Redis实践

> RedisUtil是工具类，封装了Java对redis操作

1.计数

- 限制接口访问量 **INCR**

  场景：为了防止询价接口被恶意刷，需要限制用户每日询价次数

  key：evaluate_limit_+用户唯一标识UID 例如：evaluate_limit_391128438125228

  ```
  String value = RedisUtil.get(key);
  Integer count = Strings.isEmpty(value)? 0:Integer.parseInt(value);
  if(count>=20) {
      return Result.notOk(-1, "已达到今日询价次数上限")
  }
  ....
  估价
  ....
  RedisUtil.incr(key, 24*60*60);
  ```

- 统计日活，七日活，月活 **HyperLogLog**

  key：activity_日期 例如：activity_20200924 value:用户唯一标识比如用户UID

  > Redisutil.pfAdd(key,UID);

  统计日活的话，只需要根据当日的日期拼接key：activity_+当日日期，然后调用HyperLogLog 的PFCOUNT即可；统计月活的话，使用PFMERGE命令合并该月每日的HyperLogLog 存储的数值即可。

2.缓存

- **String**

  场景：小程序AccessToken每两个小时需要获取一次，获取之后存到缓存中，同时设置缓存时间为expire_time。

```
try (ZZLock lock = zzLockClient.newLock(lockName)) {   // 防止有多个线程同时刷新accessToken
    if(lock.acquire( t: 5)) {
        String appId = recycleWHSConfig.getWhsQQConfig().getAPPID();
        String secret = recycleWHSConfig.getWhsQQConfig().getAPPSECRET();
        String getAccessTokenUrl = recycleWHSConfig.getWhsQQConfig().getAccessTokenUrl().replace( target: "APPID",appId).replace( target: "APPSECRET",secret);
        QQAccessTokenResponse accessTokenResponse = retryer.call(()->{
            Response response = okHttpUtil.get(getAccessTokenUrl);
            if(response == null || response.code() != 200){
                return null;
            }
            String result = response.body().string();
            log.info(" {} qqMsgHelper refreshAndGetAccessToken result={}",logStr, result);
            QQAccessTokenResponse qqAccessTokenResponse = GsonUtil.fromJson(result, QQAccessTokenResponse.class);
            if(StringUtils.isNotEmpty(qqAccessTokenResponse.getAccess_token()) && (
                    qqAccessTokenResponse.getExpires_in() != null && qqAccessTokenResponse.getExpires_in() > 0)){
                RedisUtil.set(WHSConstant.Redis.QQ_ACCESS_TOKEN_KEY,qqAccessTokenResponse.getAccess_token(),qqAccessTokenResponse.getExpires_in());
                return qqAccessTokenResponse;
            }
            return null;
        });
        if(accessTokenResponse != null) {
            return accessTokenResponse.getAccess_token();
        }
    }
```

获取小程序AccessToken接口，返回值为AccessToken以及token有效时间expire_time

获取的AccessToken放入缓存中，有效时间为调用小程序接口返回的expire_time

- **List**

场景：批量生成太阳码，放入list构成的队列中，需要的时候从队列中获取，避免接口直接调用小程序接口带来的接口超时

预生成太阳码并放入队列中

```
Runnable task = () -> {
    log.info("异步检查二维码生成队列");
    String logStr = LogContext.getLog();
    String lockName = [redacted]
    String queueName = [redacted]
    // 锁失效时间
    int ttl = 5;
    // 重试间隔
    int interval = 1;
    // 重试次数
    int retryCount = 3;
    try ([redacted] lock = [redacted].newLock(lockName)) {
        // 阻塞等待竞争锁，如果失败阻塞重试
        if (lock.acquire(ttl, interval, retryCount)) {
            Long length = RedisUtil.llen(queueName);
            log.info("{} QR_CODE_QUEUE size={}", logStr, length);
            //判断是否低于最小阈值，扩充标贴
            if (Objects.isNull(length) || length.intValue() < MIN_BUFFER_NUM) {
                produceQrCodes(logStr);
            }
        }
    } catch (Exception e) {
        log.error("{} error=try_to_produce_failed", logStr, e);
    }
};
```

分布式锁，防止同时多个线程来生成小程序码

```
private void produceQrCodes(String logStr) {
    Integer produceNum = MAX_BUFFER_NUM.intValue();
    for (int i = 0; i < produceNum; i++) {
        //获取标贴
        QrCode qrCode = createQrCode(logStr);
        if (qrCode != null) {
            //插入队列
            pushQueue(logStr, qrCode);
        }
    }
    log.info("{} produceQrCodes queue length={}", logStr, RedisUtil.llen([redacted]));
}

private void pushQueue(String logStr, QrCode qrCode) {
    String queue = [redacted]
    RedisUtil.lpush(queue, JSON.toJSONString(qrCode));
    log.info("{} desc=push_queue qrCode={}", logStr, JSON.toJSONString(qrCode));
}
```

获取太阳码

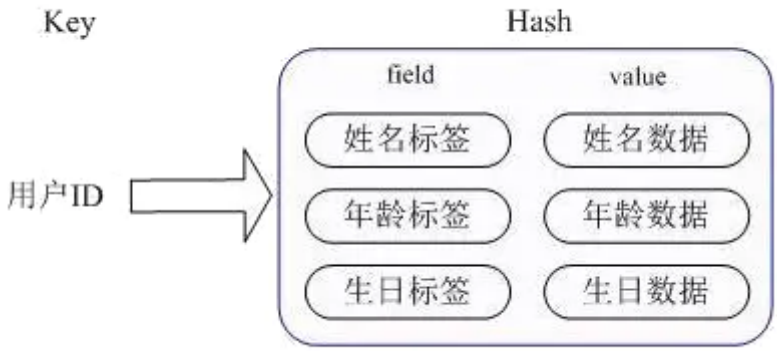
```java
public Optional<QrCode> popQcCode(String logStr) {
    String queueName = [redacted];
    String value = RedisUtil.rpop(queueName);
    QrCode qrCode = null;
    if (Strings.isEmpty(value)) {
        qrCode = createQrCode(logStr);
    } else {
        qrCode = JSONObject.parseObject(value, QrCode.class);
    }
    log.info("{} desc=pop qrCode={}", logStr, GsonUtil.toJson(qrCode));

    //检查队列是否到达阈值
    asynCheckQueue();

    return Optional.ofNullable(qrCode);
}
```

- **Hash**

  场景：存储对象



  比起将对象转化为json字符串再存入缓存，直接用Hash存储对象的话，可以单独更改某个filed的值。`hset key field value`

  `hset`是以哈希散列表的形式存储，超时时间只能设置在键`key`上，单个域`field`不能设置过期时间。

  机型选择列表页需要展示价格跟图片信息，品类-品牌-机型的三级类目ID体系，启用定时任务，从第三方RPC接口拉取数据放入缓存提升接口性能。以品牌ID作为key，以机型ID作为field，以价格/图片作为value。

- **Sorted set** （ZSET）

  场景：需要记录用户的搜索记录并且把最近一个月的历史搜索记录展示给用户（按照用户搜索时的时间倒序）需要去重

  添加用户搜索记录

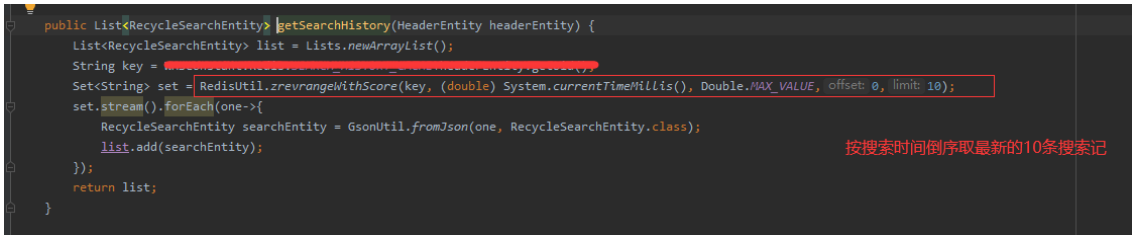
```java
public void addSearchRecord(HeaderEntity headerEntity, String keyword, String cateId, String brandId,String brandName, String modelId) {
    String key = [redacted];
    RecycleSearchEntity searchEntity = RecycleSearchEntity.builder().keyword(keyword).cateId(cateId).brandId(brandId).brandName(brandName).modelId(modelId).build();
    Long expireTime = System.currentTimeMillis() + 30 * 24 * 60 * 60 * 1000L;
    RedisUtil.zadd(key, expireTime.doubleValue(), GsonUtil.toJson(searchEntity));
    Set<String> delSet = RedisUtil.zrangeWithScore(key, min: 0, (double) System.currentTimeMillis());
    if(CollectionUtils.isNotEmpty(delSet)) {
        String[] arr = delSet.stream().toArray(n -> new String[n]);
        RedisUtil.zrem(key, arr);
    }
}
```

设置score为当前时间时间戳+30天毫秒

删除30天之前的历史记录，因为不需要展示给用户了

  注意score为当前时间时间戳+30天的毫秒数

  获取用户搜索记录



相比于集合Set，有序集合增加了score，引入了分数的概念，可以按照分数来进行排序。比如

排行榜。在实际应用中，可以按照是否需要排序来选择使用集合Set还是有序集合Sorted set

SET

- 微信微博点赞，收藏，标签

```
1》点在

SADD like:{消息ID} {用户ID}

2》取消点赞

SREM like:{消息ID} {用户ID}

3》检查用户是否点过赞

SISMEMBER like:{消息ID} {用户ID}

4》获取点赞的用户列表

SMEMVERS like:{消息ID}

5》获取点赞用户数

SCARD like:{消息ID}
```

帖子点赞

用户A查看用户B的个人页，会提供一个共同好友的列表，这个时候用set的交集取A跟B的交集即可

3.消息队列 List

除了使用list作为队列缓存数据之外，List也可以作为消息队列。不过由于已经有比较流行的RocketMq

等消息中间件，所以在实际的应用中一般不使用Redis作为消息队列。

4.分布式锁

分布式锁的实现方式有多种，比如数据库，etcd，zookeeper等，比较热门的还有redis来实现分布式锁。感兴趣可以查阅先关分布式锁资料

缓存穿透/击穿/雪崩

缓存穿透

缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，如发起为id为“-1”的数据或id为特别大不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大。

解决方案： 数据库查询不到的时候设置key对应的value为null，时间设置的可以短一些，比如30s，对请求的参数校验

缓存击穿

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力

解决方案：1. 设置key不过期或者开启定时任务更新缓存并重置缓存有效期

2. 加互斥锁

缓存雪崩

缓存雪崩是指缓存中数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至down机。和缓存击穿不同的是， 缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决方案： 添加一个随机时间/定时任务定期刷新

数据库与缓存双写一致性

1.先更新缓存再更新数据库

不行，更新完缓存，数据库异常回滚导致数据不一致

2.先更新数据库再更新缓存

可能出现A线程更新数据库->B线程更新数据库->B线程更新缓存->A线程更新缓存

导致数据不一致

3.先删除缓存再跟新数据库

该方案会导致不一致的原因是。同时有一个请求A进行更新操作，另一个请求B进行查询操作。那么会出现如下情形:
（1）请求A进行写操作，删除缓存
（2）请求B查询发现缓存不存在
（3）请求B去数据库查询得到旧值
（4）请求B将旧值写入缓存
（5）请求A将新值写入数据库

采用延时双删策略

（1）先淘汰缓存
（2）再写数据库（这两步和原来一样）
（3）休眠1秒，再次淘汰缓存，这个操作可以作为异步的

4.先更新数据库，再删缓存(这种方式出现不一致的情况比较少)

- **失效**：应用程序先从cache取数据，没有得到，则从数据库中取数据，成功后，放到缓存中。
- **命中**：应用程序从cache中取数据，取到后返回。
- **更新**：先把数据存到数据库中，成功后，再让缓存失效。

数据不一致的场景： 比如，一个是读操作，但是没有命中缓存，然后就到数据库中取数据，此时来了一个写操作，写完数据库后，让缓存失效，然后，之前的那个读操作再把老的数据放进去，所以，会造成脏数据。

> 删除key失败可以发送MQ消息，然后在MQ中异步删除