法如下：**

（1）：进入：[http://awsmbsrv.kibana.pt.xiaomi.com/app/kibana#/dev\_tools/console?\_g=()](http://awsmbsrv.kibana.pt.xiaomi.com/app/kibana" \l "/dev_tools/console?_g=()" \t "_blank)

（2）: 去user数据库查询到该用户信息，然后根据信息修改下列post请求，在es里面执行一下就可以。

（3）：post请求如下：

POST /mig5\_2\_zili\_user/\_doc/

{

"userId" : "c6920adc3fc86f5e",

"name" : "dream\_boy\_5star",

"gender" : 1,

"region" : "IN",

"ppId" : "42099289",

"intro" : "",

"fansCount" : 5667,

"followCount" : 0,

"likeCount" : 88,

"videoCount" : 1,

"createTime" : 1608951655000,

"updateTime" : 1585290185000

}

图形用户界面, 文本, 应用程序

描述已自动生成

# 六、站内信相关问题

## 1.站内信的查询方式分为两种。

**1）如果是内部cms使用。**查询时可以根据pageIndex 和 pageSize来进行查询。示例参考

puri-cms项目的/list接口

|  |
| --- |
| @Select(  "<script>"+  "SELECT \* FROM puri\_notice WHERE "  +" (type = 1 or type = 2 or type = 3) "  +"<if test='status != -1'>"  +" and online\_status = #{status}"  +"</if>"  +"<if test='noticeIdSet != null'>"  +" and id in (<foreach collection='noticeIdSet' item='item' index='index' separator=','>#{item}</foreach>)"  +"</if>"  +" order by id desc"  **+" limit #{pageIndex},#{pageSize}"**  + "</script>") @ResultMap(value = **"**noticeDO**"**) List<NoticeDO> list(@Param(**"**pageIndex**"**)**int** pageIndex, @Param(**"**pageSize**"**)**int** pageSize, @Param(**"**status**"**)Integer status, @Param(**"**noticeIdSet**"**) Set<String> noticeIdSet); |

**2）如果是客户端使用。**查询是应该传入noticeId和pageSize来进行查询。这是因为传入noticeId后，可以根据id>传入的id查询。此时可以使用到索引，因此查询速度较快。

**如果给客户端使用时，查询仍然使用pageIndex 和 pageSize，（puri-mt项目）会有以下问题：**

* **如果站内信的信息与接收者的信息在一个表中，那么站内信无法根据消息进行聚合展示。且消息字段冗余大，数据库占用内存大，会对以后的查询产生影响。**
* **如果站内信的信息与接收者的信息分别放在不同的表中。那么需要根据pageIndex与pageSize分页查询时，必须要进行join。两表的join操作一般是不允许。且join的表不能太大，否则数据库容易因为join而卡死。**

**解决方案：**

1. 缩小数据库。需要操作的数据库仅保存仅3个月的数据。超过3个月的数据放入备份表归档。且需要启动定时任务，定时备份超过最近3个月的数据。
2. 布隆过滤器的位运算法。在mysql中快速判断某个用户是否在某个集合中。

假设集合S中有元素A、B、C。如何快速判断元素A是否在S中？

定义运算 F(A,B,C) = A & B & C

如果**F(A,B,C) & A** = A & B & C **= F(A,B,C)，那么A在集合S中。**

**这个计算有三个问题：**

1. 字符本身不能直接进行位运算。如果要进行位运算，需要首先将字符或者字符串转成数字。
2. &运算是只要有0，相&之后结果就是0.因此多数相&之后，最终的结果很有可能为0。而0与任何数相&都是0。如果**F(A,B,C)=0，那么**

**F(A,B,C) & N** = 0。N为任意数。这样还是无法区分该元素是否在指定集合中。

（3）F(A,B,C) & A = F(A,B,C),当A的每一位全为1时也成立

3.将历史数据归档后，join两表。

4.将puri\_notice\_user表分表后再进行join。因为需要根据接收者列出当前用户的站内信，所以按照receiver\_id分表。同时需要根据noticeId查询站内信详情，因此还需要冗余一份，按照noticeId分表。

# 七、mq相关问题

## 1.mq通信的数据格式应该放在jar包中。如果放在jar中，是统一放在一个common中，还是按业务需求放在不同的jar包？还是需要使用方新建一个model接收消息数据？

|  |  |  |  |
| --- | --- | --- | --- |
|  | 使用方新建类 | 按业务打小jar包 | 所有常量类打jar包 |
| 优点 | 1. 使用方与消息解偶。 2. 使用方根据自身需要接收字段，不必全盘接受 | 1. 使用方只需要引入消息jar包即可。 2. 消息字段有修改时，只需要升级包即可   3.便于业务的拆分宇扩展 | 1.2同按业务打小jar包  3.所有的常量都在同一个工程，便于查找 |
| 缺点 | 1. 使用方有多个时，每个使用方都需要复制一份消息model 2. 当消息变动时，需要手动添加修改 | 1.使用方与消息存在耦合  2.使用方可以乱用jar包中的model。比如使用方可以将消息的model用户接受http请求参数等 | 1.2同按业务打小jar包  3.当需要按业务拆分时，不易扩展。如jar包中既存在push相关的model，也存在notice相关model。现在想要将push相关的业务拆分出去时，原有的常量包就不好拆分 |

# 八、redis问题

## 1. redis集群迁移问题（李鹏）？

Zili的redis集群注册在Zookeeper中。

**线上节点注册**在：/cache/redis-cluster/global-big-data-cluster.json

<https://zkview.d.xiaomi.net/clusters/azmbmiuisrv/nodes?path=%2Fcache%2Fredis-cluster/global-big-data-cluster.json>

图形用户界面, 文本, 应用程序, 聊天或短信

描述已自动生成

**线上节点的密码**注册在：/cache/redis-service/bdata

<https://zkview.d.xiaomi.net/clusters/azmbmiuisrv/nodes?path=%2Fcache%2Fredis-service/bdata>

图形用户界面, 文本, 网站

描述已自动生成

**由于两个节点不在一处，因此无法同时修改新集群的ip和密码。存在一定的时间差。为了100%迁移新集群，采用了临时方案：**

1. 在zookeeper集群上新增一个节点bdata2：/cache/redis-service。

图形用户界面, 应用程序

描述已自动生成

原来的getKey方法：key = d(redis.cache.prefix) + “:” + bdata(节点名称)

在redis-kit项目底层，新增getKey2（String key）方法, return key。固定直接传入旧key = “d:bdata”

然后在使用处调用getKey2(“d:bdata”)方法即可。

**（较好）方案二：**

1.新建bdata2节点

2.将节点的share属性改为false

3.将bdata2节点的前缀改成需要的前缀。（比如d:bdata）

# 九、其他问题

## 1.枚举类型对象的获取

枚举的valueOf方法传入的参数是枚举名。根据枚举名获取枚举对象

|  |
| --- |
| **public enum** RankEnum {  *//Zili Star 活动榜* ***RANKING\_OF\_ZILI\_STAR\_NEWFANS***(**"1"**, **"rankingOfZiliStarNewFans"**, **"Zili Star 活动榜榜单名"**),  **private static** Map<String, RankEnum> *rankMap*;  **private** String **code**; *//榜单名code* **private** String **rankName**;*//榜单名* **private** String **desc**; *//描述* } |

***RANKING\_OF\_ZILI\_STAR\_NEWFANS*** **rankingOfZiliStarNewFans** = RankEnum.valueOf(“***RANKING\_OF\_ZILI\_STAR\_NEWFANS***”);

## 2. 前端跨域问题

<https://www.cnblogs.com/xuepei/p/5068236.html>

跨域(Cross-origin)是指a页面想获取b页面资源，如果a、b页面的协议、域名、端口、子域名不同，或是a页面为ip地址，b页面为域名地址，所进行的访问行动都是跨域的，而浏览器为了安全问题一般都限制了跨域访问，也就是不允许跨域请求资源。

在排行榜需求中，前端访问服务端资源时，需要跨域。

tail -f access.log | grep origin

报纸上的文字

中度可信度描述已自动生成

其中 /puri/v1/rank/origin/activecards 是客户端在手机端内发送的请求

/puri/v1/rank/origin/homepage是h5 从榜单首页（http://sandbox-h5.zilivideo.com/h5/creator/rank）发送的请求，请求服务端数据

**跨域问题的解决：**

[CORS](https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS)（Cross-Origin Resource Sharing）,  通过在服务器端设置：Access-Control-Allow-Origin响应头，允许浏览器发起跨域请求，例如，跨域XMLHttpRequest, Fetch请求。

|  |
| --- |
| @Override **protected void** preProcess() {  Response response = **this**.getResponse();  response.setAccessControlAllowOrigin(**serverConfiguration**.getString(**"security.access.control.allow.origin"**, **"\*"**));  response.setAccessControlAllowCredentials(**true**); } |

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

## 3. 批处理问题及解决方式

使用批处理的方式进行业务处理时，有两个通用的问题：

1. 服务端在批处理过程中，如果遇到服务器重启，那么该批处理可能会丢失数据与状态。
2. 如果批处理的量较大，可能出现超时问题。

**解决**批处理的一般方式：

使用某种存储介质保存需要批处理的数据，保存成功后向上游返回成功。然后从该介质中依次读取数据，异步进行处理。

存储介质一般选择mq和mysql。选择哪种介质，取决于场景

|  |  |  |
| --- | --- | --- |
|  | Mq | mysql |
| 优点 | 1. 框架较多，一般都有自动重试机制和分组机制 | 1.不容易确定失败的消息  2.容易确定批处理的进度 |
| 缺点 | 1. 不容易确定失败的消息 2. 不容易确定批处理的进度 | 1. 失败时需要手动重试 2. 无法对消息指定消费组 |
| 使用场景 | 上游给mq发送消息后，不关心下游的执行成功与否。或者能够接受下游的失败结果 | 需要了解批处理的执行进度，且在失败时可以明确得到失败的数据与原因。 |

## 4. 项目重构问题及一般解决方式

在进行项目重构时，可能遇到如何切分系统的问题：以业务纬度重构系统？以数据源纬度重构系统？

一般采用据源纬度重构系统

如在重构puri-reco-srv系统与重构puri-reco-pool时，就遇到此问题：

puri-reco-srv 只负责读mango（首页视频读取）

puri-reco-pool 只负责写mysql,mango与es，并保证三端数据源一致。

从业务纬度切分，puri-reco-srv 与 puri-reco-pool就是两个项目。为了保证首页视频的稳定性，需要将首页视频的读取单独拆分成一个服务。而为了保证三端数据的写入一致性，则需要为数据的写入另外生成一个服务。

从数据的纬度切分，应该将mango的读写聚合成一个服务。而将mysql、es的读写聚合成另一个服务。需要同步mango时，向mq中发送消息，指定mango服务为消息者，再由mango服务自行写入数据，以此达到写入三端数据源。

## ？？？？thrift项目如何实现服务隔离？？？？？？？？？？？？？？？

puri-video-mt工程是一个thrift服务。调用该服务的有puri-api与puri-cms。为了保证puri-api请求puri-video-mt的服务不受puri-cms请求puri-video-mt服务的影响，需要对请求puri-video-mt的请求进行隔离。

实现服务隔离的一般框架有：dubbo。

thrift不支持服务隔离，为了实现上述需求，采用下面的解决方式：

1. 可以通过在thrfit的外层套dubbo框架的方式实现（代价较大）
2. ？？？？？？》