Redis的使用

1. addTask 操作

当 reason 非空时：

- **key**: personPullTaskCountKeyReason，其中 category + source + reason 来自请求。
- 查找最小 score 对应的 (value, score)。
- 如果 score == 1000，删除 personPullTaskCountKey。
- 如果 score != 1000，查找该 key 中所有 score == score 的 value 对应的 userIdSet。
- 如果查到的 userIdSet 非空：
  - 设置该 key 一小时过期。
  - 比较 personPullTaskCountKey（默认值为 1）和 userIdSet.size，取二者的最小值。
  - 遍历从 0 到最小值，执行以下操作：
    - 如果 cancelAddZremPullTaskCountSwitch <= 0（默认值为 0），将当前的 value 移除。
    - 将当前的 value 加进来，并将 score 设置为 1000。
    - 调用 pushTaskToUser。

当 reason 为空时：

- **key**: category + source + reason。
- 使用 withScores 0,0 获取 userIdTuples。
- 如果 score == 1000，删除该 key。
- 否则，使用 byScore 查找所有 score 等于最小 score 的 value，得到 userIdSet。
- 如果 userIdSet 非空，设置该 key 一小时后过期。
- 取 userIdSet.size 和 pushAddTaskDesignationNumTaskSwitch（默认值为 1）的最小值，在 0 到最小值范围内遍历 value：
  - 移除当前的 value。
  - 将当前的 value 加进来，并将 score 设置为 1000。
  - 调用 pushTaskToUser。

2. finishTask操作

- **key**: category + source + reason。
- 获取该 key 中审核员 ID 的 score：
  - 如果 score < 0，则将 score 设置为 0。
  - 否则，将 score 减 1。
- 不论 score 为多少，都设置该 key 一小时后过期。

3. doRedisPullTask 操作

- **key**: category + source + reason。
- 向该 key 的有序列表中加入 score 为 拉任务 + 推任务 + milo 的任务数，value 为审核员 ID。
- 设置该 key 一小时过期。

异步向 Redis 添加任务

- **锁**: lock key（与一般 key 不同）。
- **key**: category + source + reason。
- 如果 Redis 中该 key 对应的数量小于预定义的最小值（例如 100），则加入新的数据：
  - 如果锁 + category + source + reason + task id 的 key 存在，则执行以下操作：
    - zadd，以请求的 category + source + reason 为 key，score 为优先级，value 为 task record id。
    - 如果优先级为高，则 score = int.max - id + 优先级；如果是 LIFO，则 score = id(long)；否则，score = int.max - id(long)。
    - 设置 14 天过期。
    - 解锁（lock key, uuid）。

5. releasetask 操作

- **任务审阅**：变为 unrevised。
- **key**: category + source + reason（有无 reason 两个版本）。
- 如果带 ID 的锁 key 存在，则：
  - zadd，score = double.max - 1，value = task id。
  - 设置 14 天过期。

updateTask

- **步骤 1**：先更新数据库，将 taskTYPE 修改为 normal。
- **步骤 2**：再更新 Redis：
  - 如果原 taskTYPE 是 double，则 score = double.max - 1。
  - 否则，使用与异步添加逻辑一样的方式更新。
  - 设置 14 天过期。

PullRedis

使用lua脚本从Redis中对任务进行拉取

Java 复制代码

```java
protected List<Long> pollRedis(String categoryType, String sourceType, String reason,
Integer remainCount) throws TException {
    String key = buildSlotRedisCommonTaskSortedSet(categoryType, sourceType, reason);

    List<String> ids = new ArrayList<>(
        (Collection<? extends String>)
```

```
            corvusMarsRedis.eval(
                "local entries = redis.call('ZRANGE', KEYS[1], ARGV[1], ARGV[2]); if
#entries > 0 then redis.call('ZREMRANGEBYRANK', KEYS[1], ARGV[1], ARGV[2]); for k,v in
pairs(entries) do "
                    + "redis.call('SET', string.format(KEYS[2], entries[k]), ARGV[3],
'NX', 'PX', ARGV[4]) end return entries; end return entries;",
                Arrays.asList(key,
RedisClusterUtils.buildClusterSlot(getRedisCommonTaskSortedSet(categoryType, sourceType,
reason), LOCK_SYNC_PROCESSING_TASK_TASK_ID)),
                Arrays.asList(String.valueOf(-remainCount), String.valueOf(-1),
UUID.randomUUID().toString(), String.valueOf(ONE_SECOND_TIMESTAMP * 5))
            )
    );

    log.info(key + "fetch id{}", CollectionUtils.isEmpty(ids) ? "0" : ids.toString());
    return ids.stream().map(Long::valueOf).collect(Collectors.toList());
}
```

- 用ZRANGE拉取“最多 remainCount”个任务ID（从右边，即优先级最高的元素 — -remainCount到-1）。
- 如果拉取到entries：
  - 用ZREMRANGEBYRANK删除这些ID，保证任务不会重复分配。
  - 遍历entries，对每个ID用SET key value NX PX ttl加锁：key是KEYS[2]格式化后的slot，value是UUID，TTL=5秒（防止死锁）。

> **防止并发处理时的竞态条件**
> 即使任务ID通过原子操作（ZRANGE + ZREMRANGEBYRANK）被移除，但在分布式系统中，可能存在多个消费者同时执行类似操作。若某个任务因异常被**重新加入队列**（如处理失败后的重试机制），加锁可确保同一时刻仅有一个进程处理该任务，避免重复执行。

- 返回拉取到的entries。

Thrift使用从-1到0

Thrift从0到0.1

如何作为服务端对外暴露新接口

1. 编写.thrift文件
   a. 在service.thrift定义接口，包括入参、出参格式（建议用optional，如果必传，在类里校验）
   b. 定义用到的需要和外部交互的实体，例如request.thrift, response.thrift, entity.thrift
2. 如果是现有方法只是修改一下入参，可以到这步直接git commit\ push，会自动触发流水线生成可用snapshot。如果需要加新方法，继续👇
3. 执行thrift命令，上一步定义的方法会加进.iface接口，生成对应java类

Java 复制代码

```
生成 java 接口代码文件：
mkdir javagen
thrift -out javagen/ --gen java ping.thrift
执行完之后可以看到 javagen 文件夹下多出含包名（com.zhengdekai.thrift.demo）路径的 PingService.java 文件。
移动或复制生成的上一步文件到项目源码对应的包目录下
```

4. 在impl.java实现XX.Iface接口，方法里可以调用service bean，写xxxservice.java实现具体逻辑
5. 按需进行entity-dto的转换

idl文件编写

在xx.thrift文件修改完之后推送到远端

提交之后git有一个ci，在ci那里看上次改完的流水线

编译完了之后有一个版本号 在pom里面把版本号替换后测试一下

没问题了就合并到主分支然后idl仓库建个tag

把服务pom的版本号改为这个

异步查询

例 getUserInfoAll 一种典型的异步任务管理模式

启动多个异步任务

- 使用 CompletableFuture.runAsync(() -> { }) 可以在异步线程池中运行一段代码（即一个无返回值的任务）。
- 每次调用都会返回一个 CompletableFuture<Void>，表示该任务的异步执行状态。

b. 将任务加入列表

- add() 方法将每个启动的异步任务（即 CompletableFuture<Void>）添加到 List<CompletableFuture<Void>> 中。
- 这个列表可以用来管理所有异步任务的生命周期。

c. 等待所有任务完成

- 使用 CompletableFuture.allOf() 方法可以统一等待列表中的所有任务完成。
- 它是常见的"批量任务管理"模式，尤其适用于需要并发处理多个操作的场景。

Java 复制代码

```java
CompletableFuture<Void> f = CompletableFuture.allOf(allUserFuture.toArray(new CompletableFuture[0]));
```

- **创建任务列表**
- 使用 List<CompletableFuture<Void>> futures 来存储所有异步任务的 CompletableFuture 实例。
- **启动异步任务**
- CompletableFuture.runAsync(() -> { ... }) 启动一个异步任务，每个任务在独立的线程中运行。

- 每个任务的 CompletableFuture 实例通过 add() 方法添加到 futures 列表中。
- **管理所有任务**
- CompletableFuture.allOf(futures.toArray(new CompletableFuture[0])) 将多个 CompletableFuture 聚合成一个新的 CompletableFuture，用于等待所有任务完成。
- **同步等待**
- allOf.join() 阻塞主线程，直到所有任务完成，确保主程序不会提前结束。
- **任务完成后**
- 所有任务完成后，主程序继续执行并打印 All tasks are completed!。

Lambda 表达式只能用于**函数式接口**。函数式接口是指只包含一个抽象方法的接口。

非常适合completablefuture

Java 复制代码

```java
CompletableFuture.runAsync(new Runnable() {
    @Override
    public void run() {
        System.out.println("Running in an asynchronous task");
    }
});

```