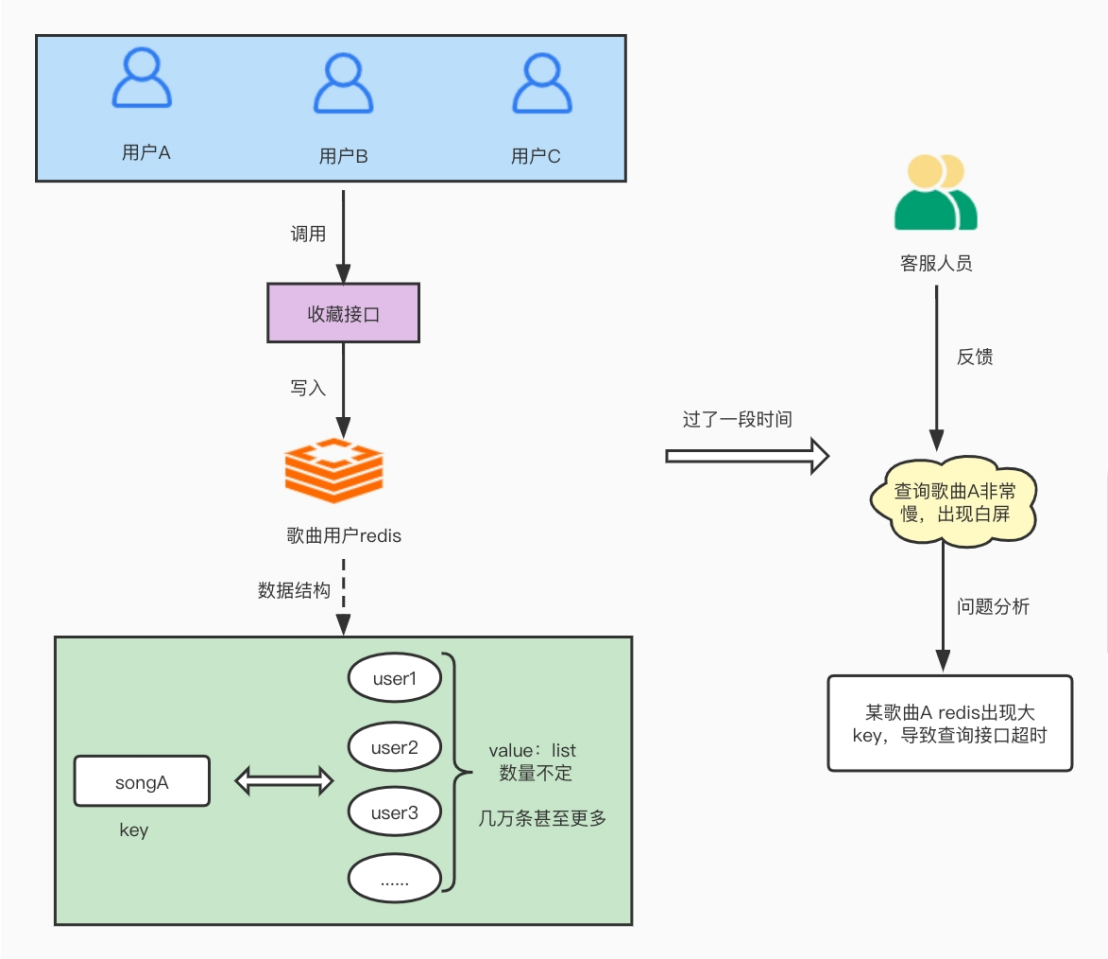
Redis 大 key 解决方案

tojson 老师

1、redis 大 key 经典生产问题分析



某音乐节，收到客服反馈通知，说 APP 查询某爆款歌曲收藏的用户列表非常缓慢。于是开发人员找到一个出问题的爆款歌曲 A，通过搜索日志系统找到 traceId，根据 traceId 从我们的 skywalking 分布式调用链路系统跟踪发现，是操作 redis 时间比较久，并且从日志系统搜到一些 redis 查询超时的异常。最后我们定位到的原因如下：

在收藏歌曲接口的时候拿 redis 做了一个缓存，记录收藏该歌曲的用户列表，redis 数据结构：key = song， value = 用户 id 列表，而 redis 查询发现多了许多大 key，体现在一个爆款歌

曲几万甚至更多的用户收藏，导致 Redis 响应缓慢，查询经常超时，导致服务这边出现服务器异常，APP 页面没拿到接口的数据从而出现白屏的现象。

2、redis 大 key 基本概念及常见场景

很多朋友肯定在想 redis 的 key 能有多大呀？这里就有个误区了，所谓的大 key 问题是某个 key 对应的 value 比较大，所以本质上是大 value 问题。

key 往往是开发过程中可以自行设置，可以控制大小，value 往往不受程序控制跟业务场景有关系，因此可能导致 value 很大。

2.1 基本概念

在 Redis 中，大 key 指的是 key 对应的 value 值所占的内存空间比较大。

- ✓ value 是 string 类型，大小控制在 10kb 以内
- ✓ value 是 hash、list、set、zset 等集合类型，元素个数不要超过 5000（或则 1 万、几万）

上述的定义并不绝对，主要是根据 value 的大小和元素个数来确定，业务也可以根据自己的场景确定标准。

2.2 常见场景

大 key 的产生往往是业务方设计不合理，没有预见 vaule 的动态增长问题.

- ✓ 一直往 value 塞数据，没有删除及过期机制，迟早要爆炸
- ✓ 数据没有合理做分片，将大 key 变成以一个个小 key

接下来我们看看几类比较经典的场景：

- ➢ 社交类：如果某些明星或者大 v 的粉丝列表不精心设计下，必是 bigkey。
- ➢ 统计类：例如统计某游戏活动玩家用户的榜单列表，除非没几个人完，否则必是 bigkey。
- ➢ 缓存类：将数据从数据库 load 出来序列化放到 Redis 里，这个方式非常常见，但有两个地方需要注意:

  第一，是不是有必要把所有字段都缓存

  第二，有没有相关联的数据，关联数据分开存储

例如： 遇到过一个例子，该同学将某明星一个专辑下所有视频信息都缓存一个巨大的json 中，造成这个 json 达到 6MB，后来这个明星发了一个官宣。用户浏览专辑，因为 redis 大 key 瞬间扛不住了。

3、redis 大 key 带来的影响

- ✓ **客户端超时阻塞**。由于 Redis 单线程的特性，操作 bigkey 的通常比较耗时，也就意味着阻塞 Redis 可能性越大，这样会造成客户端阻塞或者引起故障切换，会出现各种 redis 慢查询中。
- ✓ **内存空间不均匀**。集群模式在 slot 分片均匀情况下，会出现数据和查询倾斜情况，部分有大 key 的 Redis 节点占用内存多，QPS 高。
- ✓ **引发网络阻塞**。每次获取大 key 产生的网络流量较大，如果一个 key 的大小为 1MB，每秒访问量为 1000，那么每秒会产生 1000MB 的流量。这对于普通千兆网卡的服务器来说是灾难性的。
- ✓ **阻塞工作线程**。执行大 key 删除时，在低版本 redis 中可能阻塞线程。

4、redis 大 key 如何检测

- ✓ 改写 redis 客户端，在 sdk 中加入埋点，实时上报数据给 redis 大 key 检测平台、监控告警。
- ✓ scan + debug object bigkey 命令。循环遍历 redis key 序列化后的长度。debug object bigkey 可能会比较慢，它存在阻塞 Redis 的可能，建议在从节点执行该命令，官方不推荐。
- ✓ scan + memory usage。该命令是在 Redis 4.0+以后提供的，可以循环遍历统计计算每个键值的字节数。
- ✓ 通过 python 脚本迭代的 scan key，对每次 scan 的内容进行判断是否大 key。
- ✓ redis-cli --bigkeys。可以找到某个 redis 实例 5 种数据类型(string、hash、list、set、zset)的最大 key。但如果 redis key 比较多，执行该命令会比较慢，建议在从节点执行该命令。
- ✓ rdbtools 开源工具包。rdbtools 是 python 写的一个第三方开源工具，用来解析 Redis 快照文件，redis 实例上执行 bgsave，然后对 dump 出来的 rdb 文件进行分析，找到其中的大 key。

例如：rdb dump.rdb -c memory --bytes 10240 -f redis.csv

从 dump.rdb 快照文件统计（bgsave）,将所有 > 10kb 的 key 输出到一个 csv 文件

5、redis 大 key 如何删除

如果对这类大 key 直接使用 del 命令进行删除，会导致长时间阻塞，甚至崩溃。因为 del 命令在删除集合类型数据时，时间复杂度为 O(M)，M 是集合中元素的个数。Redis 是单线程的，单个命令执行时间过长就会阻塞其他命令，容易引起雪崩。那我们怎么解决呢？

✓ 主动删除大 key

1）分批次渐进式删除

一般来说，对于 string 数据类型使用 del 命令不会产生阻塞。其它数据类型分批删除，通过 scan 命令遍历大 key，每次取得少部分元素进行删除，然后再获取和删除下一批元素。对 Hash, Sorted Set, List，Set 分别处理，思路相同，先对 key 改名进行逻辑删除，使客户端无法使用原 key，然后使用批量小步删除。

- 删除大 Hash

步骤：

（1）key 改名，相当于逻辑上把这个 key 删除了，任何 redis 命令都访问不到这个 key 了

（2）小步多批次的删除

伪代码：

```
# key改名
newkey = "gc:hashes:" + redis.INCR( "gc:index" )
redis.RENAME("my.hash.key", newkey)

# 每次取出100个元素删除
cursor = 0
loop
  cursor, hash_keys = redis.HSCAN(newkey, cursor, "COUNT", 100)
  if hash_keys count > 0
    redis.HDEL(newkey, hash_keys)
  end
  if cursor == 0
    break
  end
end
```

- 删除大 List

伪代码：

```
# key改名
newkey = "gc:hashes:" + redis.INCR("gc:index")
redis.RENAME("my.list.key", newkey)

# 删除
while redis.LLEN(newkey) > 0
  redis.LTRIM(newkey, 0, -99)
end
```

- 删除大 Set

伪代码：

```
# key改名
newkey = "gc:hashes:" + redis.INCR("gc:index")
redis.RENAME("my.set.key", newkey)

# 每次删除100个成员
cursor = 0
loop
  cursor, members = redis.SSCAN(newkey, cursor, "COUNT", 100)
  if size of members > 0
    redis.SREM(newkey, members)
  end
  if cursor == 0
    break
  end
end
```

- 删除大 Sorted Set

伪代码：

```
# key改名
newkey = "gc:hashes:" + redis.INCR("gc:index")
redis.RENAME("my.zset.key", newkey)

# 删除
while redis.ZCARD(newkey) > 0
  redis.ZREMRANGEBYRANK(newkey, 0, 99)
end
```

2）采用 unlink + bigkey 异步非阻塞删除。这个命令是在 redis 4.0+提供的代替 del 命令，不会阻塞主线程。

✓ 被动删除大 key

被动删除是指利用 redis 自身的 key 清除策略，配置 lazyfree 惰性删除。但是参数默认是关闭的，可配置如下参数开启，如下所示：

lazyfree-lazy-expire on	#过期惰性删除
lazyfree-lazy-eviction on	#超过最大内存惰性删除
lazyfree-lazy-server-del on	#服务端被动惰性删除

6、redis 大 key 如何设计与优化

主要针对以下两种经典场景进行优化：

✓ **单个 key 存储的 value 很大（超过 10kb）**

1）从业务角度评估，value 中只存储有用的字段，尽量去掉无用的字段。

2）可以考虑在应用层先对 value 进行压缩，比如采用 LZ4/Snappy 之类的压缩算法，配合 redis 客户端序列化配置，可以无侵入完成 value 的压缩。

3）value 设计的时候越小越好，关联的数据分不同的 key 进行存储。

4）大 key 分拆成几个 key-value， 使用 multiGet 获取值，这样分拆的意义在于分拆单次操作的压力，将操作压力平摊到多个 redis 实例中，降低对单个 redis 的 IO 影响。

5）对 redis 集群进行扩容

✓ 集合数据类型 hash，list，set，sorted set 等存储过多的元素（超过 5000 个）

类似于场景一种的第一个做法，可以将这些元素分拆。

以 hash 为例，原先的正常存取流程是 hget(hashKey, field) ; hset(hashKey, field, value)

现在，我们可以分拆构建一个新的 newHashKey，具体做法：固定一个桶的数量，比如

10000，每次存取的时候，先在本地计算 field 的 hash 值，取模 10000， 确定了该 field 落在哪个 newHashKey 上。

```
newHashKey    =    hashKey + (*hash*(field) % 10000）;
hset (newHashKey, field, value) ;
hget(newHashKey, field)；
```

set, sorted set, list 也可以类似上述做法.