云享书库 第4期
bbs.huaweicloud.com

企业级Redis 技术与应用解读

互联网业务神器 GaussDB(for Redis)

企业级稳定可靠

百万QPS并发，秒级扩容，海量存储

华为云自研架构

存算分离，多副本强一致

对垒开源Redis

1/4成本，10倍大容量

- 华为云开发者社区 | 华为云数据库架构与规划团队 -

序言

提起 Redis，互联网从业者无人不知，无人不晓。毕竟，开源 Redis 作为一款经典的“缓存”产品，其提供的丰富数据类型非常好用，能支撑众多业务架构搭建，广受开发者青睐。在游戏、电商、社交媒体以及其它互联网行业中，发挥着重要的作用，也有着巨大的产品市场。

然而，随着近年来各行业规模逐渐拓展，业务需求急速增加，数据量和并发访问量呈指数级增长，几乎只能依附于关系型数据库的传统“缓存”逐渐难以支撑上层业务，开源 Redis 也面临着如“容量有限，高并发写入容易 OOM”、“内存昂贵，成本降不下来”、“可靠性有限，容易丢关键数据”等种种难解问题。

为了解决开源 Redis 痛点以及自运维数据库的难用问题，华为云推出了云原生分布式数据库 GaussDB(for Redis)。

GaussDB(for Redis) 是一款基于计算存储分离架构的，兼容 Redis 生态的云原生 NoSQL 数据库，并提供强一致、三副本存储，高度保证数据的安全可靠。

GaussDB(for Redis)突破了开源 Redis 的内存限制，通过将数据进行冷热分离，在保证热数据驻留计算节点内存满足业务低时延要求的同时，将冷数据置换入分布式存储池进行持久化存储，最大程度的降低使用成本，具有高兼容、高性价比、高可靠、弹性伸缩、高可用、冷热分离等特点。

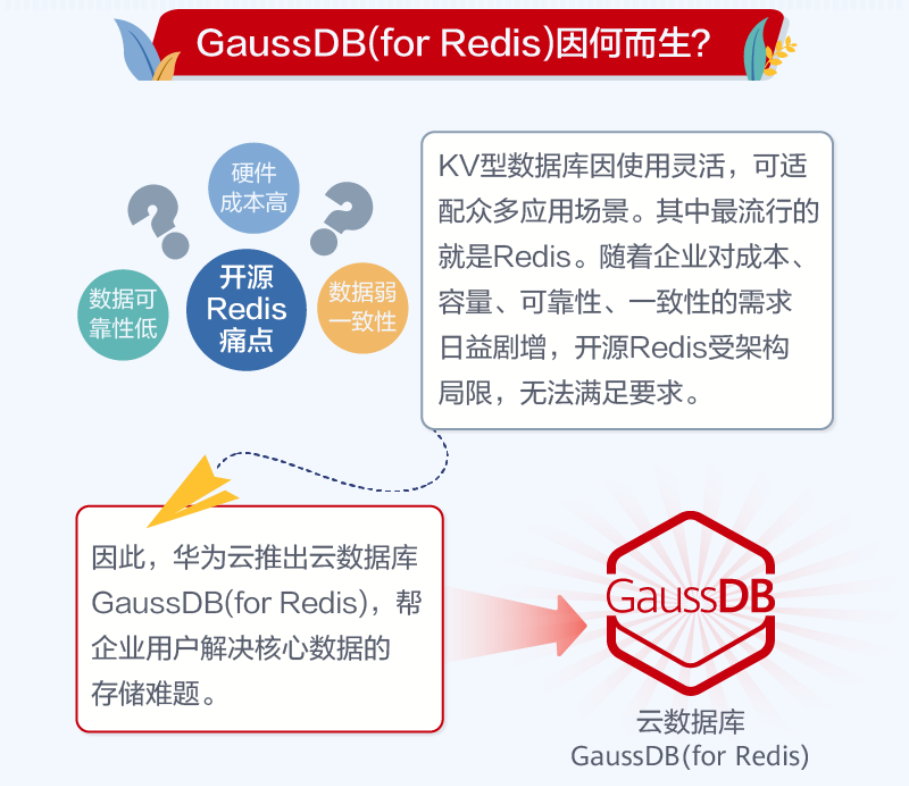
为便于数据库领域开发者系统性的了解学习 Redis，本书通过入门篇、性能篇、测评篇、应用篇四个章节，与大家分享华为云 GaussDB(for Redis)的技术创新与实践，内容聚焦问题解决、场景应用和开发实战。欢迎开发者与我们交流讨论，也希望 GaussDB(for Redis)能帮助大家构建性能更好、运行更稳定的应用。

入门篇

初识云数据库 GaussDB(for Redis)

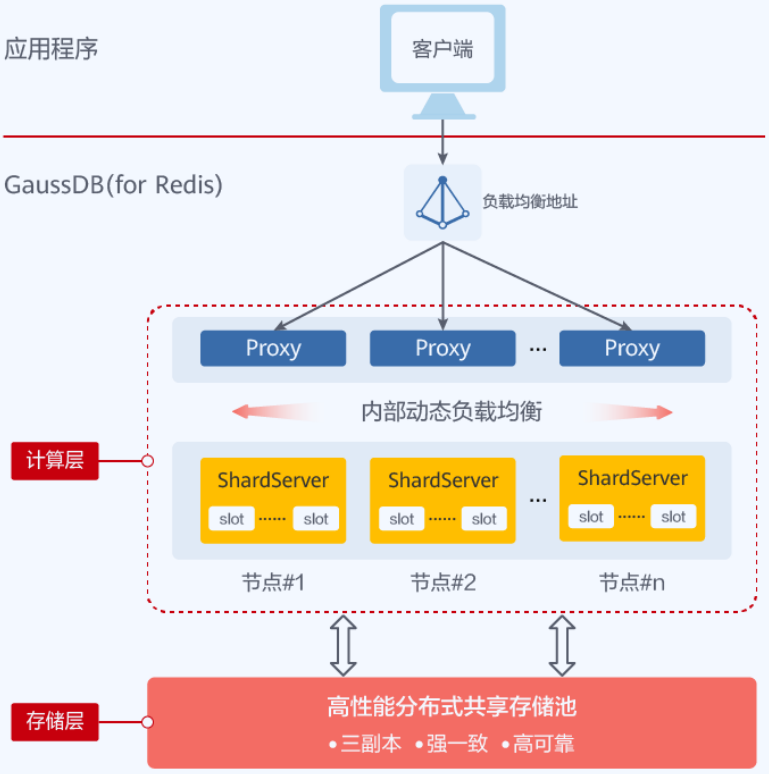
一、什么是 GaussDB(for Redis)?

GaussDB(for Redis)是一款基于计算存储分离架构，兼容 Redis 生态的云原生 NoSQL 数据库。GaussDB(for Redis)突破了开源 Redis 的内存限制，通过将数据进行冷热分离，在保证热数据驻留计算节点内存满足业务低时延要求的同时，将冷数据置换入分布式存储池进行持久化存储，最大程度的降低使用成本。



GaussDB(for Redis)长什么样?

在实例内部，GaussDB(for Redis)采用自研的计算、存储分离架构。架构图如下：



计算层

提供可弹性伸缩的吞吐性能。用户只需使用最简单的StandAlone客户端，即可快速接入稳定可靠的企业级Redis。

存储层

提供高可靠的数据存储能力。扩容秒级完成，业务0感知。

GaussDB(for Redis)有何价值?

低 成本

- 采用高性能持久化技术，数据**实时落盘**。
- 无fork问题，容量全部可用。

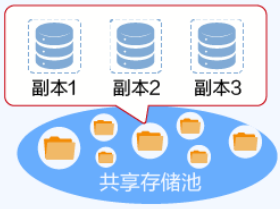
成本降低75%~90%

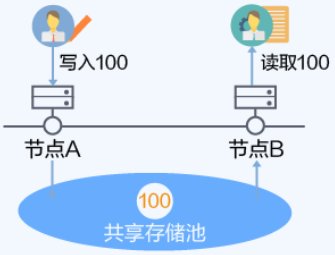
高 稳定性

- 即使N−1节点故障，全量数据依旧可用。

高 可靠性

- 数据**三副本**冗余存储，无丢失风险。



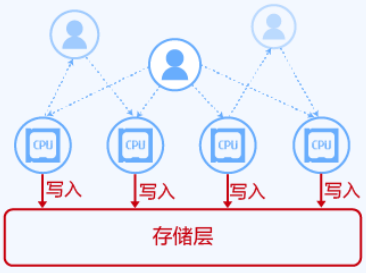


强 一致性

- **强一致性**保障，多点访问无脏读问题。

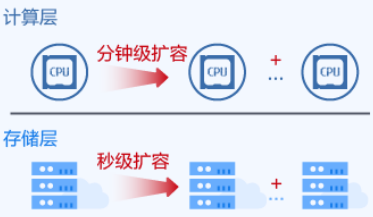
强 抗写能力

- **多线程**设计，不易发生命令阻塞。
- 全部节点都可读、可写，业务高峰无压力。



强 扩展能力

- 节点扩容分钟级完成，业务仅秒级感知。
- 容量**扩容秒级**完成，业务0感知。



二、GaussDB(for Redis)典型应用

GaussDB(for Redis)作为兼容 Redis 接口的 key-value 数据库，扩展了社区版原生 Redis 的应用场景，使其不再仅仅运用于缓存，而是可以更好的满足持久化，混合存储等多样化的业务需求。

1. 电商行业

电商应用的商品数据具有较为明显的冷热特征，使用 GaussDB(for Redis)后，热门商品信息作为热数据驻留在内存中，冷门商品信息会置换到共享存储池中，这样既满足了热门商品的快速访问需求，又解决了海量商品数据纯内存存储成本高的问题。

电商应用的海量历史订单数据，可使用 GaussDB(for Redis)进行持久化存储。通过 Redis 接口完成数据存取，可支持 TB 级海量数据存储。

电商大促活动会导致短时间出现大量并发访问，可选择 GaussDB(for Redis)作为前端缓存（需要配置大内存），帮助后端数据库抗过业务高峰。GaussDB(for Redis)可针对计算节点一键式秒级无损扩容的特点，也可以帮助客户更加从容的应对此类计划性的流量突发行为。

2. 游戏行业

游戏业务数据 Schema 较为简单，可选择 GaussDB(for Redis)作为持久化数据库，通过使用简洁的 Redis 接口快速完成业务开发上线。例如，可使用 Redis 的有序集合结构完成游戏排行榜的实时展现。

对于时延非常敏感的游戏场景，也可以使用 GaussDB(for Redis)作为前端缓存（需要配置大内存），加速应用访问。

3. 视频直播

热门直播间往往占据了视频直播应用的大多数流量，使用 GaussDB(for Redis)，可以更加有效的利用有宝贵的内存资源，通过在内存中保留热门直播间数据，在共享存储中保留冷门直播间数据，为客户降低使用成本。

4. 在线教育

在线教育应用的特点是，系统中存储有大量的课程，试题，解答等信息，但通常只有热门数据（包括热门课程，最新题库，名师讲解等）会被高频访问。使用 GaussDB(for Redis)，可以有效的根据数据的热度，决定存入内存或共享存储，实现性能与成本的平衡。

其他需要支持持久化存储的应用

除上述场景外，随着互联网飞速发展，各种大型应用对持久化存储的需求与日俱增，具体来说，需要存储包括：历史订单、特征工程、日志记录、位置坐标、机器学习、用户画像等信息在内的海量数据。这些数据的共同特点是：数据量大，有效期长，需要一个支持大容量，低成本的 key-value 存储服务完成数据的采集和流转。Redis 作为当前应用最为广泛的 key-value 服务，其丰富的数据结构和操作接口对于存储此类数据具有先天优势，但由于原生 Redis 只能作为缓存，因此无法在持久化存储领域发挥作用。

GaussDB(for Redis)在兼容 Redis 接口的同时，又提供了大容量，低成本，高可靠的数据存储能力，可以作为此类持久化存储场景的理想解决方案。

三、GaussDB(for Redis)的购买与使用

GaussDB(for Redis)的购买与使用，具体内容见**快速入门**和**用户指南**：
https://support.huaweicloud.com/redisug-nosql/nosql_02_0071.html

GaussDB(for Redis)免费体验：企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，**点击体验**

四、附录

- **GaussDB(for Redis)资料导航**
- 更多技术文章，关注 **GaussDB(for Redis)官方博客**
- 数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！点击【**数据库论坛**】

性能篇

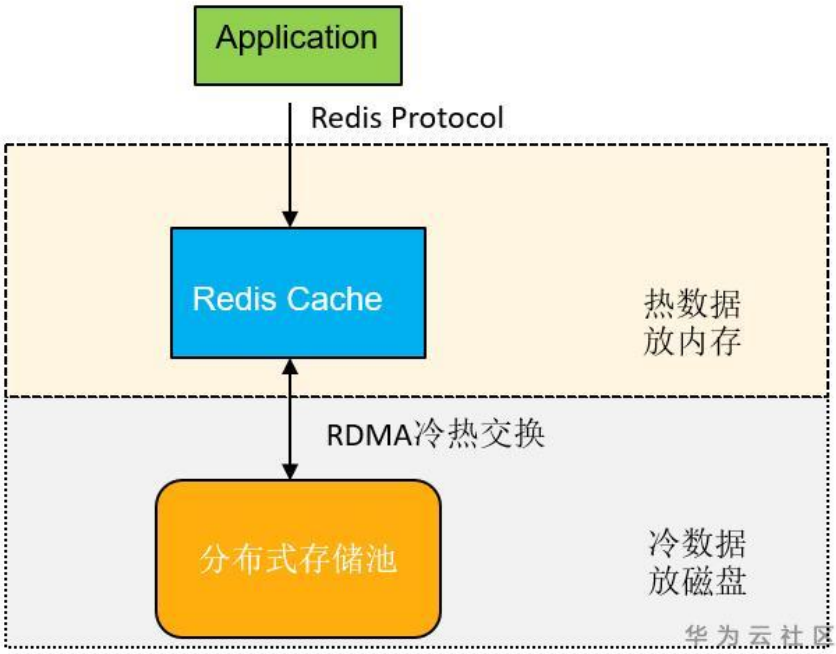
突破开源 Redis 的内存限制，存算分离的高斯 Redis 到底有多能“装”？

GaussDB(for Redis)(下文简称高斯 Redis)是华为云数据库团队自主研发的兼容 Redis 协议的云原生数据库，该数据库采用计算存储分离架构，突破开源 Redis 的内存限制，可轻松扩展至 PB 级存储。

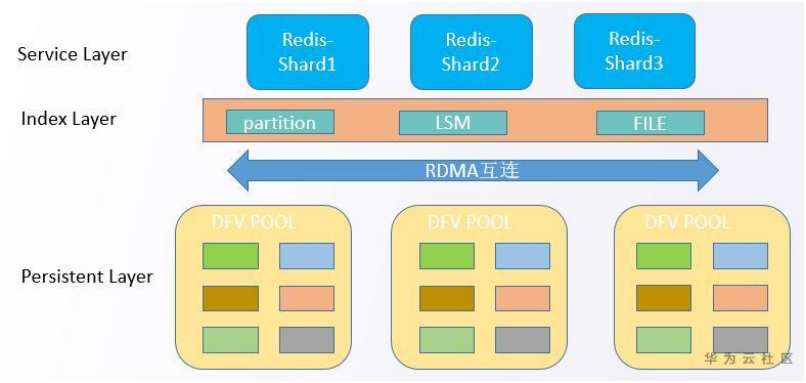
本文将从存储架构、四大特性、竞争力、应用场景等方面进行介绍。

一、 存储架构

高斯 Redis 基于计算存储分离架构，计算层实现热数据缓存，存储层实现全量数据的落盘，中间通过 RDMA 高速网络互连，通过算法预测用户的访问规律，实现数据的自动冷热交换，最终达到极致的性能提升。



二、 四大特性



该架构基于华为内部强大且广泛使用的自研分布式存储系统 DFV，实现了一套 Share Everything 的云原生架构，充分发挥了云原生的弹性伸缩、资源共享的优势，使得高斯 Redis 具备强一致、秒扩容、低成本、超可用的四大特点，完美避开了开源 Redis 的主从堆积、主从不一致、fork 抖动、内存利用率只有 50%、大 key 阻塞、gossip 集群管理等问题。

1. 强一致

数据复制是存储的事情，因此专业的事情交给专业的团队来做。通过分布式存储 DFV，高斯 Redis 轻松实现了 3 副本强一致，并可轻松支持 6 副本，为业界首创。

在强一致架构下，用户再也不用担心开源 Redis 的主从堆积，带来的丢数据、不一致、OOM 等极端问题，更不用担心业务出错，比如计数器、限流器、访问统计、hash 字段等不一致。

2. 秒扩容

数据规模膨胀之后，扩容是个高危且困难的操作。高斯 Redis 基于云原生架构，将扩容分成计算层和存储层。计算层扩容，无需任何数据搬迁，只需修改路由映射，即可秒级完成。存储层是个共建的超级数据湖，其容量巨大，而且扩容是切成细腻度的 64MB 数据分区，对上层数据库业务几乎无感。

因此高斯 Redis 可以轻松支撑业务的大规模膨胀，并且真正做到计算/存储分层的按需扩容和购买。

3. 低成本

高斯 Redis 相对于开源 Redis，在存储介质上使用了磁盘替代内存。一方面，由于采用存算分离架构，计算资源少了一半，即没有从节点；另一方面，存储资源按需购买，无任何浪费，并且采用了逻辑/物理压缩。最终，每 GB 综合成本不到开源 Redis 的十分之一。

4. 超可用

开源 Redis 或友商 Redis 不管单分片还是集群，其数据复制都采用主从架构，导致 N 个节点的集群，如果同时挂掉一对主从（即 2 个节点），整个集群就不可用。而高斯 Redis 采用存算分离之后，每个计算节点都可以看到并共享所有数据，因此 N 个节点，最多可以容忍挂掉 N-1 个节点，真正做到比高可用还高的可用性。

三、 竞争力分析——详细对比请见链接

四、 场景推荐

高斯 Redis 不仅性能逼近缓存，而且其存储能力（扩展性、高性能、易用性）超越数据库。因此除了缓存场景可以选择高斯 Redis 以外，上至 PB 级别大规模数据存储都可以选择高斯 Redis。场景参考如下：

GaussDB(for Redis)免费体验：企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，[点击体验]

五、 选择建议

推荐指数: ★★★★★	推荐指数: ★☆☆☆☆
关心成本，数据量较大	不差钱，也不关心成本
不要求极致时延，p99时延满足5-10ms即可	时延要求极高，关心p9999或max时延
对经常访问的数据要求低时延，不经常访问的数据，接受时延较高	时延要求稳定，不允许波动
正在使用pika/ssdb/tendis/tedis/kvrocks/Redis混合存储/ardb等	总数据量小于50G
关心数据一致性	库存、粉丝、限流、统计、特征、标签等出现不一致，也无所谓
需要稳定平滑的自动备份	不需要备份，或者备份导致OOM也不关心

华为云社区

【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专

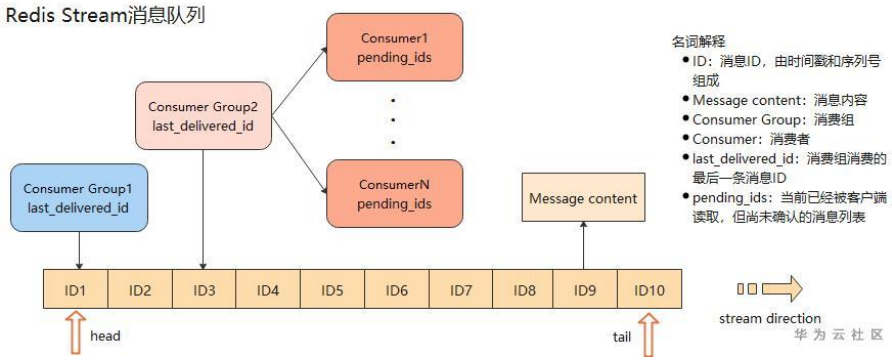
业技能！**点击前往**

Redis 消息队列 Stream 应用探讨

引言：Redis Stream 是 Redis 5.0 引入的一种新的数据类型，其本质是一个消息队列，类似于 kafka 等消息中间件。它提供了消息的落地存储功能，并实现了类似 kafka 消费组和消费者的功能。与 kafka 相比，Redis Stream 同样拥有强大的功能，但因原生 Redis 无法有效支持大规模数据存储，成本昂贵，并存在数据丢失/不一致风险等原因，导致其未能流行起来。本文将对 Stream 的常用命令和应用场景进行介绍，并探讨原生 Redis Stream 消息队列的缺陷以及 GaussDB(for Redis)提供的解决方案，供大家学习和选用。

一、 Redis Stream 简介

与 Pub/Sub 相比，Redis Stream 具有消息的落地存储功能，每一个客户端能访问任意时刻的消息，并且能记录每一个客户端的访问位置，还能保证消息不会丢失。Redis Stream 的结构如下所示，它有一个消息链表，将所有加入的消息都链接起来，每个消息都有唯一的 ID 和对应的内容。



如图所示，每一个 Stream 队列包含多条消息，每条消息由唯一的 ID 进行标识，由时间戳和序列号组成，例如 1627849609889-0。每条消息以追加的方式添加到 Stream 队列中。同一个 Stream 队列可以包含多个消费组（Consumer Group），每个消费组的状态都是独立的，同一个 Stream 队列的消息可以被多个消费组重复消费。

同一个消费组又包含多个消费者（Consumer），这些消费者之间是竞争关系，不同消费者不会重复消费同一条消息，任意一个消费者读取了队列中的一条消息都会使消费组中的游标 last_delivered_id 往前移动。该方式提高了并发效率，例如，多个进程并发处理 Stream 队列中的消息。每个消费者中维持一个状态变量 pending_ids，简称为 PEL(Pending Entries List)，记录了当前已经被客户端读取的但尚未被 ACK 的消息，确保消息被客户端成功消费。

Redis Stream 命令可以分为消息队列命令和消费者命令两类，如下所示：

以即时通讯中的聊天室场景为例，使用 Redis Stream 作为中间件，实现聊天室的发言以及信息查看。

- 使用 XADD 命令进行发言

```
# XADD:向stream队列中添加一条消息，若该stream队列不存在，则创建队列并添加消息。返回值为插入消息对应的ID。
# 命令格式为：XADD key ID field value [field value ...]。其中的参数含义如下：
#   a) key：steam消息队列的名字
#   b) ID：消息ID，若指定为*，表示该消息ID由Redis生成。
#   c) field value：消息对应的kv记录。
127.0.0.1:6379> XADD chatRoom * date 2021/02/22-13:59:20 name John message "Is anyone
going shopping this afternoon ?"
"1613992491502-0"
127.0.0.1:6379> XADD chatRoom * date 2021/02/22-14:01:10 name Mary message "ok, I will go
with you."
"1613992500582-0"
127.0.0.1:6379> XADD chatRoom * date 2021/02/22-14:03:20 name Tena message "sorry, I have
English class this afternoon"
"1613992506645-0"
```

- 使用 XLEN 命令获取聊天室发言的数量

```
# XLEN:获取stream消息队列的长度
# 命令格式为：XLEN key。其中key为steam消息队列的名字
127.0.0.1:6379> XLEN chatRoom
(integer) 3
```

- 使用 XRANGE 获取消息队列的消息

```
# XRANGE：从指定的消息区间内获取消息集合。返回值为消息集合。
# 命令格式：XRANGE key start end [COUNT count]。参数含义如下：
#  a) key：steam消息队列的名字。
#  b) start：查询区间起始消息ID，若为“-”，表示消息ID的最小值
#  c) end:  查询区间结束消息ID，若为“+”，表示消息ID的最大值
#  d) count：要获取消息的数量，为可选参数
127.0.0.1:6379> XRANGE chatRoom - + count 1
1) 1) "1613992491502-0"
   2) 1) "date"
      2) "2021/02/22-13:59:20"
      3) "name"
      4) "John"
      5) "message"
      6) "Is anyone going shopping this afternoon ?"
```

- 使用 XREAD 命令读取消息。可以在不设置消费组和消费者的情况下，使用 XREAD 的命令进行消息读取，此时 Stream 队列类似于一个普通的列表（list）。

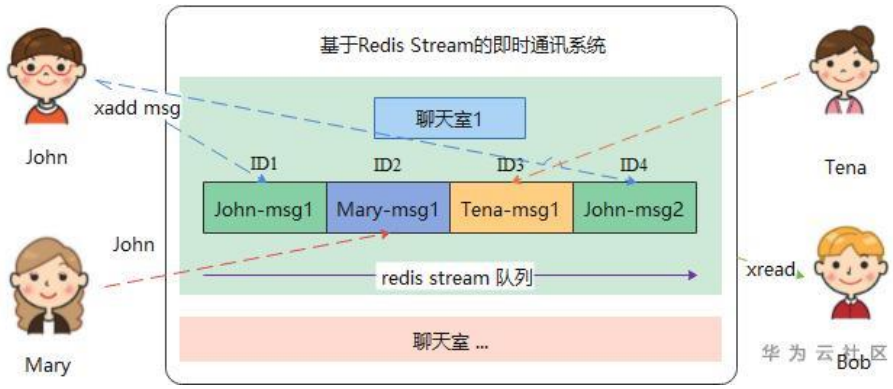
```
# XREAD:从一个或多个stream队列中以阻塞或者非阻塞的方式获取数据。返回值为消息集合，若获取不到数据则返回“nil”。
# 命令格式：XREAD [COUNT count] [BLOCK milliseconds] STREAMS key [key ...] ID [ID ...]，参数含义如下：
#  a) count：要获取消息的数量
#  b) milliseconds：阻塞消息的毫秒数，若不设置BLOCK则为非阻塞模式。
#  c) key：steam消息队列的名字
#  d) ID:  查询stream队列起始消息ID，若ID为“0-0”，用于获取消息队列中的全部消息；若ID为具体的ID，则获取的消息列表从下一个ID开始计算；若ID为“$”，则表示获取最新消息。
# 使用xread以非阻塞的方式获取消息
127.0.0.1:6379> XREAD count 5 streams chatRoom $
(nil)
# 以阻塞的方式获取最新消息，新开启一个窗口，并输入XADD chatRoom * date 2021/07/09-16:38:21 name Bob message “ok, Agree”,XREAD阻塞窗口接收到的消息如下：
127.0.0.1:6379> XREAD BLOCK 50000 count 5 streams chatRoom $
1) 1) "chatRoom"
   2) 1) 1) "1613995207772-0"
         2) 1) "date"
            2) "2021/2/22-14:08:21"
            3) "name"
            4) "Bob"
            5) "message"
            6) "ok, Agree"
(3.01s)
```

更多的 Redis Stream 命令使用请参考官方文档（https://redis.io/commands/xread）。

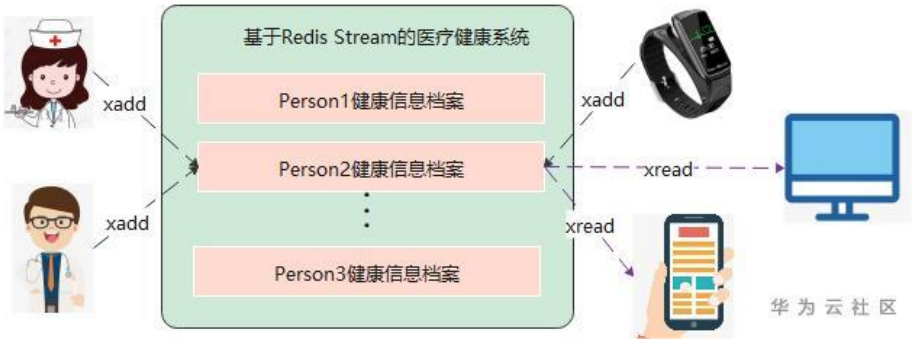
二、 应用场景

由于 Redis Stream 天然有序，特别适合存储时序数据，应用场景包括即时通讯、智慧医疗、流量削峰、智慧城市等领域。

（1）即时通讯：微信、QQ 等是我们日常生活中常用的通讯软件，常用的聊天方式包含点对点通讯和群聊两种方式。下图是一个群聊的模型图，当采用 Redis Stream 作为通讯的中间件，创建一个群聊时，在 Redis 中对应地为该群聊创建一个 Stream 队列。在发送消息时，将每个用户的消息按照时间顺序添加到 Stream 队列中，保证了消息的有序性。由于 Stream 是一个持久化的队列，无论是在线还是离线状态，每个用户可以多次查看历史消息，保证了通讯的完整性。



（2）智慧医疗：医疗行业的信息化，可以更好地为服务于每一个人。为每一个人从出生起建立一份健康档案，记录相应的健康信息，如体检报告、诊断报告、用药信息、以及智能终端实时上传的健康指标。这些信息都是一些时序数据，同样可以采用 Redis Stream 来实现智慧医疗系统。建立起智慧医疗系统后，使用终端可以查看所有的医疗信息，并会提示患者按时吃药，在终端上传身体指标异常时，会自动报警并预约挂号。现阶段每个医院都有自己的信息系统，不同的医院很难查到同一个患者的医疗信息，在未来，医疗上云将有利于解决医疗信息孤岛，更好的帮助每一位患者。



（3）流量削峰：在常见的秒杀活动或团购中，如春运抢票、商城促销等，通常短时间内有大量的流量，导致系统崩溃。由于每一个用户在请求时对应唯一的时间戳，所有的请求都有一个先后顺序，同样可以采用 Redis Stream 作为中间件，将请求加入到 Redis Stream 消息队列。将消息转存到消息队列间接提供给应用，而非直接发送给应用，可以防止大流量

冲击导致的系统崩溃。当消息队列中的请求数量达到规定的最大值时，直接回复客户端抢购失败。

GaussDB(for Redis)免费体验： 企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，**点击体验**

三、 原生 Redis 是否真的适用于以上场景?

如上应用场景具有数据规模大、数据持续增长的特点，虽然原生 Redis 有良好的设计初衷，但是并不能解决实际问题。具体体现在：

- 无法有效应对大规模数据：原生 Redis 是一个基于内存的数据库，单个节点存储容量有限，当扩展至 TB 级别的集群，将会出现管理困难，运维成本高等问题。
- 集群扩容影响业务性能：原生 Redis 在进行集群扩容时，需要重新划分 hash 槽并进行数据迁移，必定会影响业务性能。
- 数据可能会丢失:原生 Redis 虽然可以采用 RDB 和 AOF 的方式对数据进行持久化，但是并不会实时地将每一条命令写入到硬盘中，当出现掉电或集群崩溃的情况，必定会丢失一部分数据，对于类似智慧医疗场景，是难以忍受的。

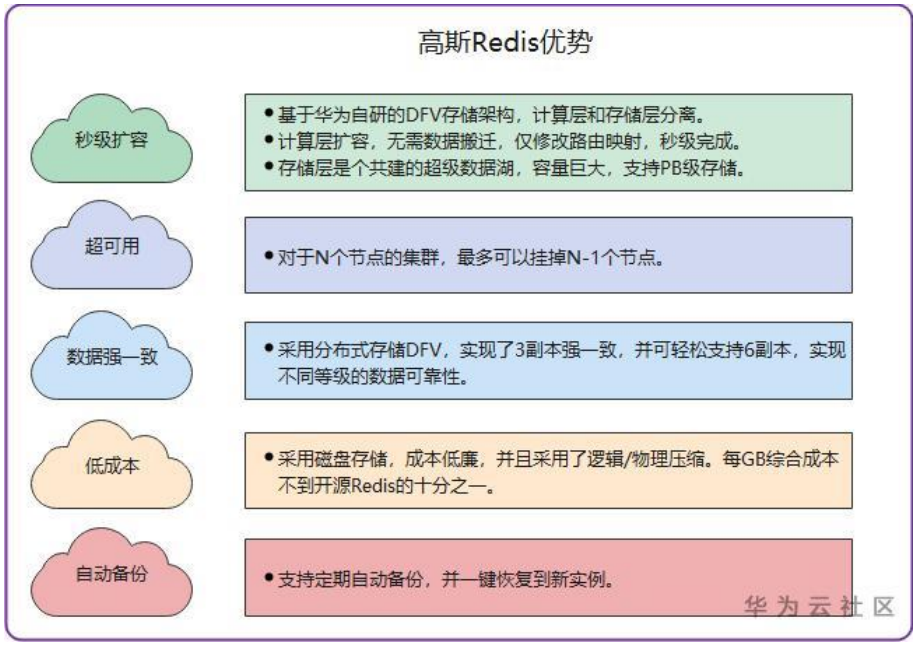
除此以外，必须考虑数据库系统的可用性、数据一致性、成本和备份恢复能力等情况：

- 可用性： 原生 Redis 若采用一主一备的集群模式，当一对主备节点下线，集群部分数据将不可用。

- 数据一致性：当主节点宕机，主备节点切换，数据存在没有完全同步的情况。
- 成本：原生 Redis 是一种内存型数据库，当内存容量扩展至 TB 级别，成本将非常昂贵。
- 备份恢复：需要人工连接数据库执行 SAVE 或 BGSAVE 命令，不能支持定期自动备份，在恢复到新实例时需要手动拷贝备份数据。

四、 是否有更好的解决方案?

在以上场景中，亟需一种能够存储和处理大规模 Stream 数据、鲁棒性强、且成本低廉的数据库系统。而 GaussDB(for Redis)（下文简称高斯 Redis）正是以上场景中一种很好的应用解决方案。高斯 Redis 是华为云数据库团队自主研发的兼容 Redis 协议的云原生数据库，该数据库突破原生 Redis 的内存限制，可轻松扩展至 PB 级存储，具有秒扩容、超可用、强一致和低成本等特点。



五、 总结

Redis Stream 可以广泛应用在即时通讯、智慧医疗、流量削峰等领域。在面对大规模的 Stream 数据时，原生 Redis 存在成本过高、容量太小、可用性差、数据不一致等问题，无法适用于海量消息队列的场景。与原生 Redis 相比，高斯 Redis 具有海量存储，低成本，可持久化等优点，可作为比原生 Redis 更理想的 Stream 队列承载方案。

【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专

业技能！**点击前往**

一场由 fork 引发的超时，让我们重新探讨了 Redis 抖动问题

一、 背景介绍

GaussDB(for Redis)服务团队在支撑某客户业务上云的过程中，发现一次由 fork 引发的时延抖动问题，本着对客户负责任的态度，我们详细探究了 fork 这个系统调用的性能影响，并且在最新的 GaussDB(for Redis)版本已解决了这个抖动问题，清零了内部的 fork 使用，与原生 Redis 相比，彻底解决了 fork 的性能隐患。

二、 问题焦点

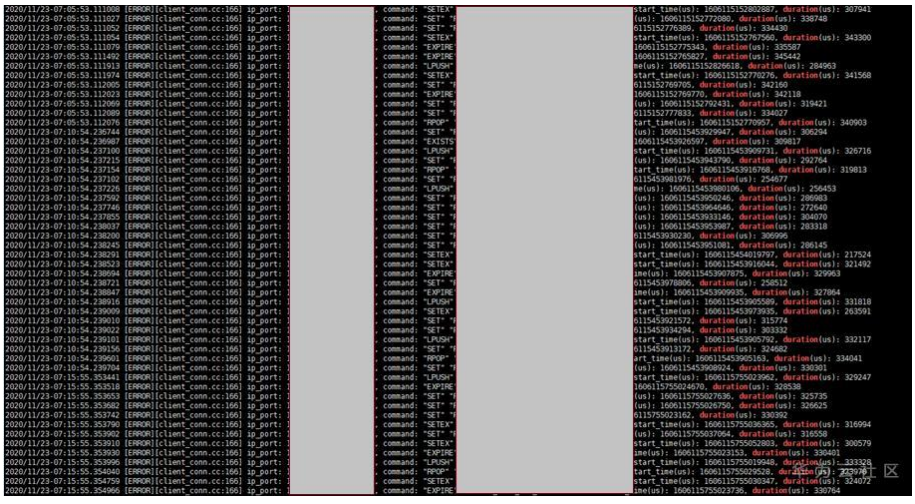
- 华为云 GaussDB(for Redis) 服务在某客户上云线调测过程中发现，系统上量后规律性的出现每 5 分钟 1 次的时延抖动问题。
- 华为云 GaussDB(for Redis)团队经过攻关，最终确认抖动原因是 fork 导致并解决了这个问题。而 fork 是开源 Redis 的一个重要依赖，希望通过本文的分享，能够帮助大家在使用开源 Redis 的时候，充分认识 fork 的影响，从而选择更优的方案。

三、 问题现象

某客户业务接入 GaussDB(for Redis)压测发现，每 5 分钟系统出现一次规律性的时延抖动：

- 正常情况消息时延在 1-3ms，抖动时刻时延达到 300ms 左右。
- 通常是压测一段时间后开始出现抖动；抖动一旦出现后就非常规律的保持在每 5 分钟 1 次；每次抖动的持续时长在 10ms 以内。

下图是从系统慢日志中捕获到的发生抖动的消息样例（对敏感信息进行了遮掩）：



四、 问题分析

排查抖动源：

- 由于故障的时间分布非常规律，首先排除定时任务的影响，主要包括：
- agent：和管控对接的周期性统计信息上报任务
- 内核：执行引擎（Redis 协议解析）和存储引擎（rocksdb）的周期性操作（包括 rocskdb 统计，wal 清理等）

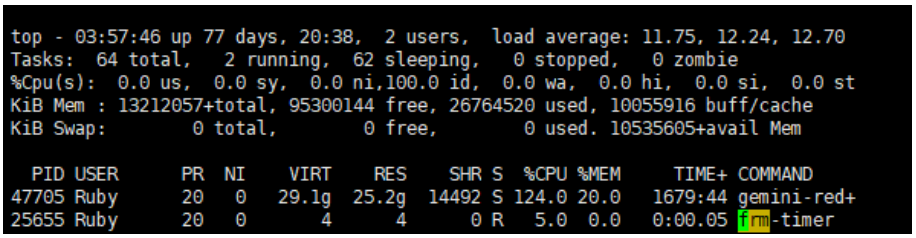
屏蔽上述 2 类定时任务后，抖动依然存在。

- 排除法未果后，决定回到正向定位的路上来。通过对数据访问路径增加分段耗时统计，最终发现抖动时刻内存操作（包括 allocate、memcpy 等）的耗时显著变长；基本上长出来的时延，都是阻塞在了内存操作上。

```
2020/11/18-06:07:11.615815 7fb7319e4700 [/memtable.cc:512] [MemTable::Add] memcpy duration:225539
```

（截图为相关日志，单位是微秒）

- 既然定位到是系统级操作的抖动，那么下一步的思路就是捕获抖动时刻系统是否有异常。我们采取的方法是，通过脚本定时抓取 top 信息，分析系统变化。运气比较好，脚本部署后一下就抓到了一个关键信息：每次在抖动的时刻，系统中会出现一个 frm-timer 进程；该进程为 GaussDB(for Redis)进程的子进程，且为瞬时进程，持续 1-2s 后退出。



- 为了确认该进程的影响，我们又抓取了 perf 信息，发现在该进程出现时刻，Kmalloc，memset_sse，memcopy_sse 等内核系统调用增多。从上述信息推断，frm-timer 进程应该是被 fork 出来的，抖动源基本可锁定在 fork frm-timer 这个动作上。

确定引发抖动的代码：

- 分析 frm-timer 的来历是下一步的关键。因为这个标识符不在我们的代码中，所以就需要拉通给我们提供类库的兄弟部门联合分析了。经过大家联合排查，确认 frm-timer 是日志库 liblog 中的一个定时器处理线程。如果这个线程 fork 了一个匿名的子进程，就会复用父进程的线程名，表现为 Redis 进程创建出 1 个名为 frm-timer 的子进程的现象。

- 由于 frm-timer 负责处理 liblog 中所有模块的定时器任务，究竟是哪个模块触发了上述 fork？这里我们采取了一个比较巧妙的方法，我们在定时器处理逻辑中增加了一段代码：如果处理耗时超过 30ms，则调用 std:: abort()退出，以生成 core 栈。

- 通过分析 core 栈，并结合代码排查，最终确认引发抖动的代码如下：

```
#define LOG_BAK_INTERVAL (3000)
dsw_u64 g_check_log_bak_tick=0;
#define LOG_BAK_SCRIPT_PATH "xx.sh"
void check_log_backup(dsw_u64 cur_timer_tick)
{
    //不满5分钟（3000个tick），则返回
    if (LOG_BAK_INTERVAL > cur_timer_tick - g_check_log_bak_tick)
    {
        return;
    }

    //满5分钟后，调用system，执行LOG_BAK脚本
    g_check_log_bak_tick = cur_timer_tick;
    char exec_cmd[256] = {0};
    sprintf_s(exec_cmd, sizeof(exec_cmd),
              "sh %s", LOG_BAK_SCRIPT_PATH);

    (void)system(exec_cmd);
}
```

华为云社区

上述代码是用来周期性归档日志的，它每 5 分钟会执行 1 次 system 系统调用来运行相关脚本，完成归档日志的操作。而 Linux system 系统调用的源码如下，实际上是一个先 fork 子进程，再调用 execl 的过程。

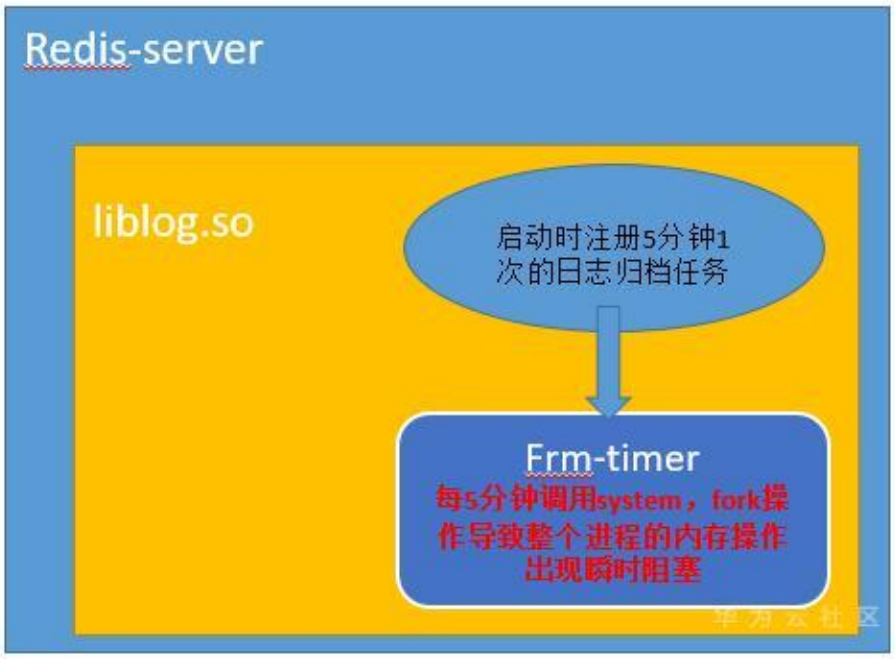
```
int system(const char * cmdstring)
 {
    pid_t pid;
    int status;
    if(cmdstring == NULL){
         return (1);
    }
    if((pid = fork())<0){
            status = -1;
    }
    else if(pid == 0){
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127); //子进程正常执行则不会执行此语句
       }
    else{
            while(waitpid(pid, &status, 0) < 0){

                if(errno != EINTER){
                    status = -1;
                    break;
                }
            }
        }
        return status;
}
```

华为云社区

分析至此，我们还需要回答最后一个问题：究竟是 fork 导致的抖动，还是脚本内容导致的抖动？为此，我们设计了一组测试用例：

- 用例 1：将脚本内容改为最简单的 echo 操作
- 用例 2：在 Redis 进程里模拟 1 个类似 frm-timer 的线程，通过命令触发该线程执行 fork 操作
- 用例 3：在 Redis 进程里模拟 1 个类似 frm-timer 的线程，通过命令触发该线程执行先 fork，再 excel 的操作

- 用例 4：在 Redis 进程里模拟 1 个类似 frm-timer 的线程，通过命令触发该线程执行 system 的操作



- 用例 5：在 Redis 进程里模拟 1 个类似 frm-timer 的线程，通过命令触发该线程执行先 vfork，再 excel 的操作

最终的验证结果：

- 用例 1：有抖动。
- 用例 2：有抖动。
- 用例 3：有抖动。
- 用例 4：有抖动。
- 用例 5：无抖动。

用例 1 结果表明抖动和脚本内容无关；用例 2、3、4 的结果表明调用 system 引发抖动的根因是因为其中执行了 fork 操作;用例 5 的结果进一步佐证了抖动的根因就是因为 fork 操作。最终的故障原因示意图如下：

进一步探究 fork 的影响：

- 众所周知，fork 是 Linux（严格说是 POSIX 接口）创建子进程的系统调用，历史上看，主流观点大多对其赞誉有加；但近年间随着技术演进，也陆续出现了反对的声音：有人认为 fork 是上个时代遗留的产物，在现代操作系统中已经过时，有很多害处。激进的观点甚至认为它应该被彻底弃用。（参见附录 1,2）

- fork 当前被诟病的主要问题之一是它的性能。大家对 fork 通常的理解是其采用 copy-on-wirte 写时复制策略，因此对其的性能影响不甚敏感。但实际上，虽然 fork 时可共享的数据内容不需要复制，但其相关的内核数据结构（包括页目录、页表、vm_area_struc 等）的复制开销也是不容忽视的。附录 1、2 中的文章对 fork 开销有详细介绍，我们这回遇到的问题也是一个鲜活的案例：对于 Redis 这样的时延敏感型应用，1 次 fork 就可能导致消息时延出现 100 倍的抖动，这对于应用来说无疑是不可接受的。

原生 Redis 的 fork 问题：

4.1 原生 Redis 同样被 fork 问题困扰（参见附录 3，4，5），具体包括如下场景：

1）数据备份

备份时需要生成 RDB 文件，因此 Redis 需要触发一次 fork。

2）主从同步

全量复制场景（包括初次复制或其他堆积严重的情况），主节点需要产生 RDB 文件来加速同步，同样需要触发 fork。

3）AOF 重写

当 AOF 文件较大，需要合并重写时，也会产生一次 fork。

4.2 上述 fork 问题对原生 Redis 的影响如下：

1）业务抖动

原生 Redis 采用单线程架构，如果在电商大促、热点事件等业务高峰时发生上述 fork，会导致 Redis 阻塞，进而对业务造成雪崩的影响。

2）内存利用率只有 50%

Fork 时子进程需要拷贝父进程的内存空间，虽然是 COW，但也要预留足够空间以防不测，因此内存利用率只有 50%，也使得成本高了一倍。

3）容量规模影响

为减小 fork 的影响，生产环境上原生 Redis 单个进程的最大内存量，通常控制在 5G 以内，导致原生 Redis 实例的容量大大受限，无法支撑海量数据。

解决方法

1. 修改日志库 liblog 中的周期性归档逻辑，不再 fork 子进程。
2. 系统排查并整改 GaussDB(for Redis)代码（包括使用的类库代码）中的 fork 调用。
3. 最终排查结果，实际只有本次的这个问题点涉及 fork。当前修改后即可确保 GaussDB(for Redis)的时延保持稳定，不再受 fork 性能影响。

注：GaussDB(for Redis)由华为云基于存算分离架构自主开发，因此不存在原生 Redis 的 fork 调用的场景。

五、 总结

本文通过分析 GaussDB(for Redis)的一次由 fork 引发的时延抖动问题，探究了 fork 这个系统调用的性能影响。最新的 GaussDB(for Redis)版本已解决了这个抖动问题，并清零了内部的 fork 使用，与原生 Redis 相比，彻底解决了 fork 的性能隐患。希望通过这个问题的分析，能够带给大家一些启发，方便大家更好的选型。

GaussDB(for Redis)免费体验： 企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，**点击体验**

【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！**点击前往**

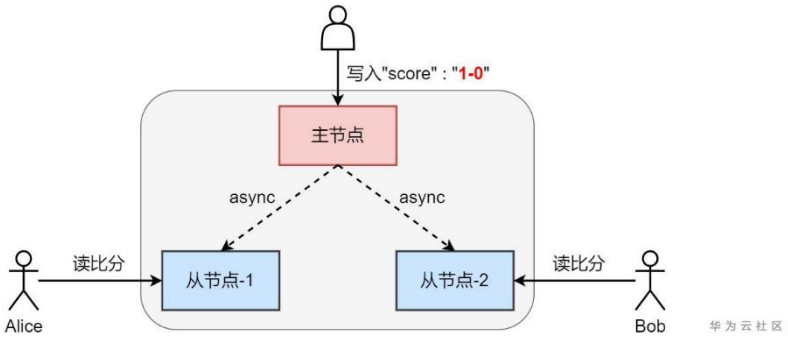
摒弃开源 Redis 异步复制机制，高斯 Redis 要做“强一致 KV 数据库”

清明刚过，五一假期就要来了。大好春光，不如去婺源看油菜花吧！小云迅速打开 APP 刷出余票 2 张，赶紧下单！唉，怎么又没抢到！转念一想倒也能理解：从勾选乘车人到正式下单，起码要 10 秒，真若是“见者有份”，恐怕这两个座位大家要挤挤共用了！每逢节假日，全国几百万小伙伴同时查票订票，12306 是如何保证余票显示准、车票不超卖的?

于是，按捺不住好奇心，笔者进行了一番深入研究。原来，问题背后隐藏着一个分布式数据库领域极其关键的技术——数据强一致性保障。

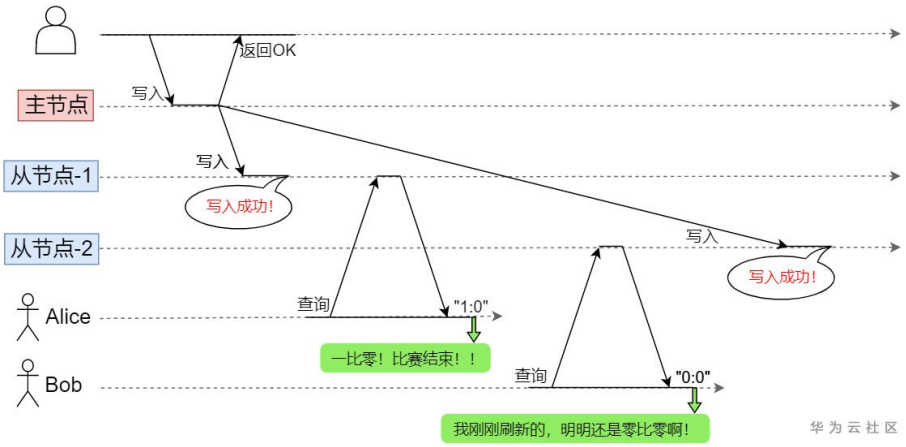
一、 什么是强一致?

在介绍概念之前，我们不妨先来模拟一场球赛直播。



假设笔者做了一款 APP，后台使用上图的主从数据库。比分写入主节点，从节点分担用户查询。比赛中，Alice 惊呼比赛结束，Bob 闻声刷新 APP，却显示比赛仍在继续！Bob 体验到了明显的数据不一致，于是默默给 APP 打了个差评……

那么，产生不一致的原因究竟是什么?



异步复制时，主节点不等待从节点写入就直接返回了。由于网络延迟等原因，从节点无法保证更新时间。Alice 和 Bob 明明在同时同地查询同一系统，得到正确结果却有先有后。其实这就是典型的弱一致性。

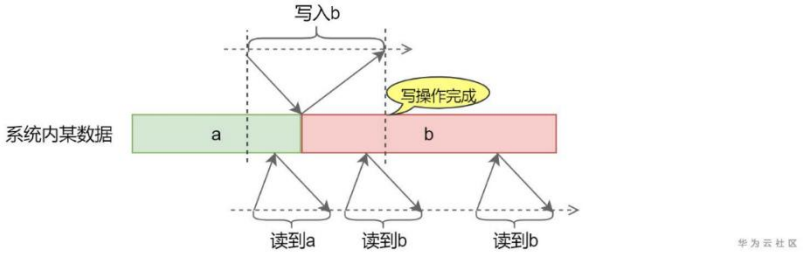
实际上，为解决单点故障、增强吞吐性能，分布式数据库内部都会对同一份数据进行复制，把冗余副本分散保存到不同节点上。简单的异步复制只能构建出弱一致系统，很难满足业务要求。

那么，究竟什么样的一致性才靠谱？有哪些类别？下面我们就来认识这个神秘家族!

1. 强一致性/线性一致性（Linearizability）

靠谱程度: ⭐⭐⭐⭐⭐

一致性的最高标准，实现难度最高。核心要求是：一旦写操作完成，随后任意客户端的查询都必须返回这一新值。以下图为例，一旦“写入 b”完成，必须保证读到 b。而写入过程中，认为值的跳变可能发生在某一瞬间，因此读到 a 或 b 都是可能的。



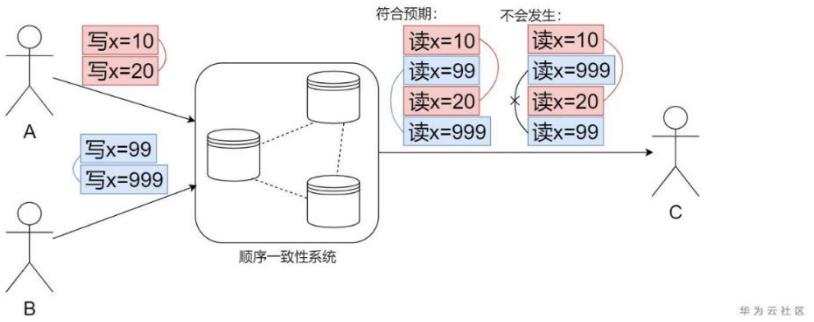
从业务角度来说，强一致性带来的体验简直可以用丝滑来形容！因为它内部的数据“仿佛”只有一份，即使并发访问不同节点，每个操作也都能原子有序。正因如此，强一致数据库在业务架构中往往被用在关键位置。

etcd 是强一致俱乐部里的元老。它基于 Raft 共识算法，真正实现了强一致，也因此在 Leader 选举、服务发现等场景起到重要作用。GaussDB(for Redis)作为一款分布式云数据库，凭借多年潜心打磨，也是强一致的代言人。

2. 顺序一致性（Sequential Consistency）

靠谱程度：⭐⭐⭐⭐

弱于线性一致，不保证操作的全局时序，但保证每个客户端操作能按顺序被执行。下图中，A 先写 x=10，后写 x=20；B 先写 x=99，后写 x=999。当 C 读取时，顺序一致性保证了 10 先于 20 被读到、99 先于 999 被读到。

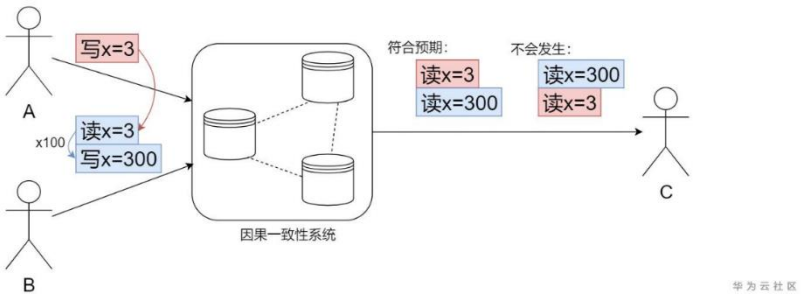


Zookeeper 基于 ZAB 协议，所有写操作都经由主节点协调，实现了顺序一致性。

3. 因果一致性（Causal Consistency）

靠谱程度：⭐⭐⭐

进一步放宽要求，只对并发访问中具有因果关系的操作保序。例如：



A 写入 3，B 读到后乘以 100 再更新它。在这个场景下，由于“A 写入 3”与“B 写入 300”有着明确因果关系，因果一致性保证 300 晚于 3 被读到。

因果一致性多用于各种博客的评论系统、社交软件等。自然，我们回复某条评论的内容，不应早于评论本身被显示出来。

4. 最终一致性（Eventual Consistency）

靠谱程度：⭐⭐

停止写入并等待一段时间，最终所有客户端都能读到相同的新数据，但具体时限不作保证。许多分布式数据库满足最终一致性，如 MySQL 主从集群等。

然而，这其实是一个非常弱的保证。由于不确定系统内部过多久才能收敛一致，在此之前，用户随时可能体验到数据不一致。因此最终一致性有天然的局限性，经常会给业务逻辑带来混乱。

5. 弱一致性（Weak Consistency）

靠谱程度：⭐

说它最为“厚脸皮”也不为过，因为它连数据写入后将来被读到都不能保证！弱一致性实现技术门槛低，应用场景也不多。严格来说，单纯的开源 Redis 主从集群就属于这一类别。

OK，一致性家族的各位成员已经跟大家打过照面。显然，一致性越强的数据库系统，能够支撑的业务场景越多。有的业务同学小声说，强一致技术再牛，可我业务简单，不用也没关系吧。实际上恰恰相反：

强一致不仅仅是技术问题，它更是一个不可忽视的业务需求、运维需求！

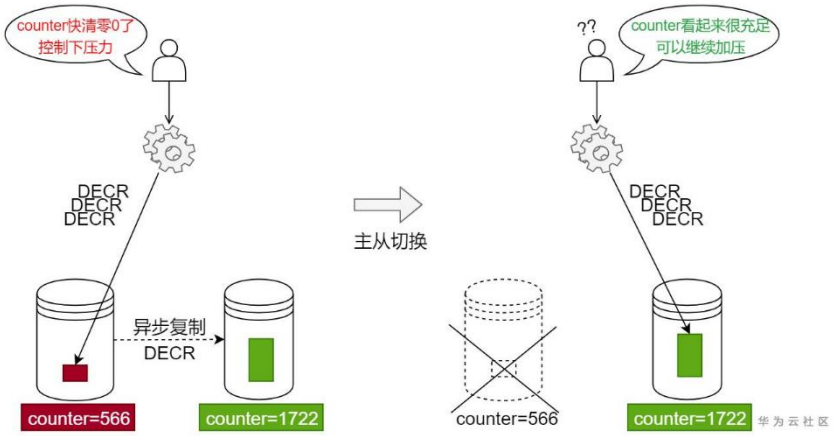
接下来我们就先来聊一聊：业务上那些只有强一致才能搞定的事儿！

二、 强一致是业务刚需

1. 计数器/限流器

计数服务是典型的强一致应用场景。电商在秒杀活动中，往往会搭建 Redis 主从集群给下层 MySQL 做缓存。因为要抗住超大流量，需要 Redis 的计数器功能做限流。简单讲，我们初始化 counter=5000。随后每次业务访问都执行 DECR 命令，当 counter 归零就阻塞后续请求。此外，每隔一个时间段重置 counter=5000，通过这样的手段来实现“细水长流”。

然而，完美的假设还不够！



开源 Redis 采用异步复制，如遇网络不畅，经常发生主节点复制 buffer 堆积。这将导致从节点 counter 偏大很多。此时，一旦主节点宕机，切换到从节点继续执行 DECR 命令，压力很容易超出阈值，全部落到下层脆弱的 MySQL，随时可能引起系统雪崩！

因此，在限流场景下，只有真正的强一致才能提供可靠的计数器。

2. Leader 选举

当业务部署的节点较多、可用性要求高时，往往要用到 Leader 选举。etcd 作为强一致 KV 存储，能完美 cover 这一场景。etcd 依赖两大功能实现 Leader 选举：

a) TTL：给 key 设置有效期，到期后 key 自动删除。
b) CAS：对 key 的原子操作。（这一功能只有强一致数据库才能实现）

使用 etcd 搭建 Leader 选举服务的设计如下：

a) 约定 key，用于选举时抢占。其 value 用于保存 Leader 节点名称。
b) 约定 TTL，用于给 key 设定有效期。
c) 启动时：每个参与节点尝试 cas create key&设置 TTL。在 etcd 集群强一致 CAS 机制保障下，只有一个节点能执行成功。该节点成为 Leader 并将名称写入 value；其余节点成为 Follower。
d) 运行中：每个节点定期 TTL/2 尝试 get key，将 value 与自身名称对比：

   - 如相同，说明已是 Leader，此后只需每隔 TTL/2 刷新 key 的 TTL 即可。

   - 如不同，说明是 Follower，接下来要每隔 TTL/2 执行 cas create key&设置 TTL。

e) 当 Leader 节点异常退出，无法刷新 TTL，key 会很快过期。此时，其余 Follow 之中便会有新的 Leader 产生。

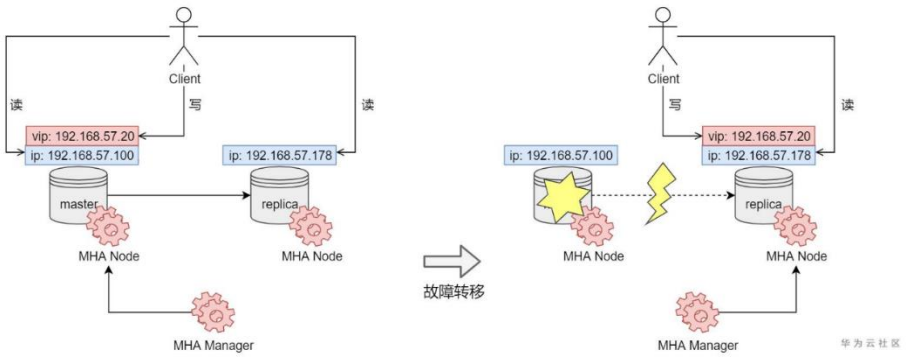
从原理上能看出，强一致能力是 Leader 选举的根基。类似的“刚需”业务场景还有很多，强一致不可或缺。

好了，业务上的事儿就聊到这里，接下来让我们听听运维怎么说。

三、 强一致为运维减负

1. 辅助组件架构复杂、问题难定位

后台架构中，MySQL 主从热备也是常见的部署方式。由于数据保存在本地磁盘中，当主库发生严重故障，仅仅依靠 MySQL 自身同步机制，主从切换后无法保证所提供数据与之前状态完全一致。于是出现了“重量级”的辅助组件——MHA（Master High Availability）。我们来看一下它的部署方式：



MHA 负责在故障转移过程中，帮助从库尽量追平主库最新状态，提供近似一致的数据。但这一能力需要额外的 Manager 节点，同时还要在每一个 MySQL 节点上部署 Node 服务。故障切换时，Manager 先为从库补充落后的数据，再通过切换 VIP 恢复用户访问，过程可能长达数十秒。

这样的 HA 系统部署和后期维护都很复杂。如未能顺利执行故障切换或发生数据丢失，运维面临的场面都将很棘手。其实运维同学何尝不希望手中的系统稳定运行呢？要是数据库自身能提供强一致保障，何苦再依赖复杂的辅助组件！

读到这里，对强一致的看法，相信各位读者心里已经有了自己的一杆秤。让我们再一次划重点：

强一致不仅仅是技术问题，它更是一个不可忽视的业务需求、运维需求！

从产品选型角度出发，开源 Redis 提供的一致性保证很弱。而 etcd 虽有强一致能力，但它单点写入性能不足，也未能提供 hash、sorted set、stream 等诱人的数据结构……纠结！

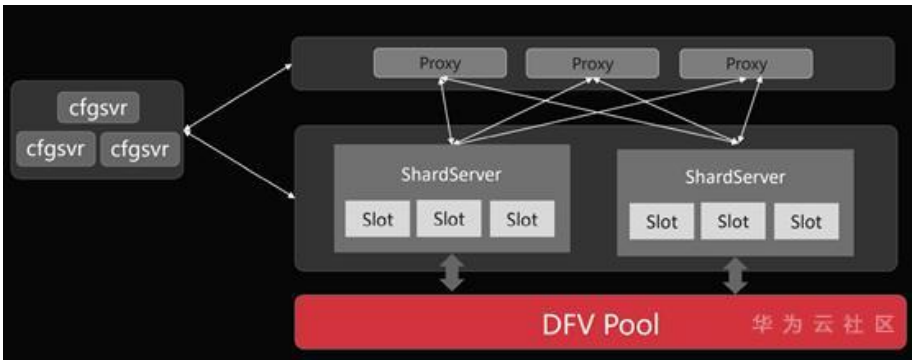
此时，有追求的读者会说——我全都要！

GaussDB(for Redis)应声而起——我，可以。

四、 GaussDB(for Redis)与强一致

自设计之初，GaussDB(for Redis)（后文简称高斯 Redis）给自己的定位就是“强一致 KV 数据库”，因此彻底摒弃了开源 Redis 的异步复制机制。借助华为云 GaussDB 系列先进的“存算分离”架构，将全量数据下沉到强一致存储层（DFV Pool），从核心技术上超越了传统开源产品的极限。

让我们来一起认识一下高斯 Redis 的强悍：



- **用户购买的实例作为一个整体，提供强一致 KV 存储。**

用户业务统一通过 Proxy 集群接入高斯 Redis，不用考虑内部复杂逻辑。多点并发访问实例，读写操作满足强一致性，再也不必担心开源 Redis 异步复制的不一致隐患。

- **计算层智能处理数据分片、动态故障转移，将数据全量下沉到共享存储池。**

cfgsvr 集群统一管理 ShardServer 节点，自动对海量数据进行分片。并能够在故障场景实现秒级接管，严格防止任何中间态下的数据不一致。

- **存储层通过 RDMA 高速网络实现高性能分布式数据持久化，三副本冗余保证强一致、零丢失。**

DFV Pool 是强一致、高性能的分布式存储系统。这是华为内部自研的公司级 Data Lake，它能够稳定支撑各类全栈数据服务。高斯 Redis 突破了开源 Redis“小格局”的内存架构，将数据全量下沉，基于 DFV Pool 强大的一致性保障能力，给用户业务带来更广阔的拓展空间。

GaussDB(for Redis)免费体验： 企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，**点击体验**

五、 结语

试想，当处在关键位置的数据库“不给力”，业务层就要忙于为系统添加复杂、易出错的一致性保障逻辑。与此同时，运维还要时刻担心故障引发的数据落后问题……这样的系统真的“香”吗?

专业的事情交给专业的团队来做!

华为云 NoSQL 航道旗舰——GaussDB(for Redis)自研发初期就持续关注数据强一致性设计。借助 GaussDB 系列先进的强一致存储池 DFV Pool，GaussDB(for Redis)始终如一，为用户提供真正强一致的海量 KV 存储解决方案。

【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！**点击前往**

常规计数器与基数计数器场景下，高斯 Redis 如何实现计数

一、 背景

当我们打开手机刷微博时，就要开始和各种各样的计数器打交道了。我们注册一个帐号后，微博就会给我们记录一组数据：关注数、粉丝数、动态数…；我们刷帖时，关注每天的热搜情况，微博需要为每个热搜记录一组搜索量。在这一串数据后面，是一个个计数器在工作。

计数器可以分为常规计数器和基数计数器，对于常规计数器，只需要对计数器进行简单的增减即可；对于基数计数器，需要对元素进行去重，比如统计搜索量时，需要保证每个用户的多次搜索只统计一次。对于这两种需求，Redis 都有对应的数据类型进行统计。然而开源 Redis 是一个弱一致性的数据库，在特定的场景下，弱一致的计数不能满足业务需求，为此，我们需要一个强一致的数据库进行计数。

GaussDB(for Redis)（下文简称高斯 Redis），是华为自研的强一致、持久化 NoSQL 数据库，兼容 Redis5.0 协议。本文将介绍常规计数器与基数计数器的应用场景及使用高斯 Redis 实现计数。

二、 常规计数器

1. 如何使用 Redis 进行常规计数

Redis 实现常规计数器有两种数据类型适合：String 和 Hash。

1）使用 string 计数

当我们需要维护的计数器数目较少，比如统计网站的注册用户数时，适合使用 String 类型的计数器。Redis 提供的 Incr 和 Decr 命令分别对 String 类型的 key 值进行增一与减一操作：

```
127.0.0.1:6379> SET counter 100
OK
127.0.0.1:6379> INCR counter
(integer) 101
127.0.0.1:6379> DECR counter
```

```
(integer) 100
```

除 Incr 与 Decr 命令外，Redis String 类型还提供 Incrby 与 Decrby 命令，语法格式为：

- incrby： INCRBY key count

将 key 增加 count，count 可正可负，返回 key 的结果：

```
127.0.0.1:6379> INCRBY counter 10
(integer) 10
127.0.0.1:6379> INCRBY counter -20
(integer) -10
```

- decrby： DECRBY key count

将 key 减少 count，count 可正可负，返回 key 的结果：

```
127.0.0.1:6379> DECRBY counter 10
(integer) -10
127.0.0.1:6379> DECRBY counter -20
(integer) 10
```

2）使用 Hash 计数

需要维护多个密切关联的计数器时，可以使用 Hash 结构进行计数。比如说，当我们注册一个微博账号时，微博会给每个用户记录一些用户数据，如粉丝数、关注数等，这些数据都绑定到对应用户上，因此可以将这组计数器记录在同一个 Hash key 中，使用 hincrby 命令，语法格式为：

- hincrby： HINCRBY key filed count

将 Hash key 的 filed 增加 count，count 可正可负，返回对应 field 的结果：

```
127.0.0.1:6379> HGET userid field
(nil)
```

```
127.0.0.1:6379> HINCRBY userid field 1
(integer) 1
127.0.0.1:6379> HINCRBY userid field -1
(integer) 0
127.0.0.1:6379> HGET userid field
"0"
```

2. 常规计数器使用场景

常规计数器的使用场景很广泛，对于社交产品，用户的粉丝数、关注数，帖子的点赞数、收藏数…；对于视频网站，需要统计视频的播放次数（PV 统计，Page View）；对于电商秒杀，需要统计商品数量并进行流量控制。在并发量高的情况下，Redis 的性能优势明显，非常适合以上场景。

以电商秒杀业务为例，为了处理高并发读写，通常在 MySQL 上层部署 Redis 作为缓存。为了抗住大流量，使用计数器作限流。比如，当我们想控制每秒 1 万次请求时，可以初始化一个 counter=10000，随后每次请求过来，都对 counter 减一，当 counter 归零后，阻塞后续的请求。每隔一段时间，重置 counter=10000，以此保证大流量不会冲击底层的 MySQL。

三、 基数统计：HyperLogLog 的原理及使用

基数计数（cardinality counting）是指在一个数据集合中，统计不重复元素的个数，是实际应用中一种常见的场景。比如统计一段时间内访问某个网站的用户数，网络游戏的日活用户数量等。

在数据量较小情况下，我们可以把所有数据保存下来进行去重统计。Redis 中，可以使用 Set 与 Zset 将数据保存下来，然后统计集合中的元素数量。而当数据量较大时，该方法会消耗较大的存储空间，需要考虑其它的算法。

考虑一种情况，当我们登录微博时，微博会记录我们的登录情况，并统计每天有多少活跃用户。很显然，我们不需要也不应该记录活跃用户的 ID，并且，少量误差对活跃用户数量的统计使用影响不大，这种场景下，我们可以使用 HyperLogLog 进行计数。
HyperLogLog 是一种使用极少内存实现巨量统计的计数算法，非常适合大数据场景的基数估计，在 Redis 中被实现为一种数据类型。

1. HyperLogLog 原理介绍

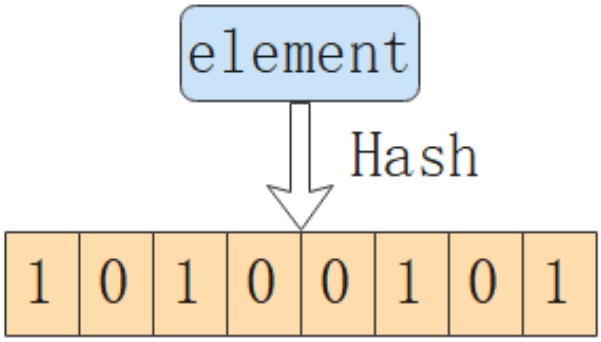
从伯努利试验到基数计数

HyperLogLog 是一种基数估计算法，其思想来自于伯努利过程。

简单来说，伯努利过程就是一个抛硬币的过程。抛一次硬币，结果为正面或者反面的概率都是 1/2。记正面为 1，反面为 0，如果抛硬币多次，直到出现第一次正面时停止，记为一次投掷试验，并且得到一个投掷结果的序列，比如“001”，我们可以知道，这个序列出现的概率是 。

反过来，如果我们持续进行投掷试验，当出现第一次“001”序列时，我们可以简单估算出，我们投掷试验次数为 8（事实上，这是一个极大似然估计）。

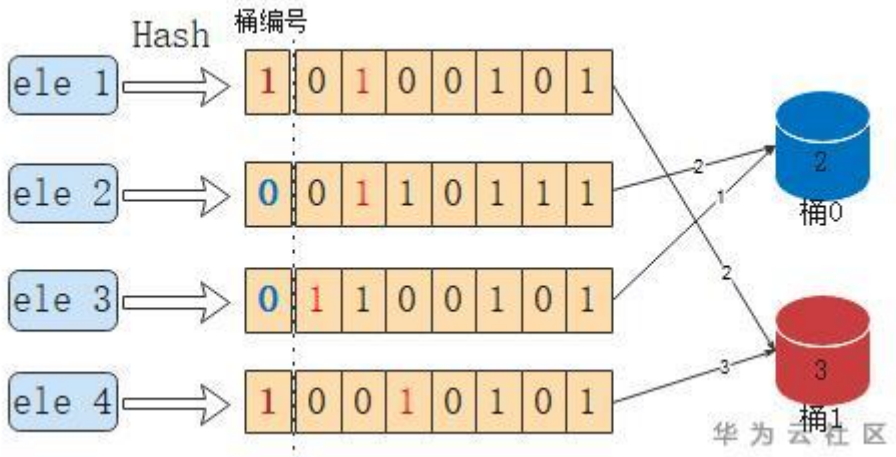
HyperLogLog 的原理就是将每个元素视为一次投掷试验，通过记录试验的最大投掷次数对元素的数量进行估计。当我们向集合中每插入一个元素，视为做了一次投掷试验，相同的元素对应一个投掷结果的序列。为了将每一个元素转化成一个“01”序列，我们可以使用一个哈希函数进行转换：



这里，我们有了一个简单的估计算法。我们只需要记录哈希结果中第一个“1”出现的位置的最大值即可，但很明显，当数据量较小时，这样一个估计值误差会很大，而且单个元素的对估计值的影响不平滑。

分桶平均减小误差

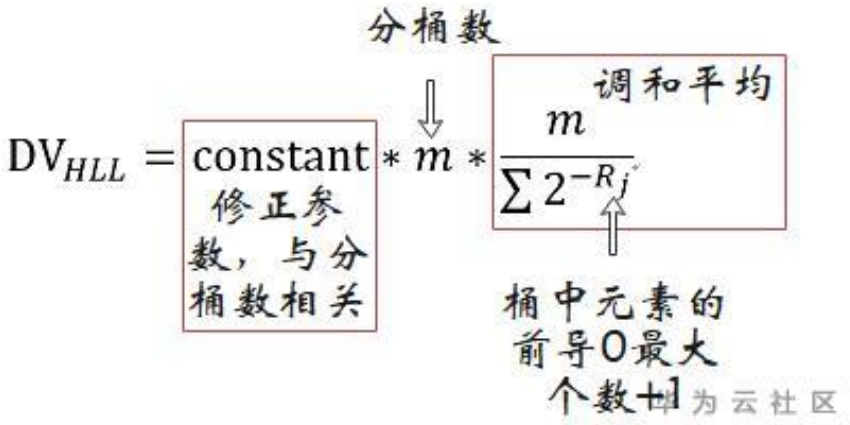
为了减小单一估计量的影响，HyperLogLog 使用分桶多次试验的方法减小误差。方法是将哈希后的 bitmap 中前若干位当成桶的编号，剩余位当成试验结果。



对于每个桶中的结果，计算其调和平均值获取基数估计值（相比算术平均，调和平均数能够有效改善基数较小情况下极值影响过大的问题）：

2. Redis 中的 HyperLogLog

Redis 将 HyperLogLog 实现成一种数据类型，对于每个元素，Redis 将其 Hash 成 64 位的二进制串，用低 14 位用来表示桶的下标（所以桶的个数为 1<<14=16384），剩余的位用来模拟伯努利分布，每个桶需要 6 个 bit；最多能够对 2 的 64 次方个元素进行统计，内存占用约 12 k；其标准误差为 0.81%。



Redis 支持的 HyperLogLog 命令只有 3 个，pfadd，pfcount，pfmerge，其语法如下：

- **pfadd**：将所有元素参数添加到 HyperLogLog 数据结构中

语法：PFADD key element1 [element2…]

如果至少有一个元素被添加返回 1，否则返回 0

如果没有指定 element，则创建 hyperloglog key

```
127.0.0.1:6379> pfadd key1 ele1 ele2
(integer) 1
127.0.0.1:6379> pfadd key1
(integer) 0
127.0.0.1:6379> pfadd key2
(integer) 0
```

- **pfcount**：返回给定的 HyperLogLog 的基数估计值

语法：PFCOUNT key1 [key2 …]

返回对应 HyperLogLog 的基数值，多个 key 时，返回多个 key 的合并后的基数值。

```
127.0.0.1:6379> pfcount key1
(integer) 0
127.0.0.1:6379> pfadd key1 ele1 ele2
(integer) 1
127.0.0.1:6379> pfadd key2 ele1 ele3
(integer) 1
127.0.0.1:6379> pfcount key1
(integer) 2
```

```
127.0.0.1:6379> pfcount key1 key2
(integer) 3
```

- **pfmerge**：将多个 HyperLogLog 合并为一个

语法：PFMERGE destkey sourcekey1 [sourcekey2 …]

将 sourcekey 与 destkey 合并，当 destkey 不存在时，会创建 destkey

返回 OK

```
127.0.0.1:6379> pfadd key1 ele1 ele2
(integer) 1
127.0.0.1:6379> pfadd key2 ele1 ele3
(integer) 1
127.0.0.1:6379> pfcount key3
(integer) 0
127.0.0.1:6379> pfmerge key3 key1 key2
OK
127.0.0.1:6379> pfcount key3
(integer) 3
```

3. HyperLogLog 的适用场景

HyperLogLog 作为一种计算大数据量的基数统计算法，在统计注册用户数，每日访问 IP 数，实时统计在线用户数等场景可以大显神威。

- 统计网站的 UV(unique visitor)

对于一个网页，我们想要知道这个网页的受关注程度，可以统计一下有多少用户（IP）点击了这个网页。为此，我们给每个时间段设置一条记录，比如，127.0.0.1 这个 IP 在 2021 年 1 月 1 日 1 点的时候访问了网页：

```
pfadd key_prefix_2021010101 "127.0.0.1"
```

当需要统计这一天 0-1 点这一个小时一共有多少 IP 访问了这个网页时：

```
pfcount key_prefix_2021010101
```

需要统计上午 8 到 12 点的网页访问情况：

```
pfcount key_prefix_2021010109 …… key_prefix_2021010112
```

一天结束了，需要统计并保存这一天访问情况：

```
pfmerge key_prefix_2021010101 ..... key_prefix_2021010124
```

对于一个热门的网页，这样一个计数的方式显然能够极大的节约存储空间。

- **用户画像**

用户画像是根据用户在互联网上留下的各种数据，给用户贴上一系列的标签，比如用户的性别，年龄，爱好等。在进行数据分析时，可以使用 HyperLogLog 进行数据的保存与分析。

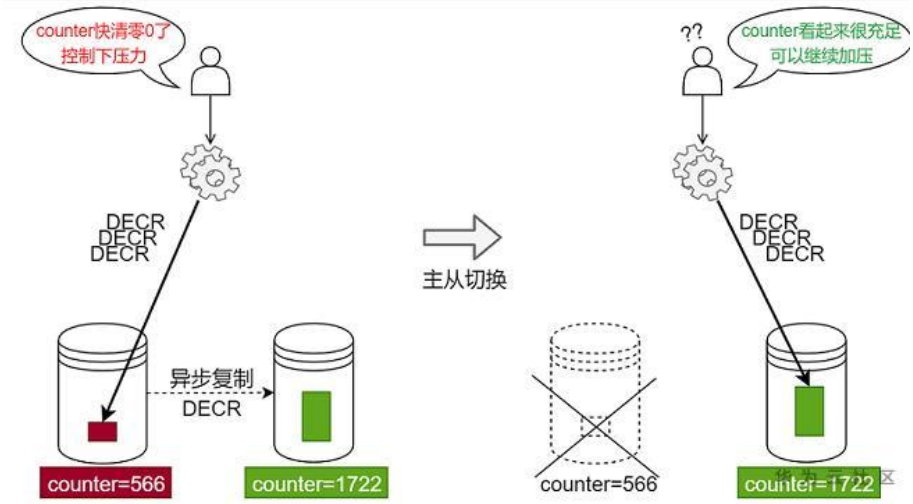
1. 对于每个标签，创建 hyperloglog key 值保存数据，如：man, woman, basketball…等，对于每个需要记录的值，都需要创建一个 key 进行记录。
2. 每多一个用户时，向所有记录的 key 里使用 pfadd 添加元素。
3. 进行数据分析时，使用 pfcount 将需要分析的数据进行统计。

四、 高斯 Redis 在计数上的优势

1. 开源 Redis 的问题

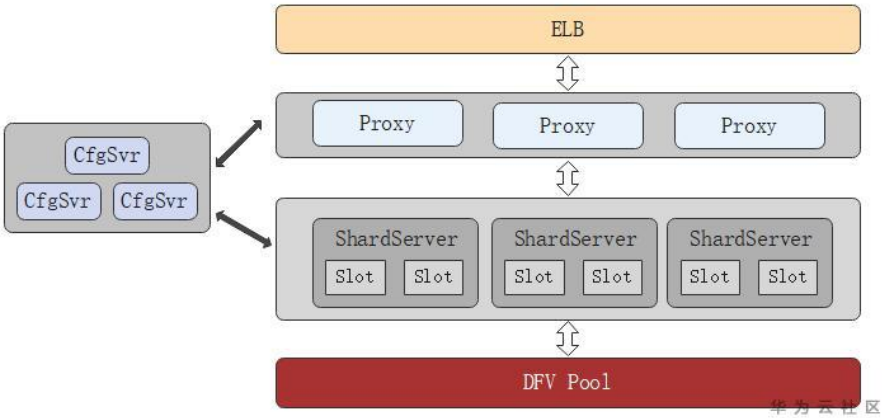
生产环境中，为避免单点故障，增强数据库可用性，Redis 通常将数据复制多个副本，保存在不同的服务器上；在大量并发请求过来时，为了尽可能利用主从节点的服务器资源，可以采用主写从读的方式。由于 Redis 的主从同步是异步的，当主节点写入数据后，从节点不保证立刻更新数据，如果此时读取数据，读到的就是过期的旧数据，产生数据不一致问题。

当主节点故障宕机后，数据不一致的问题会更严重。主节点故障后，哨兵节点会将从节点提升为主，原主节点上堆积的数据 buffer 就彻底丢失了。在电商秒杀业务中，如果发生主节点复制 buffer 堆积，导致从节点与主节点的 counter 偏大很多，一旦此时主节点宕机，发生主备倒换后，容易导致流量压力超出阈值，大量数据可能会将 MySQL 压垮，导致系统不可用。

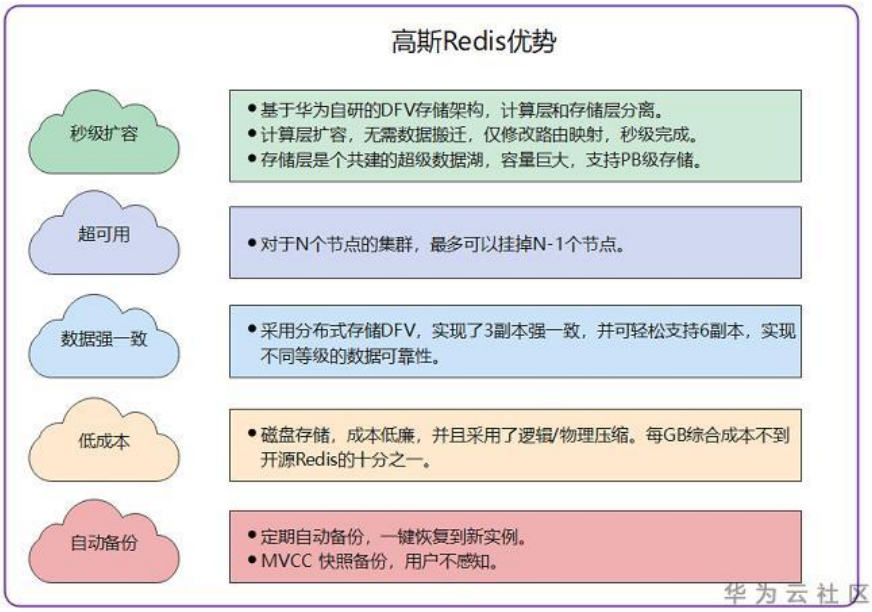


2. 高斯 Redis 如何解决

高斯 Redis 借助高斯品牌的“存算分离”架构，将全量数据下沉到强一致存储层（DFV Pool），彻底摒弃了开源 Redis 的异步复制机制；计算层将海量数据进行分片，在故障场景下，自动进行接管，实现了服务的高可用。



存储层 DFV Pool 是华为内部自研的公司级 Data Lake，是分布式、强一致、高性能的先进架构。底层实现 3 副本强一致的存储，保证了在任何时间点的数据强一致，故障情况下数



据不丢失，对于秒杀等业务满足计数的绝对精确。此外，借助存算分离架构，**高斯 Redis** 还拥有低成本、大容量、秒扩容等优势。

GaussDB(for Redis)免费体验： 企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，[点击体验]

五、结语

GaussDB(for Redis) 在社区版 Redis 的基础上，结合华为自研强一致存储 DFV Pool，具有强一致、秒扩容、超可用、低成本等优势，保证了计数的准确性、可靠性。

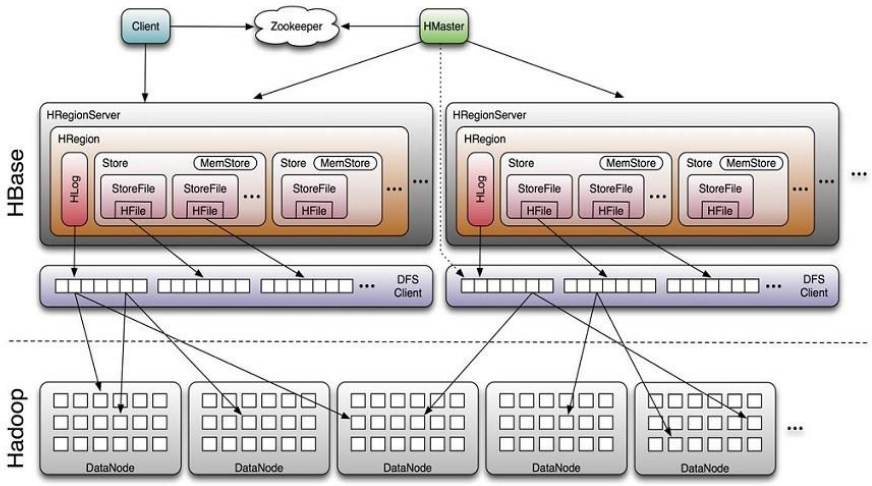
【 数据库论坛 】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！**点击前往**

高斯 Redis 破解 HBase 的阿克琉斯之踵

引言：HBase 是一个分布式的、面向列的开源数据库，基于 Hadoop 生态圈，在 NoSQL 蓬勃发展的今天被国内外众多公司选择，应用于现代互联网系统的不同业务。本文简要描述了 HBase 的基本架构和使用场景，重点分析了 HBase 关键特性在此场景下的表现，以及 HBase 在使用上尚存的痛点；同时介绍了华为自研的强一致、持久化 NoSQL 数据库 GaussDB(for Redis)（下文简称高斯 Redis）在以上场景中的表现，以及对于 HBase 痛点问题的改善。

一、 HBase 系统简述



HBase 的物理结构主要包括 ZooKeeper、HMaster、RegionServer、HDFS 等组件。ZooKeeper 用以实现 HMaster 的高可用、RegionServer 的监控、元数据的入口以及集群配置的维护等工作。HMaster 的作用是维护整个集群的 Region 信息，处理元数据变更及负载均衡工作。RegionServer 是直接处理用户读写请求的节点，实际处理所分配 Region 的读写、分裂等工作，并使用 WAL 实现容错机制。HDFS 提供最终的底层数据存储服务，提供元数据和表数据的底层分布式存储服务，同时利用数据多副本，保证的高可靠和高可用性。

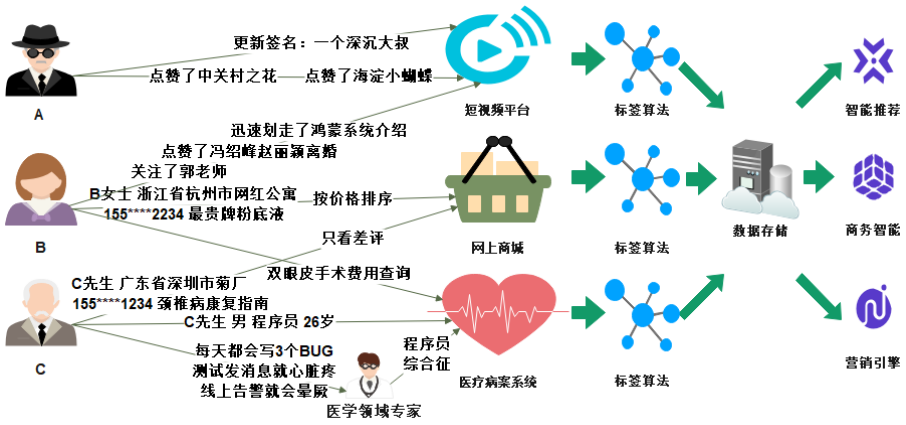
	ColumnFamily1		ColumnFamily2	
	Column11	Column12	Column22	Column23
RowKey	Cell1	Cell2	Cell3	Cell4

在逻辑结构中，RowKey 是表的主键，并按照字典序进行排列，HRegion 达到一定大小后也会按照 RowKey 范围进行裂变。ColumnFamily 在纵向上对表进行切分，将多个 Column 分成一组进行管理，在 HBase 中，ColumnFamily 是表的 schema 而 Column 不是。Cell 则是保存的具体 value，在 HBase 中，所有的数据都是以字节码的方式进行存储。

二、 HBase 大显身手

1. 标签数据的存储

标签数据是稀疏矩阵的代表，描述了实体的各类属性，主要应用于智能推荐、商务智能或营销引擎等领域。



三个不同的用户在同一公司旗下的不同 APP 中留下了大量的行为数据，这些数据中包含了直接填写的用户资料、使用 APP 的具体行为以及领域专家对某些现象的标记，通过后台的标签算法可以得到这样的数据：

姓名	性别	年龄	职业	感兴趣领域1	感兴趣领域2	不感兴趣领域	地址	电话	消费品	消费习惯	预期消费领域	健康状况
A	85%可能为男			美女								
B	99%可能为女			搞笑	明星	科技	浙江省杭州市网红公寓	155****2234	最贵牌粉底液	倾向高消费	医疗美容	
C	100%可能为男	26	程序员				广东省深圳市菊厂	155****1234	颈椎病康复指南	口碑敏感		程序员综合征

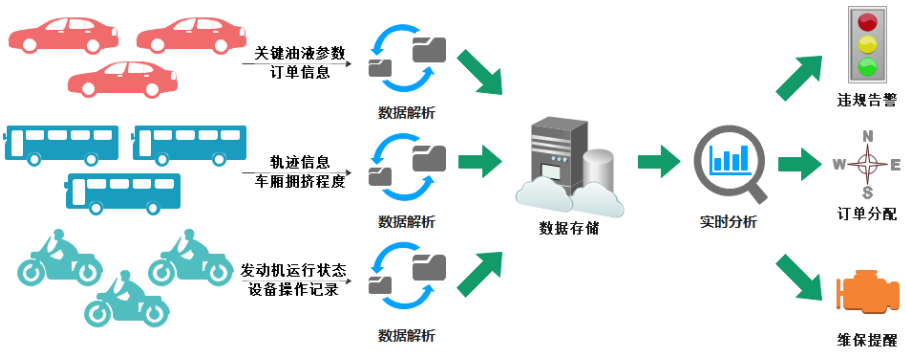
我们能发现，对用户行为采集存在局限性，因此所能得到的标签种类各不相同，表中大量的数据项只能被置空，也就是所谓的稀疏矩阵。而且随着用户更深度的使用 APP，可以预见到，对用户感兴趣领域/不感兴趣领域会逐渐被发掘，那么表的列也会随之增加。

这样的特点对于 MySQL 是灾难性的，这是因为在 MySQL 建表时就必须定义表结构，属性的动态增删是巨大的工作量，同时大量 NULL 值的存储会导致存储成本变得难以接受。

但是使用 HBase 存储时，未指定 value 的列不会占用任何的存储空间，因而可以将有限的资源高效利用，且 HBase 表在创建时只需指定 ColumnFamily，而对于 Column 的增删极为容易，有利于应对未来属性的扩张。

2. 车联网数据的收集

车联网系统是利用车载设备收集车辆运行时产生的各项数据，通过网络实时上传，在平台进行动态分析和利用。

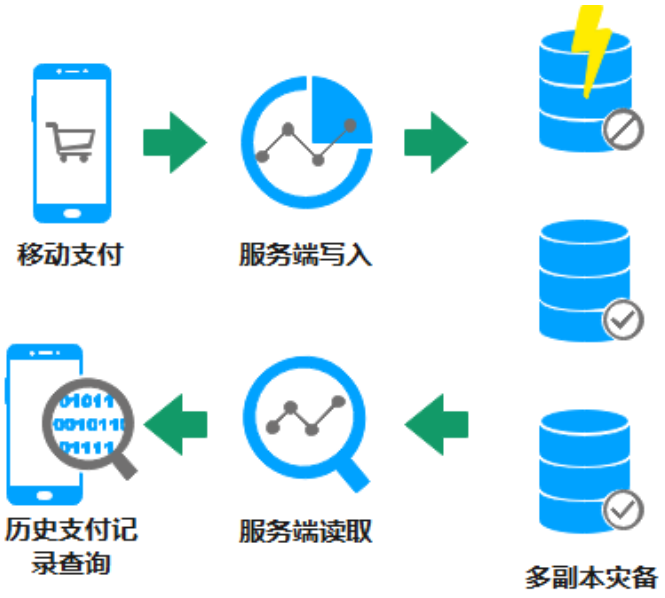


我们可以发现，车联网系统所面对的数据特点是大量车辆终端高并发的不间断写入 TB 级甚至 PB 级的数据，而且对于实时分析来说，为了保证分析结果的时效性，又要求查询的低时延响应。

HBase 采用 LSM 存储模型，可以从容应对高并发写入的场景，同时也能保证读时延在可接受的范围内。同时 HBase 具有良好的水平扩展能力。通过增减 RegionServer 来实现对存储容量动态调整，满足对使用成本的要求。

3. 交易记录的保存

在移动支付领域，保证历史交易记录等敏感信息的安全性是一个重要的话题。当数据中心遭遇自然灾害、外部攻击时，必须保证这些信息不丢，而且从业务角度要保证 RTO 尽可能短、RPO 尽可能为 0。



HBase 基于底层的 HDFS 作为存储系统，HDFS 实现了三副本策略，按照一定的规则将副本放在不同的节点或机架中，本身具有较高的容灾能力。在工程实践中，也产生了 Region replica、主备集群、互备双活等策略来尽可能进行灾备并保证高可用。

三、 HBase 并不全能

从上文三个例子可以看出，HBase 基于其本身的设计，在稀疏矩阵的存储、抗高并发大流量写入、高可用和高可靠场景下表现得相当优秀，但这并不意味着 HBase 可以没有任何弱点的适应所有场景。

HBase 的阿克琉斯之踵

a） 朱丽叶暂停

Java 系统绕不开 Full GC 的讨论。HBase 在 Full GC 造成 STW 时，ZooKeeper 将收不到来自 RegionServer 的心跳，进而将此节点判定为宕机，由其他节点接管数据，当 Full GC 结束后，RegionServer 为防止脑裂而主动自杀，称之为朱丽叶暂停。这类问题一般需要资深的 java 程序员根据业务场景进行细致的 GC 策略调优才能尽可能避免。

b） 数据类型少

HBase 支持存储的类型是字节数组，在使用中需要将字符串、复杂对象、甚至图像等数据转化为字节数组进行存储。但是这样的存储只能表示松散的数据关系，对于集合、队列、Map 等数据结构或数据关系，则需要开发人员编码实现转换逻辑才能进行存储，灵活性较差。

c） 性能之瓶颈

HBase 是按照 RowKey 的字典序分割为 Region 进行存储的，不佳的 RowKey 设计方案会造成负载不均，请求大量打到某一个 Region 形成热点，那么所在 RegionServer 的 IO 有可能被打爆。

RegionServer 掉线后，需要由 ZooKeeper 发现节点宕机，将其负责的数据移动到其他节点接管，并对 meta 表中的 Region 信息进行修改。在此过程中，RegionServer 上的数据将变得不可用，对于这部分数据的请求会被阻塞。

Redis 的伊卡洛斯之翼

a） 开源 Redis 的良好表现

开源 Redis 的特性在一定程度上解决了 HBase 的痛点问题，因其具有以下优点：

i.更丰富的数据类型

Redis 5.0 协议中包含了 String、List、Set、ZSet、Hash、Bit Array、HyperLogLog、Geospatial Index、Streams 九种数据类型，以及建立在这些数据类型上的相关操作。与 HBase 的单一数据类型相比，Redis 给了开发人员更多的选择空间来表达数据和数据间的相互关系。

ii.纯内存的丝滑感受

开源 Redis 的本质是一个 key-value 类型的内存数据库，整个数据库都加载在内存中进行操作。这也就意味着 Redis 的响应速度和处理能力远超过需要进行磁盘 IO 的 HBase，目前大量的测试结果都表明，开源 Redis 的性能可以达到每秒 10 万次读写。

b） 开源 Redis 的显著弱点

纯内存的操作也使得开源 Redis 有无法避免的弱点，主要体现在以下两方面：

i.大数据量下的噩梦

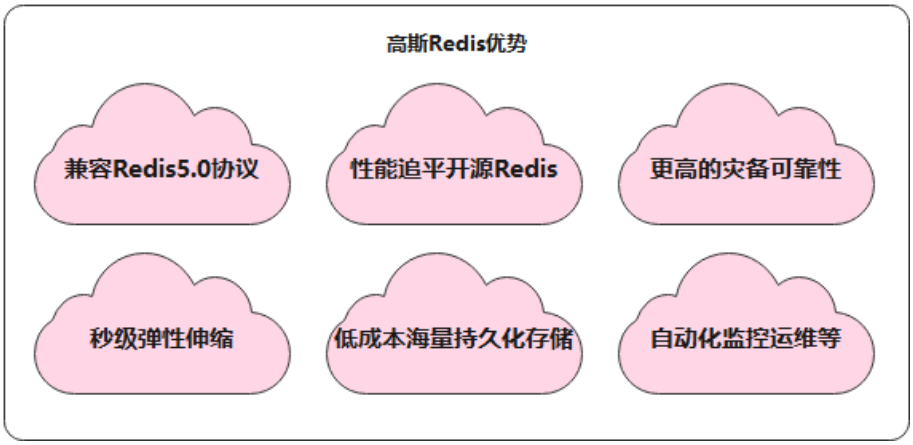
当数据量持续增大时，有限的内存成为使用限制。此时必须使用更大容量的内存才能完成数据的全量加载，而内存价格远高于磁盘价格，会导致使用成本的激增。同时常见的服务器内存多是 GB 级，也严重限制了开源 Redis 在高量级数据库领域的竞争力。

i.断电后该何去何从

纯内存操作的另一弊端是宕机后数据会全部丢失。现有的解决方案是使用 AOF 或 RDB 的方式将数据持久化，进程重启后可以在内存中将数据恢复。但这两种方式并不完备，AOF 是执行命令的集合，因此恢复速度相对较慢；RDB 是定期 dump 内存数据，因此存在数据丢失的风险。除此之外，在最坏场景下需要预留一半内存，降低了内存的使用率。

四、 高斯 Redis：成年人不做选择题

HBase 和开源 Redis 各有所长，这时一句熟悉的话在脑海中浮现：小孩子才做选择题，成年人当然是全都要，高斯 Redis 的兼具二者优点，更好的满足了对数据库服务的需求。



- **兼容 Redis5.0 协议**

延续开源 Redis 的丰富数据类型，为描述数据和数据关系提供更多选择。例如在稀疏矩阵场景使用 Hash 类型，甚至无需定义 HBase 表 ColumnFamily，可以更灵活的进行数据组织。

- **性能追平开源 Redis**

参考[华为云高斯 DB(for Redis)与开源 Redis 集群性能对比]可以看出，高斯 Redis 与开源 Redis 的性能几乎相同，在大流量高并发的场景中，可以提供比 HBase 更好的读写表现。

- **更高的灾备可靠性**

高斯 Redis 基于华为自研的分布式、强一致数据湖 DFV 构建的存储层，在部分局点的已经上线了 3AZ 特性，AZ 间做到风火水电的物理隔离，一个 AZ 的故障不会影响到其他 AZ，与 HBase 相比更好保证了关键数据的可靠性。

- **秒级弹性伸缩**

高斯 Redis 使用存算分离架构，数据下沉至存储池，计算节点扩缩容仅修改映射无需搬迁数据，实现秒级平滑伸缩，不存在 HBase 在 Region 上下线时出现的数据不可用问题。

- **低成本海量持久化存储**

全量数据经过逻辑和物理压缩，将落入共享存储池 DFV 持久化存储，无宕机数据丢失问题，每 GB 的综合成本不到开源 Redis 的十分之一。实际应用中可根据业务需要随时对 DFV 容量进行扩容，不存在开源 Redis 存储受限的问题。

- **自动化监控运维等其他优势**

高斯 Redis 配套全面的监控系统可对请求时延等关键性能指标可视化监控，同时可实现故障节点自动摘除、平滑移动、自动告警、自动恢复。此外，高斯 Redis 利用 hash 策略对数据进行均衡，与 HBase 相比更好的避免了热点问题，而且不存在 Full GC 烦恼。

GaussDB(for Redis)**免费体验：**企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，**点击体验**

五、 结语

GaussDB(for Redis)在兼容 Redis5.0 协议的基础上，兼具开源 Redis 和 HBase 各自优点，结合华为自研 DFV 存储的相关特性，规避 HBase 和开源 Redis 在典型场景下的弱点，提供成本更低、性能更好、灵活性更强的数据库服务。

【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！**点击前往**

GaussDB(for Redis)与存算分离——中国系统架构师大会 SACC 分享

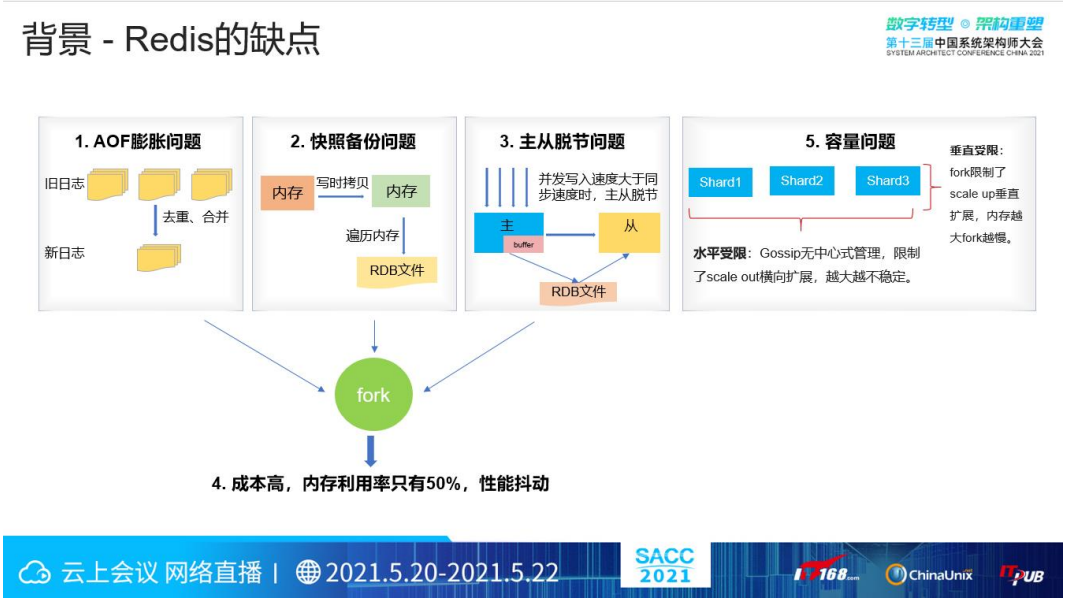
前言：本文根据华为云 NoSQL 数据库架构师余汶龙，在今年的中国系统架构师大会 SACC 上的演讲整理而成，内容如下。

本次分享的大纲分成如下四个部分：

1. 什么是 GaussDB(for Redis)?
2. 为什么选择存算分离
3. 设计与实现
4. 竞争力总结

一、 什么是 GaussDB(for Redis)

1. 开源 Redis 有哪些缺点?



要回答什么是 GaussDB(for Redis)（下文简称**高斯 Redis** ）的问题，首先要从背景讲起。开源 Redis 是个非常好的 KV 缓存，但随着各种业务的蓬勃发展，数据规模、吞吐规模、业务复杂度的不断上升，开源 Redis 暴露出诸多问题：

1）AOF 膨胀问题

开源 Redis 的定位是缓存，但为了满足业务的宕机数据快速恢复，增加了 AOF 日志来实现一定的持久化功能。可惜在 Redis 的设计里，并没有一个转储文件机制来消耗 AOF，而是通过 AOF 重写，来不断的去重合并旧日志。而该重写机制需要一次 fork 调用，该调用会带来内存翻倍、性能阻塞等问题。

2）快照备份问题

随着业务对 Redis 的依赖越来越重，数据备份也变得非常重要。众所周知，Redis 架构并非 MVCC 结构，因此想要备份数据，难免需要悲观锁定之后，拷贝内存数据。不过 Redis 作者设计了一个 copy on write 的方案，即调用 fork，创建出子进程进行数据拷贝，避免了用户态加锁。然而，这个过程其实会在内核侧加锁，依然会给业务性能带来明显抖动。

3）主从脱节问题

开源 Redis 采用主从高可用架构，数据采用异步模式传输。因此主宕机之后，很容易造成数据丢失或不一致。此外，当主节点写入压力较大时，单线程的主从复制很可能无法追平增量数据，就会导致 buffer 堆积，进一步还可能出现写失败甚至 OOM 的灾难。虽然

Redis 能够通过临时生成快照并同步大文件，来尝试追平主从巨大差异，但如前文所述，此时又会引发 fork 系列问题。

4）fork 问题

fork 其实是个非常重的系统调用，虽然是写时拷贝，但是通常也会给他预留一倍的内存。fork 工作时还需要加锁拷贝进程页表等信息，对业务的影响非常之大。上述 3 个问题的背后都有 fork 的因素，通常需要 DBA 采用关闭主节点 AOF、关闭主节点备份等复杂运维手段来避免。但在主从频繁切换、节点数很多的场景下，运维是非常困难的。甚至在主从脱节场景，理论上毫无办法规避。

5）容量问题

开源 Redis 不适合大规模使用，有两个重要因素限制了其扩展性。首先是 fork 限制了 Redis 的垂直扩展能力（Scale Up），数据量越大，fork 越慢，对业务的影响就越大，因此单个 redis 进程可承载的数据量非常有限。其次，低效率的 gossip 集群管理限制了其水平扩展能力（Scale Out）：因为节点数越多，其故障发现的时间越长，并且内部通信的网络风暴成几何级数增加，导致大集群几乎不可用。

2. 业界有哪些解决办法?

以上就是各大企业在开源 Redis 的生产实践中，真实碰到的经典问题。这些问题限制了开源 Redis 的大规模应用。因此，近年来业界提出了非常多的解决方案，见下图。

本质上，Redis 是一种 KV 存储，按照场景其实可以进一步划分为两大阵营：缓存与持久

化。

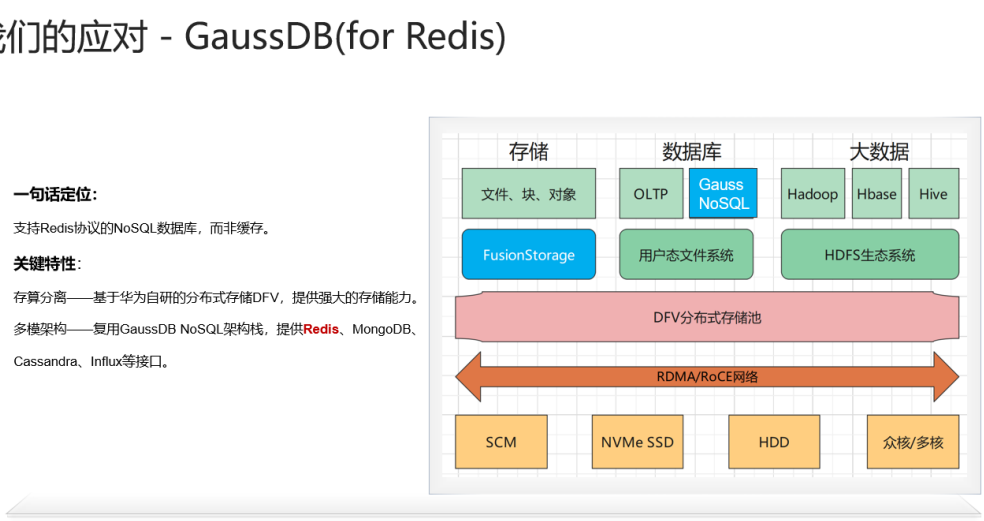
缓存场景：一般用来存放秒杀、热点事件的数据。比如微博热搜，这类数据是有有效期的，而且可丢。

持久化场景：在用 Redis 做缓存的时候，由于其接口简单、功能丰富，大家必然希望将更多重要数据也持久化存放到 Redis，比如历史订单、特征工程、位置坐标、机器学习等。这类数据的数据量往往很大、有效期也很长、一般不可丢。

缓存场景比较简单就是开源 Redis，持久化场景业界已有非常多自研产品，比如 360 的 ssdb/pika，阿里的 tair，腾讯的 tendis，当然华为云的**高斯 Redis** 也属于自研的持久化 Redis。

这里也补充另一个做持久化的理由，从成本考虑，256G 内存条价格比 256G 的 SSD 磁盘高了将近 30 倍，在可用容量上也有巨大差异。

3. 华为云数据库的解法是啥?



华为云数据库团队吸取开源 Redis 的经验，选择了自研持久化 Redis，即今天分享的主角——**高斯 Redis**。它的一句话定位是：支持 Redis 协议的 NoSQL 数据库，而不是缓存。它有两个跟业界完全不一样的特性：

1）存算分离

高斯 Redis 基于华为内部自研分布式存储 DFV，提供强大的数据存储能力，包括强一致、弹性扩缩容等高级特性。DFV 为何物？它是华为全栈数据服务的基石，比如文件

EVS、对象 OBS、块存储，还有数据库族、大数据族，都依赖于此，可以想象它的强大及稳定性。

2）多模架构

实际上**高斯 Redis** 是多模数据库 Gauss NoSQL 的一员，Gauss NoSQL 提供了全栈的分布式 KV 引擎、用户态文件系统、存储池等技术，只需要在接口上封装 Redis 协议，即可轻松实现一个全新的 NoSQL 产品。类似的，我们还提供了 MongoDB、Cassandra、Influx 等 NoSQL 引擎。

GaussDB(for Redis)免费体验： 企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，**点击体验**

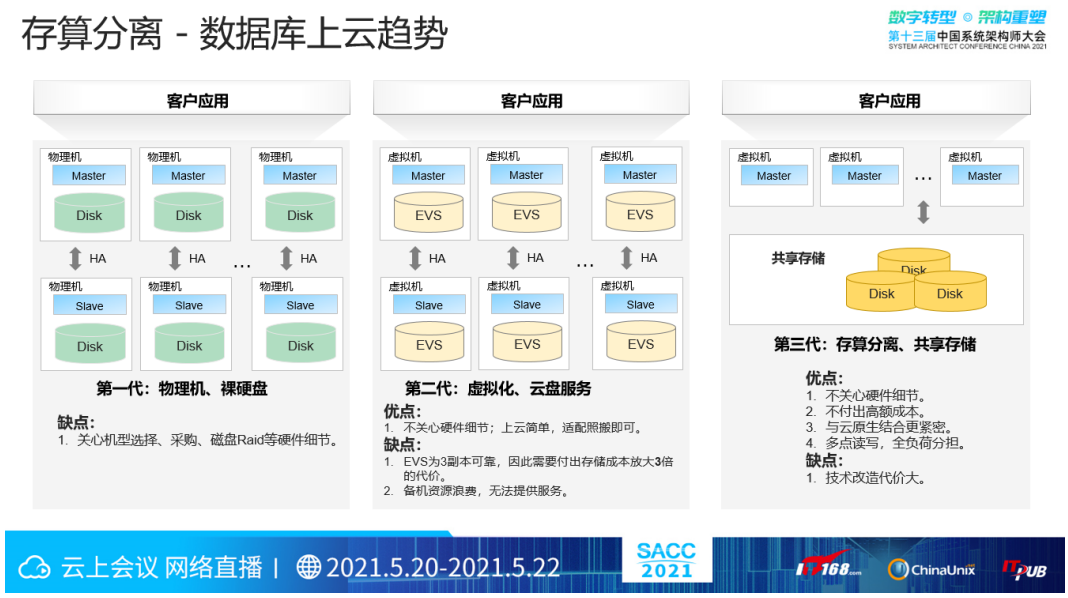
二、 为什么选择存算分离?

在云原生概念铺天盖地的今天，数据库也逐步走向云原生，而它的云原生有一个重要特点就是存算分离。存算分离也代表了数据库上云的最新趋势。

第一代数据库服务：通过下图可以看到，传统 IDC 建设时，数据库架设在裸金属之上，由于数据库服务的敏感特殊性，DBA 或者研发需要关心机型的选择、磁盘 Raid 阵列、组网，甚至采购等诸多事项。

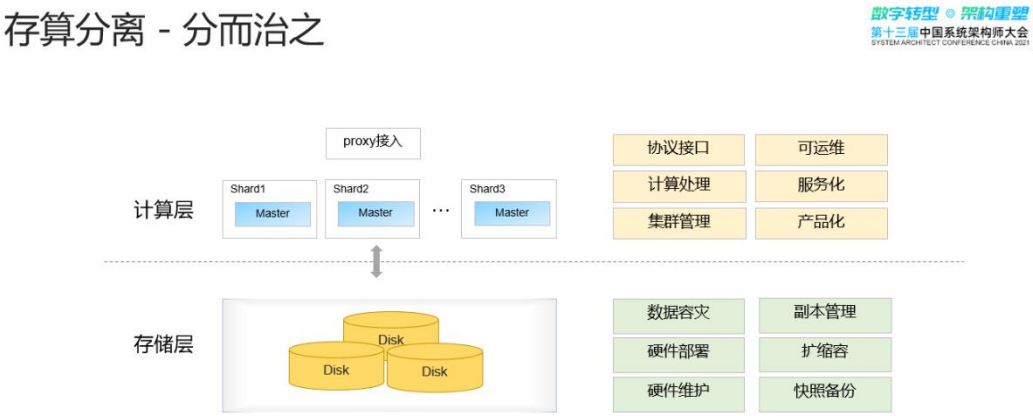
第二代数据库服务：随着虚拟化技术的普及，应用型业务大量上云，数据库也开始上云搬迁，最简单的办法是在虚拟机或容器中运行一个数据库服务即可。这样做的优点很明显，但缺点有两个：一个是通用云盘都是 3 副本，加上数据库上层的多副本，资源浪费严重；另一个是备机资源浪费，平时无法提供服务。除此以外还有云盘 IO 性能等问题存在。

第三代数据库服务：基于存算分离架构，将数据库服务分成 CPU 密集的计算层和 IO 密集的存储层。数据的副本管理完全交给存储层，计算层实现无状态转发，既能发挥云的弹性优势，又能全负荷分担。不过缺点也很明显，即基于旧架构改造难度大。



云上会议 网络直播 | 2021.5.20-2021.5.22 SACC 2021

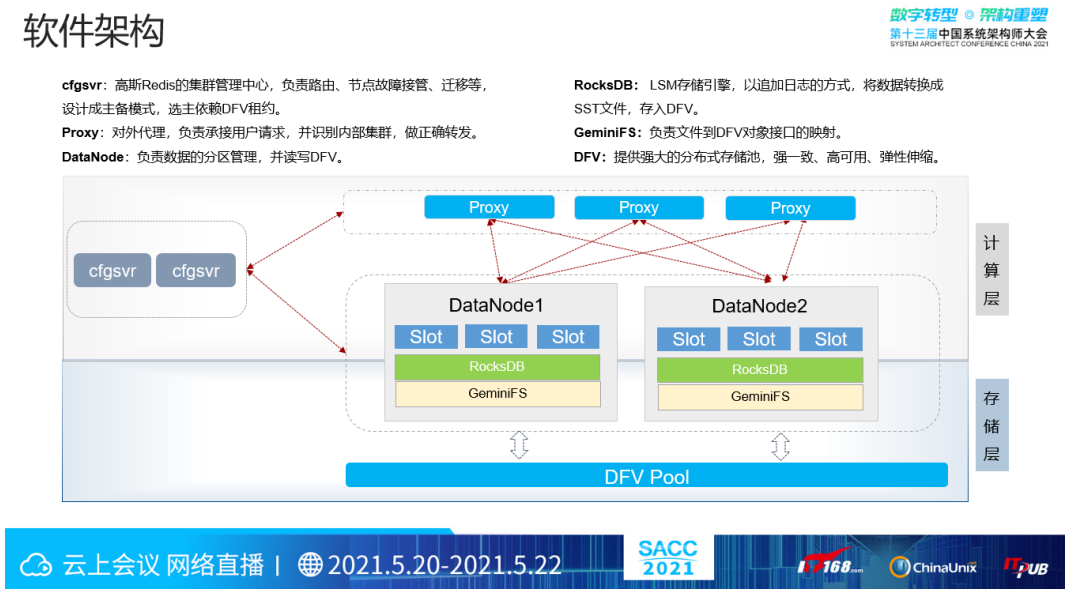
采用存算分离架构之后，数据库服务就是个分而治之的思想：计算层负责服务化、产品化的各种处理，全程无状态；而存储层，就专注于数据本身的维护，包括副本、容灾、硬件感知、扩缩容等等。



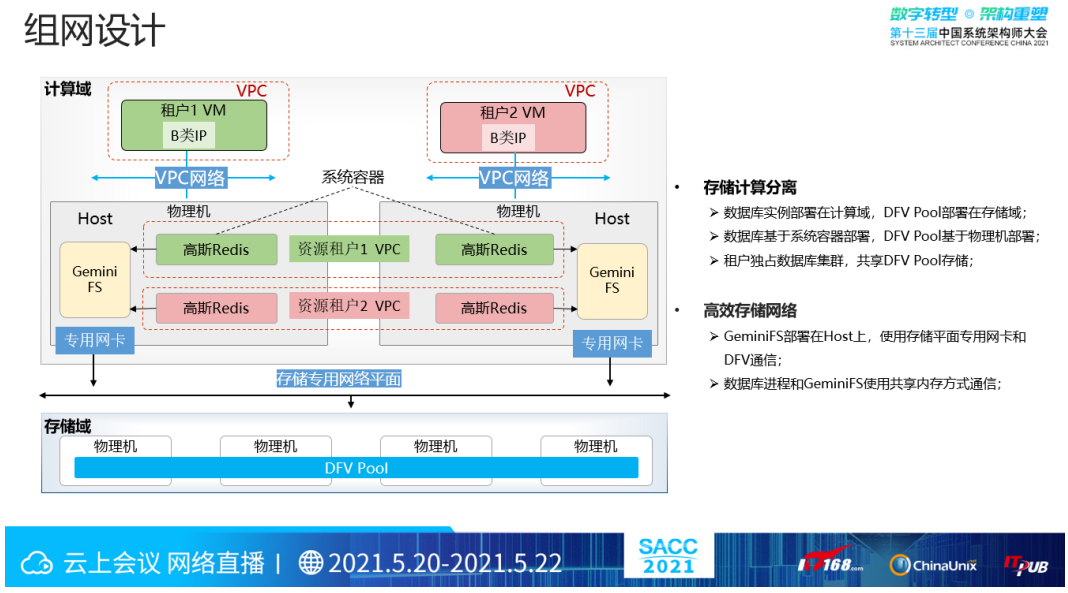
云上会议 网络直播 | 2021.5.20-2021.5.22 SACC 2021

三、 设计与实现

接下来讲整体设计与实现，首先是软件架构。**高斯 Redis** 计算层的模块如下，主要有 cfgsvr、proxy、datanode。连接计算与存储资源的有 RocksDB 和 GeminiFS（自研用户态文件系统），分别负责将 kv 数据转成 sst 文件和负责将 sst 文件下推到 DFV 的对象存储池中。



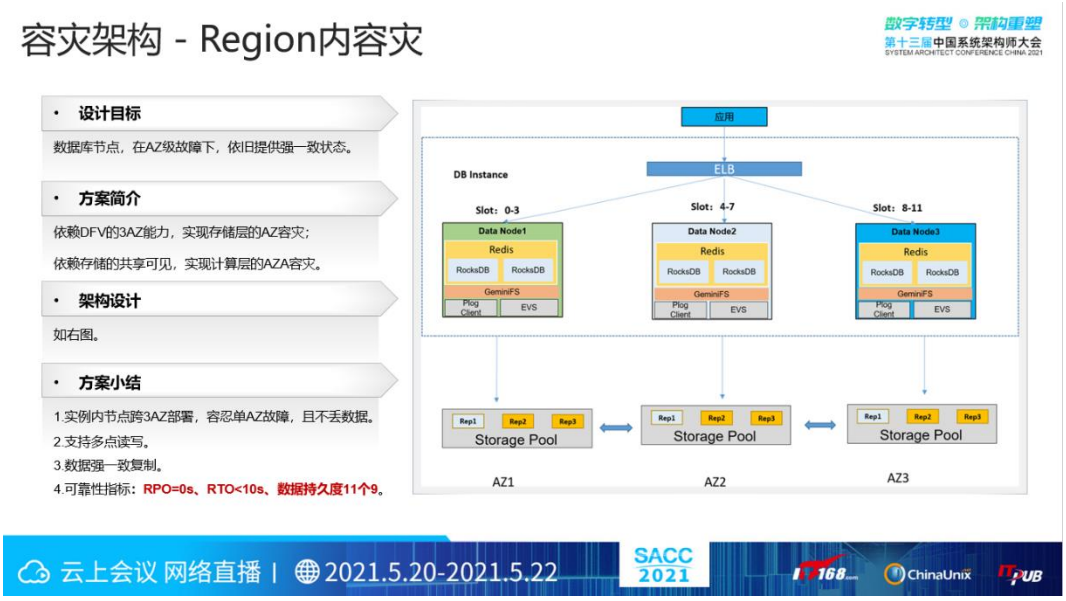
接下来是组网设计。一个租户申请的数据库资源，被我们以反亲和的方式，分布在不同的物理机容器上，都属于同一个租户的相同 VPC 下。不同用户的数据库资源虽然也有可能共享同一台物理机，但是由于 VPC 隔离，保证了数据隔离。另外，计算层的数据库资源是独占容器的，而存储层资源是共享物理硬件的。



接下来解读容灾架构。既然 GaussDB(for Redis)定位是数据库而不是缓存，那它对待数据的态度是严肃的：既实现了 region 内的 3AZ 容灾，也提供了跨 Region 的容灾。

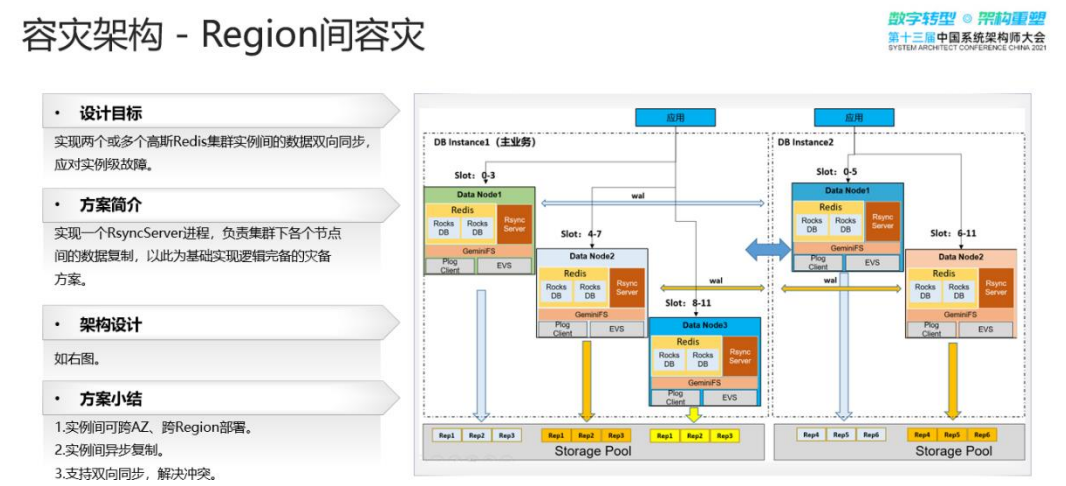
Region 内的容灾，实现了一个容忍 AZ 级故障的高可用方案。在此故障下，数据依然保持强一致状态，这对企业级应用提供了非常强大的数据安全保障。这套架构的可靠性指标可以满足 RPO 为 0，RTO 小于 10s 的标准。

具体的实现原理是，依赖 DFV 的 3 副本强一致复制能力，计算层也做 3AZ 的反亲和部署。当用户的一条数据通过 proxy 写到 datanode1 上，datanode1 通过 GeminiFS 的用户态文件系统，调用 DFV 的 SDK 找到一个 local az 的 DFV 存储节点，和一个距离最近 remote az 的 DFV 存储节点，组成多数派，写成功后即返回给用户。这样的架构下，不管是计算还是存储的 AZ 级故障，都对数据的安全性没有任何影响。



接下来继续讲跨 Region 级别的容灾。高斯 Redis 除了提供上述 3AZ 的强一致方案以外，还提供跨 Region 级别的容灾，也就是两个实例间的异步容灾。这套方案里，我们增加了一个 Rsync-Server 的模块，用来订阅主实例上新增的日志，再把日志反解编码成相应的格式，转发给对端的备实例，由备实例回放即可。

这套方案，可以实现双向同步、断点续传、冲突解决等等。其中冲突解决，针对不同的 Redis 数据结构，采用不同的解决算法，保障最终一致性。



四、 竞争力总结

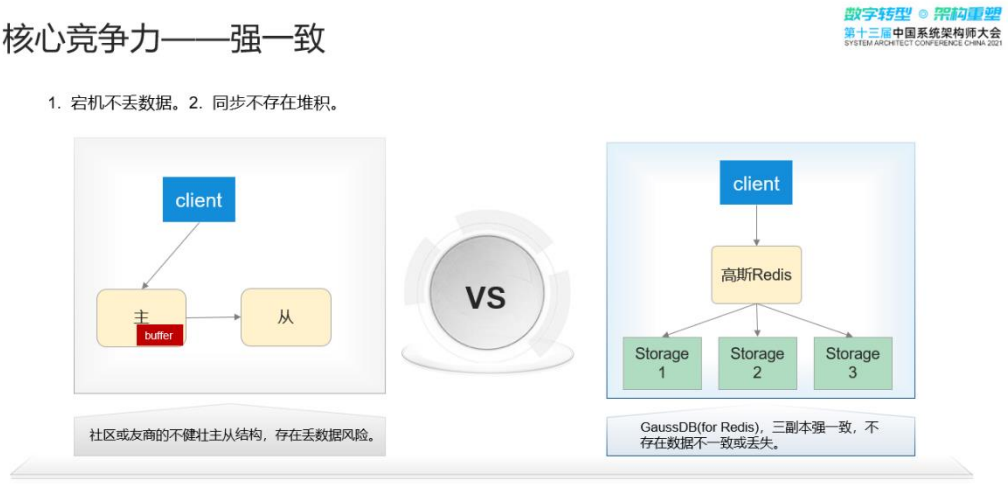
最后一节是对**高斯 Redis** 的优势总结，主要包括：强一致、高可用、冷热分离、弹性伸缩、高性能。

首先是**强一致**特性。

这一点主要受益于 DFV 的 3 副本机制，因此写入高斯 Redis 的数据，在客户端收到回复时，数据就已是 3 副本强一致的。强一致能力对业务实现非常友好，不需要忍受数据的不一致、不需要校验数据。

而开源 Redis 数据采用异步复制，因此主从之间总是有个差异 buffer，如果掉电，这部分数据就会丢失，且在大压力写的时候，还会产生 buffer 堆积，严重的时候，会导致 OOM。

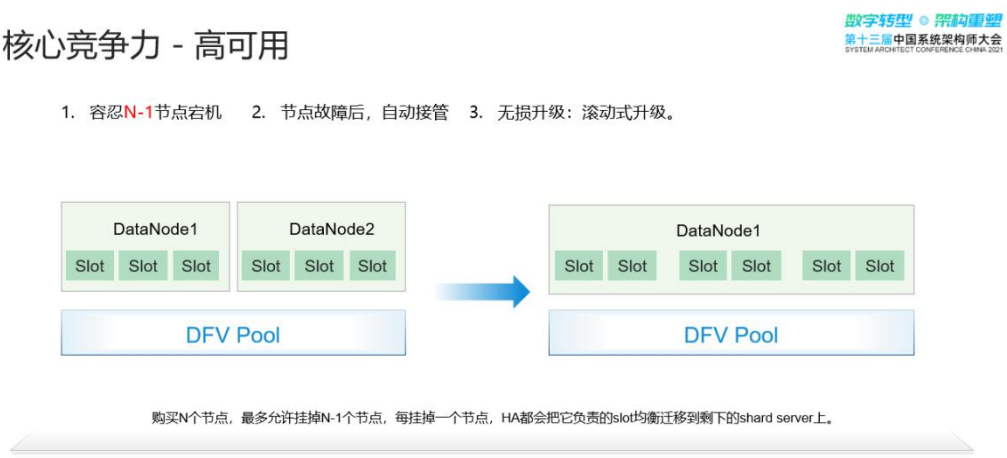
因此，**高斯 Redis** 的强一致是个非常重要的特点，能为业务提供前后一致的状态，不用担心开源 Redis 主从切换后的数据一致性问题和丢失问题。



第二个特性是**高可用**。

高可用是数据库的基本能力，这里之所以要再次强调，是因为高斯 Redis 的可用性跟其他数据库不同，它做到了可接受 N-1 个节点故障。实现原理受益于共享存储 DFV：当某一个计算节点发生故障挂掉，其维护的 slot 路由信息，会被剩下的节点自动接管。

由于不涉及底层数据的迁移，这个接管过程非常快。以此类推，可以接受 N-1 个节点故障，且不影响全部数据的读写。当然，计算节点减少会对性能造成一定影响。

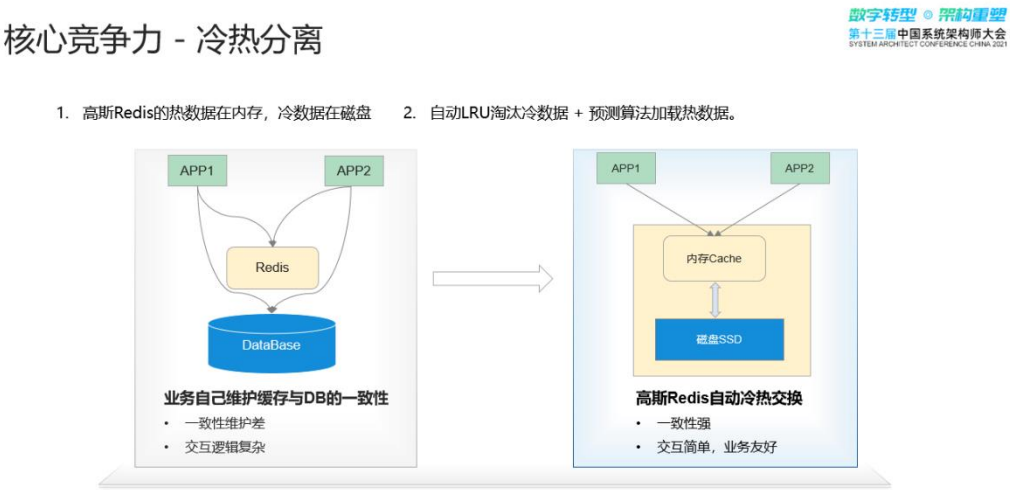


云上会议 网络直播 | 2021.5.20-2021.5.22 SACC 2021 ChinaUnix

第三个特性是**冷热分离**。

开源 Redis 的一个经典使用场景是配合 MySQL 做冷热分离，但这需要业务实现代码负责实现冷热数据交换，并维护其一致性，这个交付逻辑比较复杂。而**高斯 Redis** 实现了它自己的冷热分离，即用户刚写入的和经常访问的数据，都被当做热数据加载到内存中，而非频繁访问的数据则会被淘汰到持久化存储中。

因此使用了**高斯 Redis** 的业务，不再需要从业务层写代码维护冷热交换逻辑，并且可以得到更好的一致性。



云上会议 网络直播 | 2021.5.20-2021.5.22 SACC 2021 ChinaUnix

第四个特性是**弹性伸缩**。

采用存算分离之后的**高斯 Redis**，可以做到按需扩容，即计算不够扩计算，存储不够扩存储即可。

计算资源的扩容也很简单，前面已经提到，这个过程其实不涉及数据的拷贝搬迁，只涉及到元数据的修改，即把相应的 slot 路由信息（不超过 1MB）迁移到新增的节点上即可完成，因此速度是非常快的，秒级完成。

而存储资源的扩容更简单，由于底层采用共享存储，大多数情况进行逻辑扩容，这只需要用户在控制台上修改配额即可完成，不涉及到任何数据的搬迁和拷贝。

当然也有碰到物理扩容的情形，这种情形一般是我们运维提前发现警戒水位，在这之前做平滑的迁移扩容，该过程对用户透明无感知的。

云上会议 网络直播 | 2021.5.20-2021.5.22 SACC 2021 ChinaUnix ITPUB

第五个特性是**高性能**。

存算分离的架构看似比较重，链路比较复杂，实则在硬件采用、软件优化上，可以做的更大胆更激进，比如 RDMA 网络、用户态协议、持久化内存等等。因此受益于这些专属的存储设备，加上我们的计算层全负荷分担架构（不引入从节点，因此性能轻松翻倍），在对比友商的数据量大于内存的存储场景下，我们的性能表现很好。

另外，对比开源 Redis，在数据小于内存的点查场景下，我们的性能也有很大优势，当然范围查询还待优化中。

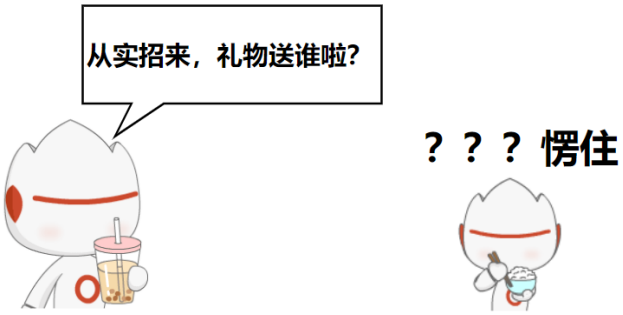
【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！[点击前往]

搞定推荐系统存储难题，高斯 Redis 带你实现想存就存的自由

一、 推荐偏差引发的思考

七夕过后，笔者的一个朋友遇到了尴尬事：当女友点开他的购物 APP，竟然自动弹出一系列推荐：玫瑰包邮、感动哭了、浪漫小夜灯……回想七夕那天，礼物并没有出现，于是问题出现了：从实招来，你送谁了？

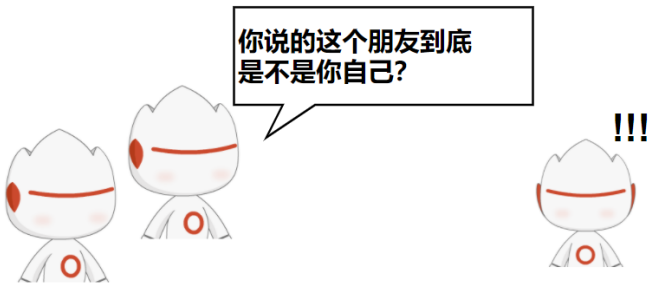


为了帮助友人重建信任，笔者进行了一番技术调研：**这一定是“推荐系统”出了偏差。**

推荐系统是一种信息过滤系统，它能够快速分析海量用户行为数据，预测出用户喜好，从而进行有效推荐。

在商品推荐、广告投放等业务中，推荐系统责任重大。根据亚马逊 2019 年度报告，其**40%的营收来自内部稳定的推荐系统**。

在本文开篇的例子中，正是由于推荐系统问题，才导致了尴尬的场面。笔者决定力挺友人，用可靠的知识让人信服！



二、 推荐系统长什么样

通常来说，在一套成熟的推荐系统中，分布式计算、特征存储、推荐算法是关键的三大环节，缺一不可。

下面介绍一类完整的推荐系统，在系统中 GaussDB(for Redis)负责核心的特征数据存储。该系统也是华为云诸多客户案例中较为成熟的最佳实践之一。

第一部分：获取特征数据

第一部分：获取特征数据

- 原始数据采集

  点赞、收藏、评论、购买……这些行为都属于原始数据，他们随时都在发生，因此数据量庞大。经由 Kafka、Redis Stream 等流组件向下游传递，或存入数仓，等待后期提取使用。

- 分布式计算

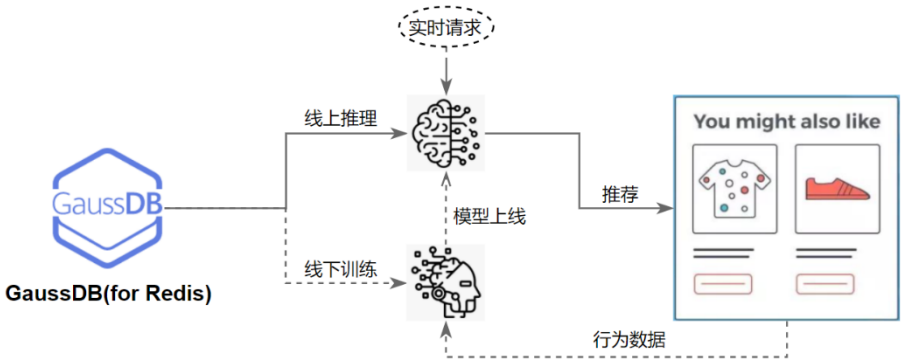
  原始数据离散、含义模糊，无法直接给算法使用。此时就要进行大规模的离线、在线计算，对数据加工。Spark、Flink 都是典型的大数据计算组件，其强大的分布式计算能力是推荐系统不可或缺的。

- 特征数据存储

经过加工的数据也就是特征、标签，是推荐算法所需的宝贵数据源。在特定场景下，也可以称之为用户画像、物品画像。这部分数据有着反复共享、复用的价值，不仅能用于训练算法模型，还能为生产环境提供服务。

确保特征数据的可靠存储，是推荐系统中极为关键的一环。

第二部分：消费特征数据



第二部分：消费特征数据

- **线下模型训练**

有了关键的特征数据，业务就可以开始训练算法模型。

只有充分利用特征库，以及最新行为数据，不断打磨推荐算法，这样才能提升推荐系统整体水平，最终带给用户更好的体验。

- **线上推理预测**

算法模型训练结束后，将被部署到线上生产环境。

它将继续利用已有的特征存储，根据用户的实时行为进行推理，快速预测出与用户最匹配的优质内容，形成推荐列表，并推送给终端用户。

三、 推荐系统的存储难题

很显然，“特征数据”在整个系统中起到了关键的衔接作用。

由于KV形式的数据抽象与特征数据极为接近，因此推荐系统里往往少不了Redis的身影。

在上述系统的方案中，数据库选型为 GaussDB(for Redis)，而不是开源 Redis。

原因是开源 Redis 在大数据场景下还是存在显而易见的痛点：

1. 数据无法可靠存储

推荐系统其实希望既能使用 KV 数据库，又能放心将数据长久保存。

但开源 Redis 的能力更侧重于数据的缓存加速，而不是数据存储。而且开源 Redis 毕竟是纯内存设计，即使有 AOF 持久化，但通常也只能秒级落盘，数据的保存并不可靠。

2. 数据量上不去，成本下不来

涉及推荐的业务往往用户体量也不会小，随着业务发展，也会有更多的特征数据需要保存。

实际上，相同容量的内存与极速 SSD 相比，价格贵 10 倍以上都很正常。

于是，当数据量达到几十 GB、几百 GB，开源 Redis 会变得越来越“烧钱”，因此一般只当做“小”缓存使用。

除此之外，开源 Redis 自身 fork 问题导致容量利用率低，硬件资源有很大的浪费。

3. 灌库表现不佳

特征数据需要定期更新，往往以小时或天为周期进行大规模数据灌入任务。

如果存储组件不够“皮实”，大量写入造成数据库故障，将导致整个推荐系统发生异常。

这就可能造成开篇提到的尴尬用户体验。

开源 Redis 抗写能力并不强，这是由于集群中有一半节点是备节点，它们只能处理读请求。当大批量写入到来时，主节点容易出问题，引发连锁反应。

理论上，架构设计并不是越复杂越好，如果可以，谁不想使用一种既能兼顾特征数据 KV 类型、成本友好、性能又有保障的可靠数据存储引擎?

四、 相见恨晚，遇见 GaussDB(for Redis)

与开源 Redis 不同，GaussDB (for Redis)基于存算分离架构，为推荐系统这一类大数据场景带来关键的技术价值：

1. 可靠存储

数据命令级落盘，在底层存储池中三副本冗余存储，真正做到了 0 丢失。

2. 降本增效

高性能持久化技术+细粒度存储池,帮助企业将数据库使用成本降低 75%以上。

3. 抗写能力强

多线程设计+全部节点可写，抗写能力足够强大，从容应对 Spark 灌库压力和实时更新。

华为云企业级数据库 GaussDB (for Redis)提供稳定、可靠的 KV 存储能力，正是推荐系统核心数据的极佳选型。

五、 完美衔接，实现想存就存的自由

其实，在 Spark 后端接入 Redis 已经成为一种主流方案，而使用 Flink 从 Redis 中提取维度表也是很常见的用法。它们也都提供了用于接入 Redis 的连接器。

GaussDB(for Redis)完全兼容 Redis 协议，即开即用,用户随时都可以快速创建实例并接入业务。

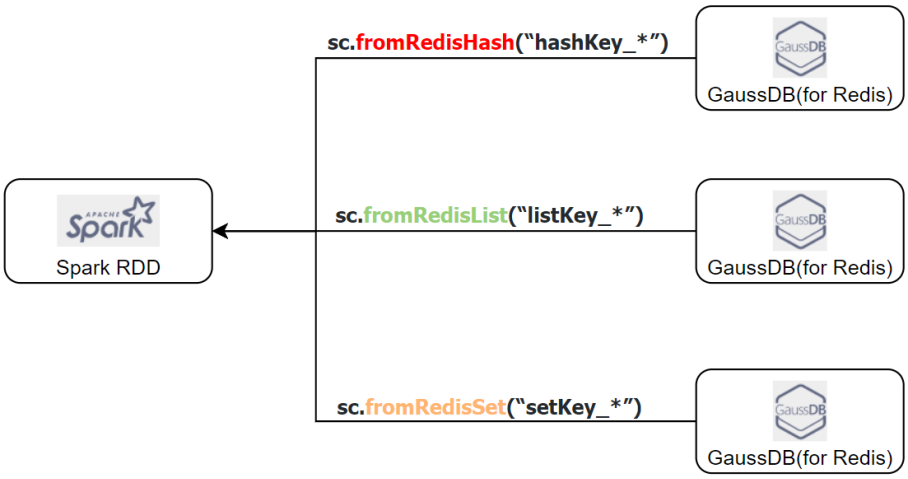
Spark-Redis-Connector

Spark-Redis-Connector 完美实现了 Spark RDD、DataFrame 到 GaussDB(for Redis)实例中 String、Hash、List、Set 等结构的映射。

用户可使用熟悉的 Spark SQL 语法轻松访问 GaussDB(for Redis),完成特征数据灌库、更新、提取等关键任务。

使用方法非常简单:

1）当需要读取 Hash、List、Set 结构到 Spark RDD 时，分别只用一行即可搞定。



2）而当推荐系统进行灌库或特征数据更新时，可以按如下方式轻松完成写入。

Flink-Redis-Connector

Flink 这款计算引擎流行程度不亚于 Spark，它同样有成熟的 Redis 连接方案。

使用 Flink 提供的 Connector 或结合 Jedis 客户端，都可轻松完成 Flink 到 Redis 的读写操作。

以使用 Flink 统计单词频次的简单场景为例，数据源经过 Flink 加工后，便可轻松存入 GaussDB(for Redis)中。

六、 结语

大数据应用对核心数据的存储有着很高的要求。云数据库 GaussDB(for Redis)拥有存算分离的云原生架构，在完全兼容 Redis 协议的基础上，同时做到了稳定性、可靠性的全面领先。

面对海量核心数据存储，它还能为企业带来相当可观的成本节约。

面向未来，GaussDB(for Redis)极有潜力成为下一个大数据浪潮的新星。

GaussDB(for Redis)免费体验：企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，**点击体验**

【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！**点击前往**

核心秒杀业务还在频频踩坑？如何彻底搞定数据一致性问题

有人说，开源 Redis 的最终一致性已经能满足大部分应用场景，也有人说，多副本的强一致代价太大，没有必要实现。要笔者说，其实弱一致性已经不满足很多应用场景的诉求。怎么，不信？请听笔者娓娓道来。

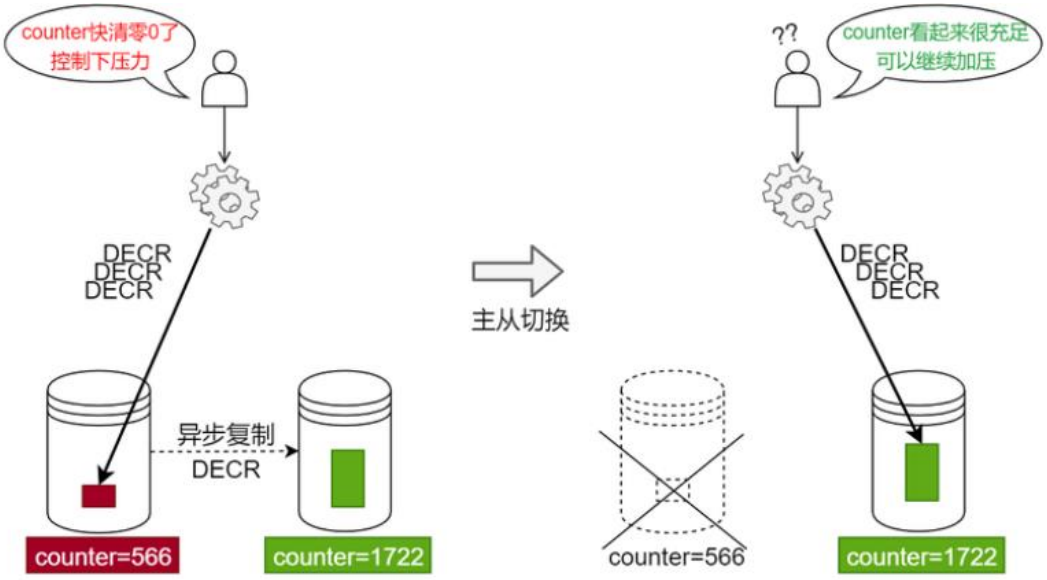
一、不一致带来的困扰

1. 秒杀变秒崩

分享一个电商秒杀活动中限流器的例子，在电商的秒杀活动中，为了扛住前端对数据库的超大流量冲击，一般使用两种方案来保护系统，一个是缓存，另一个则是限流。缓存这个容易实现，只需要在数据库前加一层缓存服务器，而对于限流来说，最简单的可以使用 Redis 的计数器来实现限流功能。

具体来说，假设我们需要对某个接口限定流量为 5000qps，即每秒钟访问的次数不能超过 5000。那么我们可以这么做：在一开始的时候设置一个计数器 counter 为 5000，并且过期时间为 1s，即 1s 后计数器失效。每当一个请求过来的时候，counter 的值减 1，判断当前 counter 的值是否等于 0，如果等于，则说明请求次数过多，直接拒绝请求。如果 counter 计数器不存在，则重置计数器为 5000，开始新一秒的接口限流，注意并发情况下计数器需要加锁。

正常情况下，这种方案不会出现问题，但是针对这种秒杀活动，不怕一万，就怕万一，万一 Redis 突然宕机怎么办，那岂不是限流器形同虚设，所有流量全部涌向后端的数据库，瞬间系统崩溃。此时聪明的你肯定会想到，给 Redis 搞一个备用服务器不就解决了，主服务器如果宕机，备用服务器顶上。没错，这种方案是对的，但是只正确了一半。为什么呢，如下图所示。

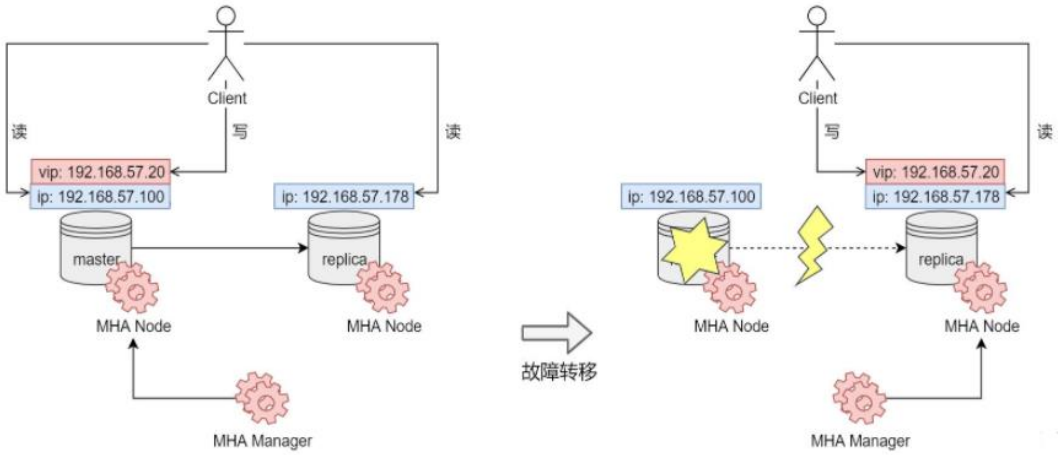


当给 Redis 配置从服务器之后，如果主服务器出现宕机，可以立刻切换到从服务器，但是由于开源 Redis 主从服务器之间的数据是异步复制的，如果网络不畅，经常发生主从数据不一致，如果此时主服务器发生宕机，切换到从服务器之后，因为限流器的判断出错，流量压力很容易超出阈值，一下子涌向数据库服务器，同样会造成系统崩溃。

仔细探究这个问题产生，根因是在于开源 Redis 的一致性机制为弱一致性，在某些时间内，主从副本数据不一致。而要彻底解决这个问题，只有真正的强一致才能解决。

2. 难以维护的 MySQL 组件

其实不止 Redis，就连大名鼎鼎的 MySQL 也逃不过弱一致的坑。MySQL 的部署中，为了保证高可用性，主从热备份是 MySQL 常用的部署方式。但是如果发生故障时，仅仅靠 MySQL 自身的同步机制，是无法保证主库和从库之前的数据一致的，于是出现了重要的辅助组件 MHA(Master High Availability)，它的部署方式如下：



MHA 由管理服务和 Node 服务组成，Node 服务部署在每个 MySQL 节点上，MHA 组件负责让 MySQL 的从库尽可能的追平主库，提供主从一致的状态。发生故障进行主从切换时，Manager 首先为从库补充落后的数据，然后再将用户访问切换到从库，这个过程可能长达数十秒。

MHA 的部署和维护都相当复杂，如未能顺利执行故障切换或发生数据丢失，运维面临的场面都将很棘手。其实运维同学何尝不希望手中的系统稳定运行呢？要是数据库自身能提供强一致保障，何苦再依赖复杂的辅助组件！

二、什么是强一致

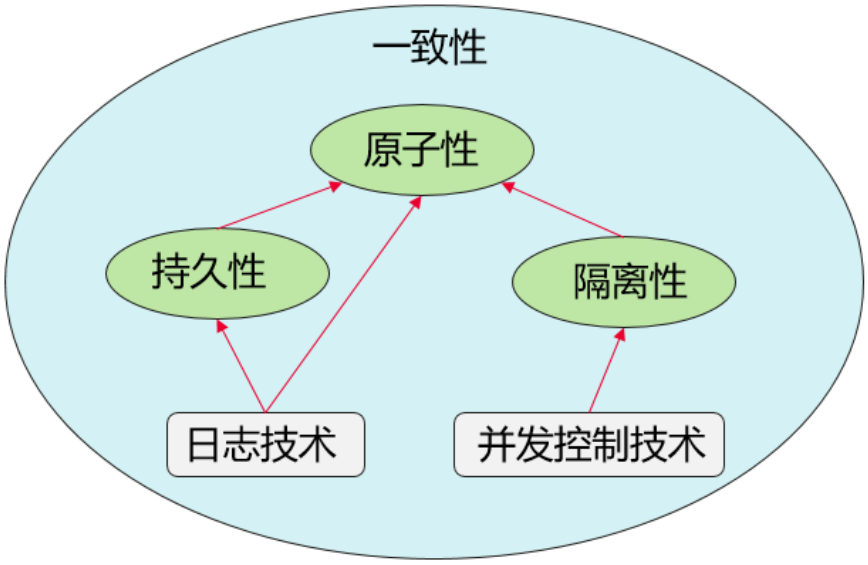
上一节中笔者介绍了弱一致带来各种问题，接下来这一节具体介绍下什么是强一致。在“分布式系统”和“数据库”这两个领域中，一致性都是重要概念，但它表达的内容却并不相同。

对于分布式系统而言，一致性是在探讨当系统内的一份逻辑数据存在多个物理的数据副本时，对其执行读写操作会产生什么样的结果，这也符合 CAP 理论对一致性的表述。而在数据库领域，“一致性”与事务密切相关，又进一步细化到 ACID 四个方面。

因此，当我们谈论分布式数据库的一致性时，实质上是在谈论事务一致性和数据一致性两个方面。

1. 事务一致性

事务的一致性主要是指的事务的 ACID，分别是原子性、一致性、隔离性和持久性，如下图所示：



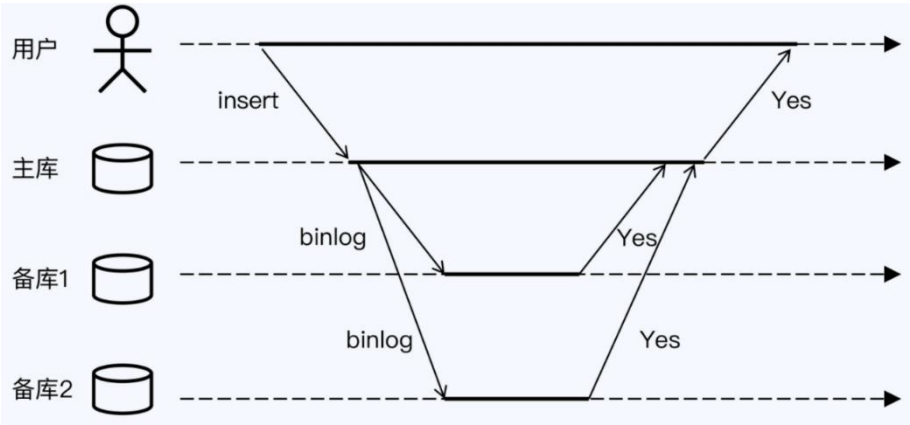
- **原子性：**事务中的所有变更要么全部发生，要么一个也不发生，通过日志技术实现；
- **一致性**：事务要保持数据的完整性，它是应用程序的属性，依赖原子性和隔离属性来实现；
- **隔离性**：多事务并行执行所得到的结果，与串行执行（一个接一个）完全相同，通过并发控制技术来实现；
- **持久性**：一旦事务提交，它对数据的改变将被永久保留，不应受到任何系统故障的影响，通过日志技术实现。

2. 数据一致性

在分布式系统中，为了避免网络不可靠带来的问题，通常会存储多个数据副本，逻辑上的一份数据存储在多个物理副本上，自然带来了数据一致性问题。

（1）状态视角

从状态的视角来看，任何变更操作后，数据只有两种状态，所有副本一致或者不一致。在某些条件下，不一致的状态是暂时，还会转换到一致的状态，而那些永远不一致的情况几乎不会去讨论，所以习惯上大家会把不一致称为“弱一致”。相对的，一致就叫做“强一致”了。以一个一主两备的 MySQL 集群为例，“强一致”的交互过程如下：

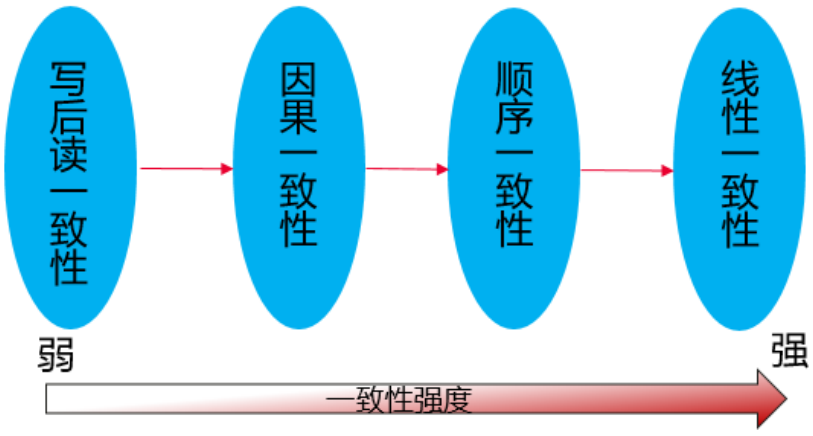


在该模式下，主库与备库同步 binlog 时，主库只有在收到两个备库的成功响应后，才能够向客户端反馈提交成功。显然，用户获得响应时，主库和备库的数据副本已经达成一致，所以后续的读操作肯定是没有问题的，这就是状态视角的”强一致”的模型。

但是，状态视角的这种强一致副作用很大：第一个是性能很差，主库必须要等备库 1 和备库 2 成功返回后，主库才返回；第二个是可用性问题，如果主备节点很多，出现故障的概率非常高。因此状态视角的强一致代价非常大，因此很少使用。

（2）操作视角

状态视角的强一致降低了系统的可用性，因此很多系统选择状态视角的弱一致性模型，通过额外的算法（如 Raft、Paxos）在不保证所有节点状态的一致的情况下，来保证操作视角的一致性，同时提高了系统的可用性。通过加入一些限定条件，衍生出了若干种一致性模型：



- 线性一致性：操作视角实现真正的强一致
- 顺序一致性：一致性强度弱于线性一致性
- 因果一致性：一致性强度弱于顺序一致性

- 写后读一致性：一致行强度相当，弱于因果一致性

这些一致性模型的介绍参考《高斯 Redis 与强一致》这篇文章。

三、强一致的刚需场景

上一节我们介绍了什么是强一致，这一节中我们介绍介绍下我们强一致的典型应用场景。

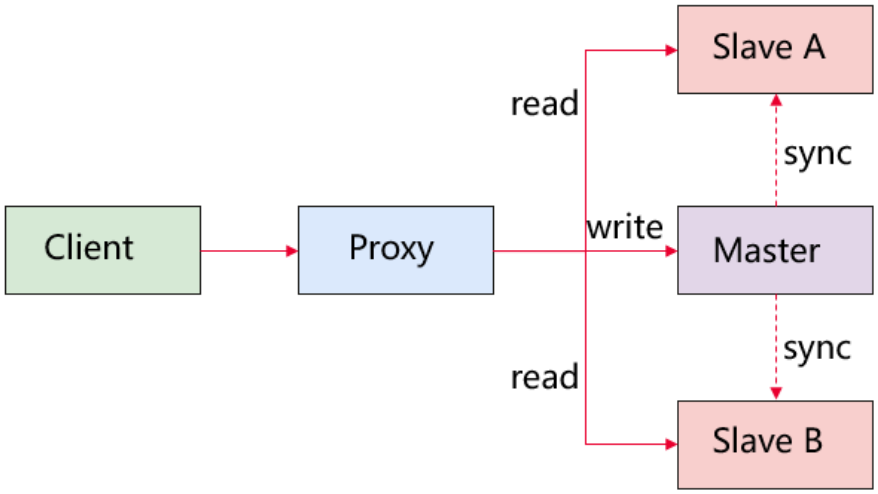
在常见的互联网应用中，如果数据库服务器只部署在单个节点上，那么应用程序所有的读和写都只会访问单个节点，一份逻辑数据在物理上也只有一份，这种场景下就谈不上强一致的问题。

但是随着系统中业务访问量的增加，如果是单机部署数据库，就会导致 I/O 访问频率过高，数据库就会成为系统的瓶颈。此时，为了降低单机磁盘的 I/O 访问频率，提高单个机器的 I/O 性能，通常会增加多个数据存储节点，形成一主一从或者一主多重的架构，

此时，我们可以将负载分布在多个从节点上，一方面可以实现读写分离，写请求访问主库，读请求访问备库。另一方面，还可以在主库如果出现宕机的情况下进行主备切换，增强系统的稳定性。在以上两个场景中，由于一份逻辑数据在物理上有多个副本，那么如何保证多个副本之间的数据一致呢，这就是强一致需要解决的问题。

1. 读写分离场景

以关系型数据库 MySQL 为例，典型的部署方案为一主两从三节点方案，主节点负责处理写操作，两个从节处理读操作，分担主库的压力，如下图所示：



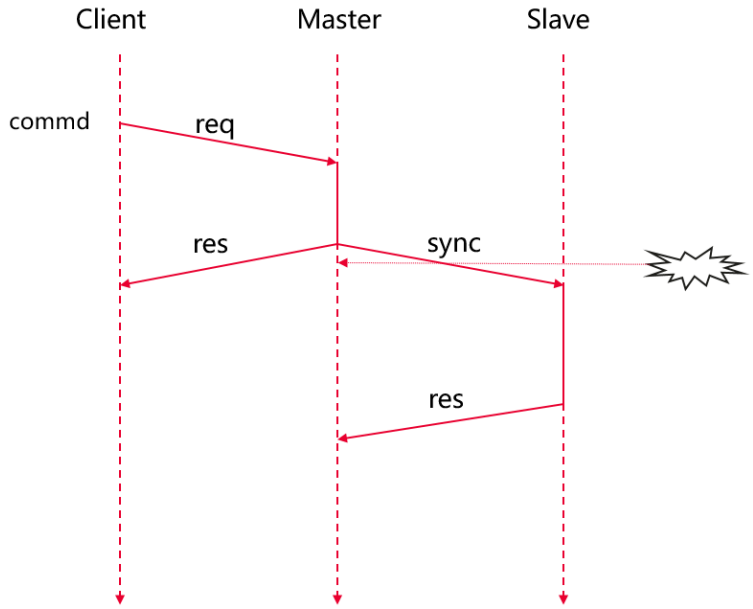
此时，如果系统没有实现强一致，就有可能会遇到执行完写操作后，立刻去读发现读不到或者读到旧状态的尴尬场景，比如操作顺序为以下操作：

- 客户端首先通过代理向主节点 Master 进行了写入操作，此时由于没有实现强一致，写操作写完后立即返回；
- 紧接着第二步去从节点 Slave A 执行读操作，此时 Master 和 Slave A 之间的同步还未完成，系统处于非强一致的状态，所以第二步的读操作读取到了旧状态。

可以看出，在一主多备读写分离的场景下，如果想要保证写入和读取操作的准确无误，系统实现强一致是非常重要的。

2. 主备切换场景

主备切换的场景也需要强一致来保证，以目前业内使用最广泛的内存数据库 Redis 为例，Redis 的主从同步如下图所示：



从上图可以看出，当 Redis 客户端向 Master 服务器发送一条命令时，Master 服务器立即回复客户端命令的执行结果，并不等待命令同步到从服务器再回复，也就是说 Redis 的主从同步其实是异步的。

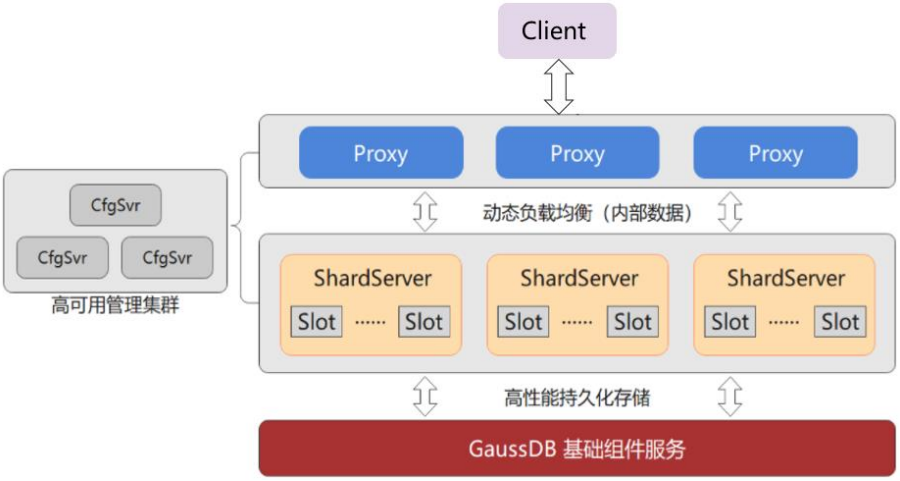
由于 Master 节点存在宕机的可能，在这种情况下，如果在 Master 收到命令但是还没同步到 Slave 服务器时发生了宕机，Redis 就会发生主备切换，然后此时 Master 服务器和 Slave 服务器的数据还没有同步，就导致了数据丢失的情况。可见，开源 Redis 弱一致性本身的缺陷和不足，而要解决这个问题，必须实现强一致性才能解决。

四、高斯 Redis 强一致

由于开源的 Redis 不具备强一致的特性，导致开源 Redis 的应用也受到了诸多限制，为了解决开源 Redis 弱一致的问题，GaussDB(for Redis)应运而生。

GaussDB(for Redis)是华为云数据库团队自主研发的兼容 Redis 协议的云原生数据库，彻底解决了开源 Redis 一致性问题带来的痛点。

1. 高斯 Redis 架构

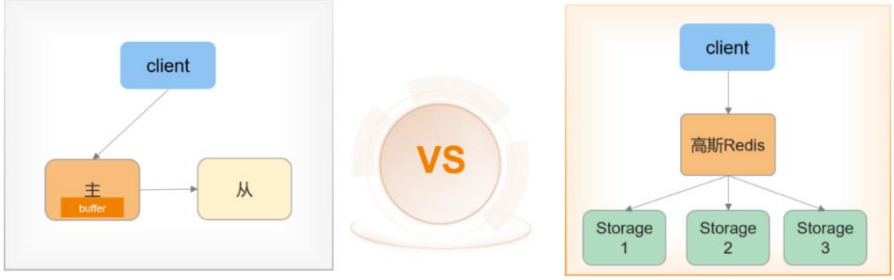


相比于开源 Redis，高斯 Redis 采用存算分离的设计思想，计算层负责计算和协议等的处理，聚焦服务。而存储层负责副本管理、扩缩容等处理，聚焦数据本身。高斯 Redis 的优势如下：

- 数据强一致：存储层使用分布式存储 DFV，轻松实现了 3 副本强一致；
- 超可用：N 个节点的集群最多可以挂掉 N － 1 个节点；
- 低成本:数据采用磁盘存储并且进行压缩,每 GB 的成本不到开源 Redis 的十分之一；
- 秒扩容：计算层仅需修改路由映射，无需数据搬迁，实现秒级扩容；
- 自动备份：高斯 Redis 可以实现 MVCC 快照备份和定期自动备份。

2. 高斯 Redis 强一致的实现

开源 Redis 和高斯 Redis 的架构如下图所示：



开源 Redis 或者传统的主从结构如左图所示，如果在读写分离的场景或者主节点出现宕机发生主从切换的时候，都会导致数据不一致的情况。

高斯 Redis 采用存算分离的架构，如右图所示，在存储层 DFV 的副本管理中采用分布式共识算法实现了 3 副本的强一致。计算层调用存储层的接口时，如果返回 OK，那么即表示存储层已经实现副本强一致的复制。

GaussDB(for Redis)免费体验： 企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，**点击体验**

五、结语

我们在做架构设计的时候，其实很多场景都隐藏着强一致的诉求。如朋友圈这类应用，如果没有实现强一致，朋友圈的评论很容易乱序。再比如限流器的场景，如果没有强一致的保证，也极容易造成数据库的崩溃。因此，必须在系统设计之初就认识到强一致的重要性，才能设计出更加稳定和可靠的系统。

【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！**点击前往**

超越开源 Redis 的 ACID“真”事务

日常生活中的 shopping、交通、手游都离不开高频的金融消费、虚拟交易。熟悉 MySQL 的读者，一定知道数据库事务（Transaction）可以搞定这类关键场景，事务不但极大简化了上层业务的编程模型，给开发者带来便利，同时它也让“交易”等核心业务正确可靠。

其实，Redis 也有事务，但社区版只做了简版实现，无法满足 ACID 要求，因此应用有限。

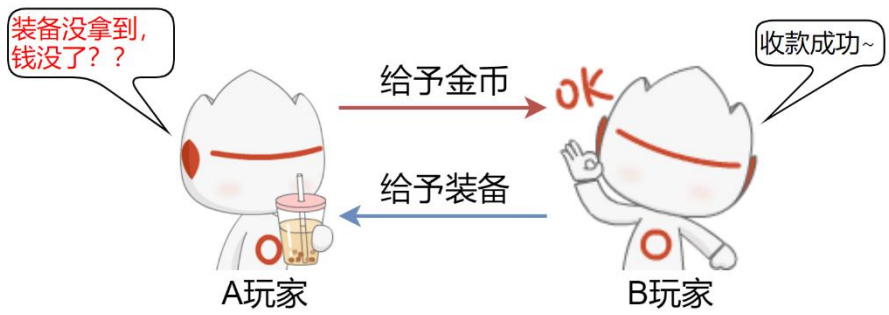
本文将介绍华为云企业级数据库 GaussDB(for Redis)（下文简称为高斯 Redis）的事务功能，与社区版不同，高斯 Redis 提供了满足 ACID 的企业级事务特性。

一、 什么是事务?

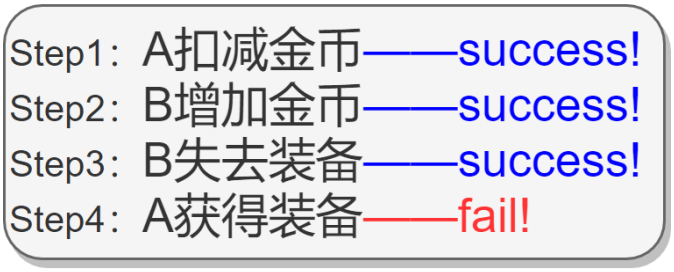
请设想如下场景：

在一个月黑风高之夜，一场交易正在进行。玩家 B 收款成功，但 A 却没拿到装备。

好了，一场 PK 在所难免。



玩家做几次点击即可完成交易，但底层数据库内部，却需要执行至少 4 个关键操作：



试想，如步骤 4 失败，而前置步骤成功，会发生逻辑错乱，这将带来灾难性的用户体验。

事务是如何解决这类场景的？这依赖于 4 大特性：

- 原子性（Atomicity）：一组操作要么全成功，要么全失败。
- 一致性（Consistency）：即满足既定约束，保证系统不脱离合理状态。例如，如果 A、B 各有 50 枚金币，那么全局总数应始终保持 100。
- 隔离性（Isolation）：多个事务的执行相互隔离，互不影响。
- 持久性（Durability）：提交成功的事务应带来永久性状态变化，即使掉电，数据也不应丢失。

其中，一致性是最关键的，而原子性、隔离性、持久性都是为了保证一致性。当然，为了确保整体系统的一致性，也需要业务层的共同设计实现，本文不详细展开。

那么，开源 Redis 事务存在哪些问题，GaussDB(for Redis)的企业级事务功能又是什么样的？下面我们来具体看看。

二、 从四大特性看 Redis 的事务

1. 原子性

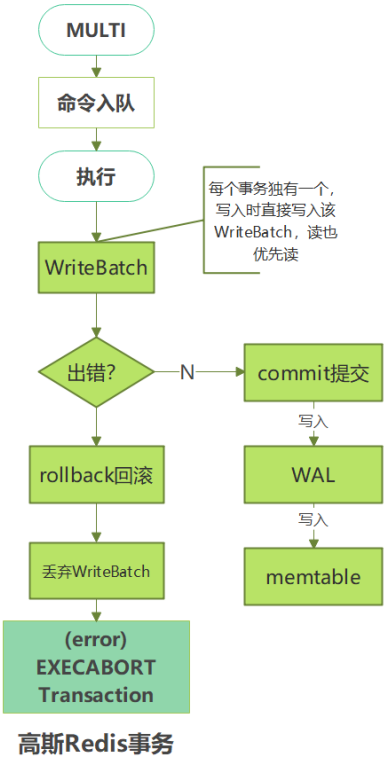
社区版 Redis

开源 Redis 事务是一种极简设计，并不支持事务失败时的回滚操作，不满足事务的原子性。

此外还有以下问题：

- **入队校验不完备：**只校验简单语法，无法识别 key 的类型错误；
- **错误处理不完备：**直接跳过错误，继续执行后续命令。

GaussDB(for Redis)




```
127.0.0.1:6666> get string
"value"
127.0.0.1:6666> multi
OK
127.0.0.1:6666> incr string
QUEUED
127.0.0.1:6666> get string
QUEUED
127.0.0.1:6666> exec
1) (error) ERR value is not an intege
2) "value"
```

开源Redis事务

```
127.0.0.1:6379> get string
"value"
127.0.0.1:6379> multi
OK
127.0.0.1:6379> incr string
QUEUED
127.0.0.1:6379> get string
QUEUED
127.0.0.1:6379> exec
(error) EXECABORT Transaction discarded b
out of range
127.0.0.1:6379> get string
"value"
```

高斯Redis事务

如图所示，开源 Redis 事务在第一条命令出错的情况下仍旧执行了下面的命令，而高斯 Redis 则避免了这样的问题。

- 每个事务的执行在提交前会进行错误判断，如有错误则会触发回滚逻辑，放弃之前的操作，对原数据无影响
- 高斯 Redis 在保留社区版事务使用逻辑的同时，在底层实现了对回滚的支持，满足了事务的原子性。

2. 隔离性

社区版 Redis

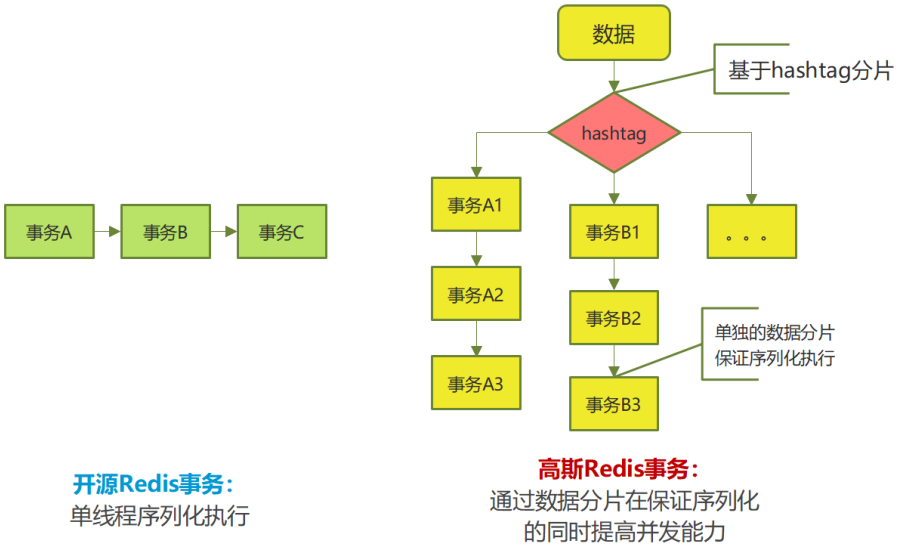
事务在并发执行时，应当处理各种隔离性问题。

开源 Redis 对命令的执行是单线程设计，因此的确可以保证**不会有两个事务同时被执行，**具体来看：

- 开源 Redis 不存在隔离性问题；
- **代价是性能的降低和对整体吞吐的影响，**事务只能单线程处理，对其他请求的干扰较大。

GaussDB(for Redis)

高斯 Redis 是多线程架构，内部对数据进行自动分片，在同数据分片内保留序列化的隔离性级别，同时极大提高整体实例的并发能力。



3. 持久性

社区版 Redis

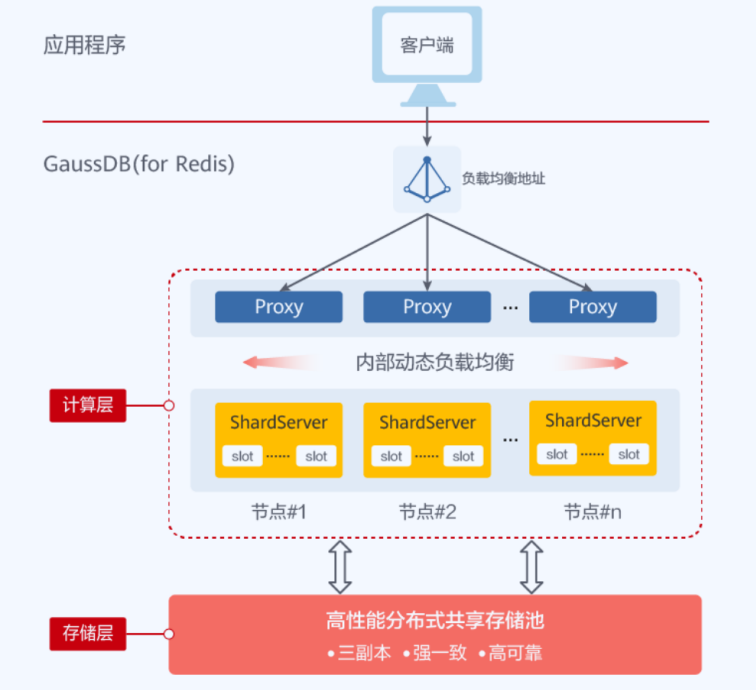
开源 Redis 是纯内存设计，不提供可靠的数据存储能力。掉电后数据丢失，即使开启 RDB 或 AOF，也只能在一定程度上挽回部分数据，具体来看：

- **RDB 方式会丢失部分数据**，数据只能恢复到上次快照的时间
- **即使是 AOF 方式，持久性也存在问题**，从命令执行到保存到硬盘之间，仍旧存在时间差
- **AOF 方式恢复缓慢、维护文件庞大**，维护管理成本都会非常高，重建时间也很长

综合来看，社区版 Redis 的持久化能力有限，不足以支持事务的持久性要求。

GaussDB(for Redis)

高斯 Redis 采用存算分离的设计理念，底层使用高性能分布式存储池保存全量数据，结合 RocksDB 存储引擎，可保证核心数据的可靠存储，架构图：



GaussDB(for Redis)**核心优势：**

- **核心数据存储**：全量数据落盘，3 副本冗余，不怕丢数据
- **高可用**：秒级故障接管，即使 N-1 节点故障，全量数据也可用
- **低时延**：自动冷热分离，亚毫秒级时延，能 Hold 缓存场景
- **高吞吐**：全部节点可写，QPS 可水平扩展，能抗流量高峰
- **强一致**：3 副本强一致同步，不会发生脏读，业务免踩坑
- **秒扩容**：例如，8G 到 64G 扩容只需 1 秒，且对业务 0 影响
- **降成本**：数据量越大价格越香，能够真正帮用户省成本

高斯 Redis 强大的持久化保障，支持核心数据的可靠存储，也为事务的执行提供了有力的保证。

4. 一致性

从数据库层面，数据库通过原子性、隔离性、持久性来保证一致性，可以说在 ACID 中一致性是事务的目的，其他特性是手段。

如上文所述，开源 Redis 在原子性和持久性上都存在种种问题，一致性自然也无法保障。在上述特性的保障下，不论是正常执行还是存在命令失败场景，高斯 Redis 事务都有可靠的一致性。

5. 对比总结

通过上述介绍我们从事务的四大特性出发详细地介绍了高斯 Redis 的事务，最后做一个总结：

事务功能	开源Redis	云数据库GaussDB(for Redis)
原子性	NO 执行时出错则跳过，无原子性	YES 支持回滚保证原子性
隔离性	YES 单线程串行化执行	YES 利用数据分区实现了多线程，增强了性能
一致性	NO 原子性缺失，破坏一致性	YES 原子性和强一致有效保证
持久性	YES 持久化需要启用AOF 高可用依赖主从同步，异步同步增加风险	YES/Better 数据全量下沉到共享存储池

三、 结语

近年来，以 Redis 为代表的 KV 数据库逐渐在越来越多的场景发挥作用，而高斯 Redis 作为华为云旗舰 KV 数据库，可满足企业核心数据的可靠存储要求。

高斯 Redis 的事务能力相比社区版 Redis 有极大提升，在 ACID 事务的加持下，更可在“库存”、“交易”等关键场景简化业务设计，带来可靠数据存储能力。

GaussDB(for Redis)免费体验：企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，**点击体验**

【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！**点击前往**

测评篇

GaussDB(for Redis) PK 原生 Redis 集群，谁更技高一筹?

一、 说明

本文为 GaussDB(for Redis)和开源 Redis 集群在 X86 架构下性能测试对比报告，分析测试结果，GaussDB(for Redis)较开源 Redis 集群能提供更高的 QPS，更低的访问延迟，以及更低的数据存储成本。

二、 测试方案

1. 测试环境

华为云北京四环境，具体部署方式参照 2.4 章节拓扑图。

2. 服务端资源配置

名称	CPU	内存	存储	数据库类型	价格（包月）
GaussDB Redis	4vCPU	48G	120GB	GaussDB (for Redis)	￥4,606.80
proxy redis		48G(主)+48G(备)	小于40G	开源 Redis 集群	￥4,874.88

3. 客户端配置

测试执行机规格:

名称	CPU	内存	操作系统
ECS 虚拟机（2 台）	8vCPUs	16G	CentOS 6.9 64bit

规格：通用计算增强型 | c6.2xlarge.2 | 8vCPUs | 16GB

三、 实例部署拓扑图

1. GaussDB(for Redis)集群部署

2. 开源 Redis 集群部署

四、 测试工具

测试工具	版本	下载地址
memtier_benchmark	1.3.0	https://github.com/RedisLabs/memtier_benchmark

五、 测试流程

1. 测试环境准备

1. 登录华为云：www.huaweicloud.com。
2. 创建 GaussDB(for Redis)集群

   购买 GaussDB(for Redis)实例参考
   https://support.huaweicloud.com/redisug-nosql/nosql_02_0071.html

3. 创建开源 Redis 集群
4. 购买测试客户端 ECS，安装 memtier_benchmark

2. 测试样例

- **string 类型纯写测试**

执行以下命令，测试 string 类型纯写的性能，并为读测试预置数据。

```
./memtier_benchmark -s ${redis_ip} -a ${password} -p ${redis_port} -c 20 -t 4  -n 1500000 --random-data --randomize --distinct-client-seed -d
```

```
1024 --key-maximum=1500000 --key-minimum=1 --ratio=1:0 --key-pattern=S:S --show-config
```

- **string 类型纯读测试**

执行以下命令，测试 get 的性能

```
./memtier_benchmark -s ${redis_ip} -a ${password} -p ${redis_port} -c 20 -t 4  -n 1500000 --random-data --randomize --distinct-client-seed -d 1024 --key-maximum=1500000 --key-minimum=1 --ratio=0:1 --key-pattern=S:S --show-config
```

- **string 类型读写 1:1 测试**

执行以下命令，测试读写 1:1 的性能

```
./memtier_benchmark -s ${redis_ip} -a ${password} -p ${redis_port} -c 20 -t 4  -n 1500000 --random-data --randomize --distinct-client-seed -d 1024 --key-maximum=1500000 --key-minimum=1 --ratio=1:1 --key-pattern=S:S --show-config
```

- **hash 类型纯写测试**

执行以下命令，测试 hset 的性能，并为下阶段读预置数据。

```
./memtier_benchmark -s ${redis_ip} -a ${password} -p ${redis_port} -c 20 -t 4 -n 1500000 --random-data --randomize --distinct-client-seed -d 1024 --key-maximum=1500000 --key-minimum=1 --command='hset __key__ field __data__' --command-key-pattern=S --command-ratio=1 --show-config
```

- **hash 类型纯读测试**

执行以下命令，测试 hget 的性能。

```
./memtier_benchmark -s ${redis_ip} -a ${password} -p ${redis_port} -c 20 -t 4 -n 1500000 --random-data --randomize --distinct-client-seed -d 1024 --key-maximum=1500000 --key-minimum=1 --command='hget __key__ field' --command-key-pattern=S --command-ratio=1 --show-config
```

- **hash 类型读写 1:1 测试**

执行以下命令，测试读写 1:1 的性能

```
./memtier_benchmark -s ${redis_ip} -a ${password} -p ${redis_port} -c 20 -t 4 -n 1500000 --random-data --randomize --distinct-client-seed -d 1024 --key-maximum=1500000 --key-minimum=1 --command='hset __key__ field __data__' --command-key-pattern=S --command-ratio=1 --command='hget __key__ field' --command-key-pattern=S --command-ratio=1 --show-config
```

说明: ${password}为创建数据库时设置的密码，${redis_ip}，${redis_port}表示数据库的连接地址和端口号,测试使用并发数 80，data_size：1024。

六、 测试结果

1. 性能测试

GaussDB(for Redis)					
类型	data_size	qps	latency(ms)	p99(ms)	p9999(ms)
set	1024	168245.41	0.48	0.80	7.20
get	1024	169466.14	0.47	0.79	5.70
set && get	1024	168127.72	0.47	0.79	5.63
hset	1024	166739.55	0.48	0.80	5.65
hget	1024	166503.99	0.48	0.79	5.95
hset && hget	1024	167421.87	0.47	0.79	5.43

Redis集群					
类型	data_size	qps	latency(ms)	p99(ms)	p9999(ms)
set	1024	140398.84	1.67	3.30	8.55
get	1024	147784.27	1.63	3.00	6.95
set && get	1024	146621.82	1.63	3.00	8.05
hset	1024	143943.09	1.67	3.40	8.55
hget	1024	147419.35	1.63	3.00	7.15
hset && hget	1024	150614.64	1.64	3.00	8.20

通过分析上表的数据，通过 GaussDB(for Redis)的测试结果和开源 Redis 集群测试结果的比值，得到下表的对比数据。

GaussDB(for Redis) / Redis集群（增长或下降百分比）								
类型	qps		latency		p99		p9999	
set	119.83%	↑	28.87%	↓	24.24%	↓	84.21%	↓
get	114.67%	↑	28.52%	↓	26.33%	↓	82.01%	↓
set && get	114.67%	↑	28.82%	↓	26.17%	↓	69.88%	↓
hset	115.84%	↑	28.73%	↓	23.53%	↓	66.08%	↓
hget	112.95%	↑	29.19%	↓	26.33%	↓	83.22%	↓
hset && hget	111.16%	↑	28.90%	↓	26.17%	↓	66.16%	↓

2. 大数据量写测试

进行一组数据量超过内存的写测试，开源 Redis 集群在耗尽内存后会报错：server xxxxxxxxx:xxxx handle error response: -OOM command not allowed when used memory > 'maxmemory'

相比之下，GaussDB(for Redis)没有任何的影响，业务正常进行，以下为 GaussDB(for Redis)在数据量超过内存时的性能结果：

类型	qps	latency	p99	p9999
set	255861.78	0.47	1.05	37.50
get	276799.67	0.43	0.97	34.50
setget	272387.7	0.44	1.06	28.90

3. 数据压缩测试

GaussDB(for Redis)提供数据压缩服务，数据存储成本更低。预置一定数据量，查看 GaussDB(for Redis)和原生 Redis 集群实际占用的存储空间。以下为预置的数据量和实际占用的存储空间：

数据库类型	数据类型	预置数据量	存储占用空间
GaussDB(for Redis)	string	1.43G	0.187G
Redis集群	string	1.43G	1.96G
GaussDB(for Redis)	hash	1.43G	0.282G
Redis集群	hash	1.43G	2.18G

七、 结果分析

上述测试表明了 GaussDB(for Redis)的各方面优势如下：

1. **性能优势**

分析性能测试结果，在相同测试条件下，GaussDB(for Redis)的 QPS 较开源 Redis 集群**提高了 11%~19%**，平均延迟和 P99 比 Redis 集群**降低了 70%以上**，p9999 比 Redis 集群**降低了 15%以上**。

2. **抗写优势**

在数据量大于内存的写测试中，原生 Redis 集群因内存限制而 OOM，GaussDB(for Redis)依然可以提供不俗的性能服务。实际上，GaussDB(for Redis)可用的存储空间是底层 SSD 大小决定的，相比原生 Redis 集群抗写优势显著。

3. **数据存储成本更低**

GaussDB(for Redis)提供了高效的数据压缩服务，数据压缩测试结果显示，GaussDB(for Redis)占用的存储空间只有开源 Redis 集群的**十分之一**，相当于数据存储成本**降低了 10 倍**。

GaussDB(for Redis)免费体验：企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，**点击体验**

【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！**点击前往**

现身说法：GaussDB(for Redis)的大肚量与真稳定

GaussDB(for Redis)是华为云推出的企业级 Redis，采用计算存储分离架构，兼容 Redis 生态的云原生 NoSQL 数据库，基于共享存储池的多副本强一致机制，支持持久化存储，保证数据的安全可靠。具有高兼容、高性价比、高可靠、弹性伸缩、高可用、无损扩容等特点。

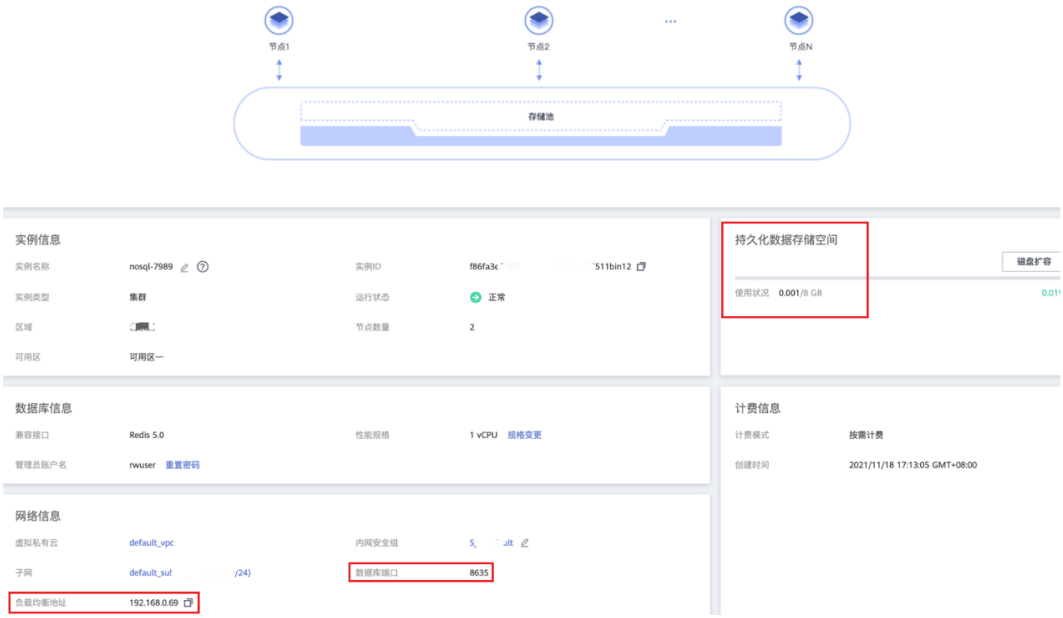
GaussDB(for Redis)满足高读写性能场景及容量需弹性扩展的业务需求，广泛使用于电商、游戏以及视频直播等行业。即可作为前端缓存支撑大并发的访问，也可作为底层数据库负责核心数据可靠存储。

接下来我们使用采用 Redis Labs 推出的多线程压测工具 memtier_benchmark 对比测试下 GaussDB(for Redis) 和原生 Redis 的特性差异。

目录导航

一、 创建 GaussDB(for Redis)实例

在华为云通过控制台购买 GaussDB(for Redis)实例，测试实例的配置为 8G 容量，如下所示。



如截图所示，GaussDB(for Redis)提供了统一的负载均衡地址和端口，方便应用程序访问高可用的 Redis 服务。持久化数据存储空间直观展示了数据量及容量上限。另外，依托于 GaussDB(for Redis)存算分离的架构，实例的容量和性能可以按需分别扩展：

- 如需更多容量，只需点击“磁盘扩容”；
- 如需更高的吞吐性能，则通过“规格变更”或“添加节点”完成。

二、 安装 memtier_benchmark

使用与 GaussDB(for Redis)测试实例相同子网的 ECS 云服务器，部署 memtier_benchmark 测试环境

```
# yum install autoconf automake make gcc-c++
# yum install pcre-devel zlib-devel libmemcached-devel openssl-devel
# git clone https://github.com/RedisLabs/memtier_benchmark.git
# cd memtier_benchmark
# autoreconf -ivf
# ./configure
# make && make install

如 libevent 版本较低，需要在安装 memtier_benchmark 前 按以下步骤安装 libevent
# wget https://github.com/downloads/libevent/libevent/libevent-2.0.21-
stable.tar.gz
# tar xfz libevent-2.0.21-stable.tar.gz
# pushd libevent-2.0.21-stable
# ./configure
```

```
# make
# sudo make install
# popd
# export
PKG_CONFIG_PATH=/usr/local/lib/pkgconfig:${PKG_CONFIG_PATH}

确认安装成功
# memtier_benchmark --help
```

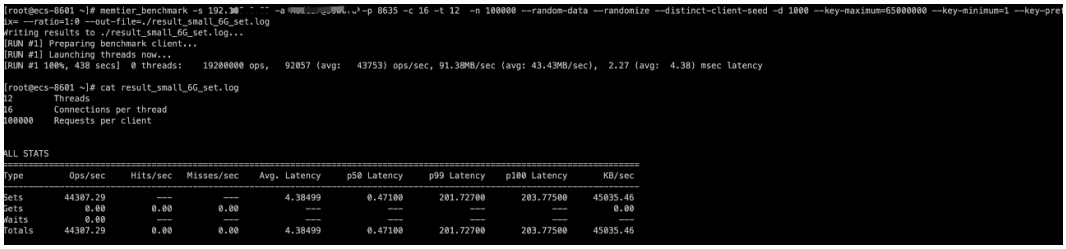
三、 数据批量装载

1）向 GaussDB(for Redis) 中装载数据

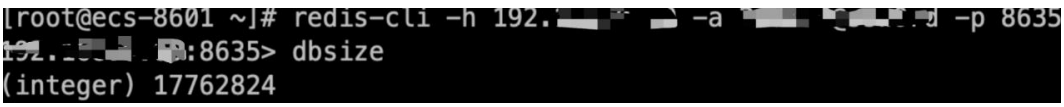
使用 memtier_benchmark 向 GaussDB(for Redis) 中装载数据命令如下，单个 value 长度 1000 字节，12 个线程，每个线程 16 个客户端，每个客户端发出请求数 100000 个，全部是写入操作。

```
memtier_benchmark -s 192.XXX.XXX.XXX -a XXXXXXX -p 8635 -c 16 -t 12 -n 100000 --random-data --randomize --distinct-client-seed -d 1000 --key-maximum=65000000 --key-minimum=1 --key-prefix= --ratio=1:0 --out-file=./result_small_6G_set.log
```

可以看到执行了 1920 万次操作，平均每秒 4.4w 的 ops，总耗时 438 秒。

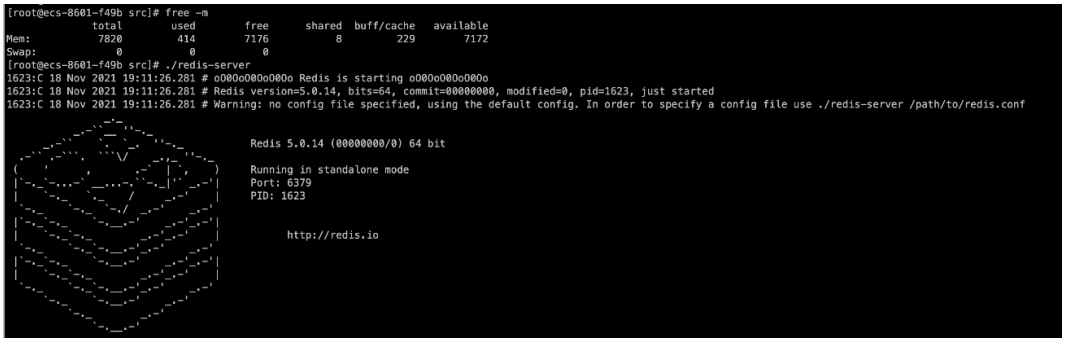


使用 redis-cli 登录实例，查看 dbsize（注意：由于采用 MVCC 机制，查询结果为 key 数量的预估值，非实时的准确值。）



2）向原生 Redis 中装载数据

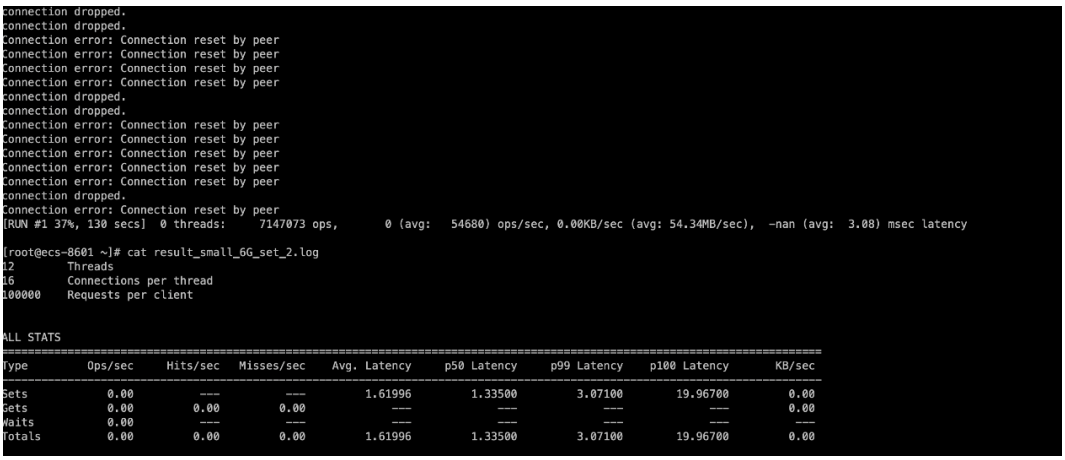
为了对比方便，我们在另一台 4 核 8G 的 ECS 上部署一个单节点的开源 Redis，版本与 GaussDB(for Redis)一致使用 5.0



还是使用 memtier_benchmark 相同的配置向原始 redis 中插入数据

```
memtier_benchmark -s 192.XXX.XXX.XXX  -a XXXXXXX  -p 6379 -c 16 -t 12  -n 100000 --random-data --randomize --distinct-client-seed -d 1000 --key-maximum=65000000 --key-minimum=1 --key-prefix= --ratio=1:0 --out-file=./result_small_6G_set_2.log
```

执行一段时间后出现大量报错



从 Redis 日志中查看，是在做 RDB 快照的时候出现了问题。从系统日志中分析当时发生了 OOM 故障。

```
1711:M 18 Nov 2021 19:34:35.608 * Ready to accept connections
1711:M 18 Nov 2021 19:35:36.092 * 10000 changes in 60 seconds. Saving...
1711:M 18 Nov 2021 19:35:36.096 * Background saving started by pid 1722
1711:M 18 Nov 2021 19:35:39.308 # Background saving terminated by signal 9
1711:M 18 Nov 2021 19:35:42.020 * 10000 changes in 60 seconds. Saving...
1711:M 18 Nov 2021 19:35:42.031 * Background saving started by pid 1723
1723:C 18 Nov 2021 19:35:54.232 * DB saved on disk
1723:C 18 Nov 2021 19:35:54.238 * RDB: 98 MB of memory used by copy-on-write
1711:M 18 Nov 2021 19:35:54.259 * Background saving terminated with success
Killed
```

```
Nov 18 19:35:39 ecs-8601-f49b kernel: redis-server invoked oom-killer: gfp_mask=0x10200da, order=0, oom_score_adj=0
Nov 18 19:35:39 ecs-8601-f49b kernel: redis-server cpuset=/ mems_allowed=0
Nov 18 19:35:39 ecs-8601-f49b kernel: CPU: 0 PID: 1722 Comm: redis-server Kdump: loaded Not tainted 3.10.0-1160.15.2.el7.x86_64 #1
Nov 18 19:35:39 ecs-8601-f49b kernel: Hardware name: OpenStack Foundation OpenStack Nova, BIOS rel-1.10.2-0-g5f4c7b1-20210417_160349-szxrtosci10000 04/01/2014
Nov 18 19:35:39 ecs-8601-f49b kernel: Call Trace:
Nov 18 19:35:39 ecs-8601-f49b kernel: [<ffffffff97781fba>] dump_stack+0x19/0x1b
Nov 18 19:35:39 ecs-8601-f49b kernel: [<ffffffff9777c8da>] dump_header+0x90/0x229
Nov 18 19:35:39 ecs-8601-f49b kernel: [<ffffffff971065a2>] ? ktime_get_ts64+0x52/0xf0
Nov 18 19:35:39 ecs-8601-f49b kernel: [<ffffffff9715dd7f>] ? delayacct_end+0x8f/0xb0
Nov 18 19:35:39 ecs-8601-f49b kernel: [<ffffffff971c227d>] oom_kill_process+0x2cd/0x490
```

这其实和原生 Redis 的 RDB 快照处理方式有关，Redis 是 fork 了一个进程使用 copy-on-write 的方式持久化内存数据，这必然会导致更多内存的申请和使用。并且除了 RDB 快照，原生 redis 在执行 aof 重写，新加从库的操作时也会申请使用更多的内存。为了避免 OOM 的情况出现，操作系统往往要预留出一倍的空闲内存，限制了内存资源的使用率造成极大的浪费。

反观 GaussDB(for Redis) 由于摒弃了 fork 机制，使得架构更健壮。

从上面的测试也可以看到，导入同样数量的数据时，GaussDB(for Redis) 的可用性和响应的性能没有受到任何的影响。

四、 实例紧急扩容

为了测试能进行下去，我们将 GaussDB(for Redis) 和原生 Redis 分别扩容到 16G。

- **GaussDB(for Redis)扩容到 16G**

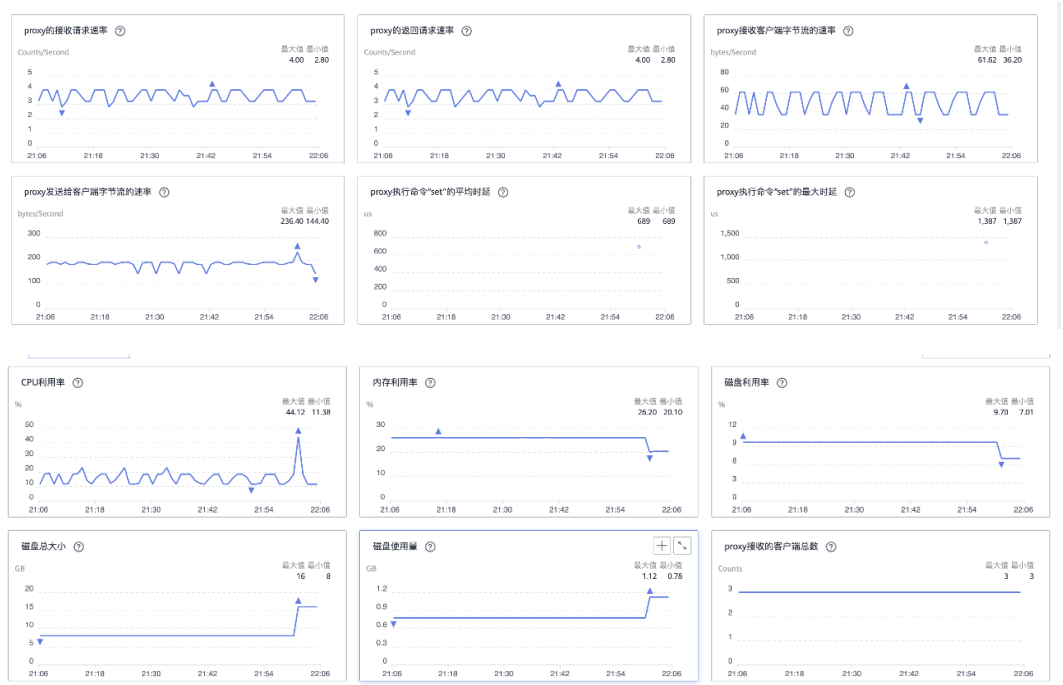
  对 GaussDB(for Redis) 来说由于采用了存算分离的架构，分布式存储池海量在线，按额度分配给用户使用。扩容过程没有数据拷贝，也不会影响业务使用。接下来我们测试使用 memtier_benchmark 在持续的 RW 操作场景下 GaussDB(for Redis)的扩容过程，看看是否会影响业务的读写；

```
memtier_benchmark -s 192.XXX.XXX.XXX -a XXXXXXXX -p 8635 -c 16 -t 12  -n 10000 --random-data --randomize --distinct-client-seed -d 1000 --key-maximum=65000000 --key-minimum=1 --key-prefix= --ratio=1:0 --out-file=./result_small_6G_set_get.log
```

在执行命令的同时进行扩容操作，查看测试结果和监控发现，扩容期间未见报错，

GaussDB(for Redis) 响应时延没有明显变化。

```
[root@ecs-8601 ~]# memtier_benchmark -s 192.... -a ... -p 8635 -c 16 -t 12 -n 10000 --random-data --randomize --distinct-client-seed -d 1000 --key-maximum=65000000 --key-minimum=1 --key-prefi
x --ratio=1:0 --out-file=./result_small_6G_set_get.log
Writing results to ./result_small_6G_set_get.log...
[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 100%,  38 secs]  0 threads:     1920000 ops,   46353 (avg:   49346) ops/sec, 46.01MB/sec (avg: 48.98MB/sec),  4.49 (avg:  3.89) msec latency

[root@ecs-8601 ~]# cat result_small_6G_set_
result_small_6G_set_2.log    result_small_6G_set_get.log
[root@ecs-8601 ~]# cat result_small_6G_set_get.log
12        Threads
16        Connections per thread
10000     Requests per client


ALL STATS
============================================================================================================================
Type         Ops/sec     Hits/sec   Misses/sec    Avg. Latency     p50 Latency     p99 Latency    p100 Latency       KB/sec
----------------------------------------------------------------------------------------------------------------------------
Sets        54388.58          ---          ---         3.88759         1.19100        61.69500       201.72700     55282.44
Gets            0.00         0.00         0.00             ---             ---            ---             ---         0.00
Waits           0.00          ---          ---             ---             ---            ---             ---          ---
Totals      54388.58         0.00         0.00         3.88759         1.19100        61.69500       201.72700     55282.44
```



- **原生 Redis 扩容到 16G**

  原生 Redis 实例受服务器内存限制，要扩容到 16G 只能先升级 ECS 配置。需要重启服务器，存在短时间业务不可使用的问题。升级后再次使用 memtier_benchmark 插入数据依旧报错，检查发现还是出现了 OOM

```
Connection error: Connection reset by peer
Connection error: Connection reset by peer
Connection error: Connection reset by peer
Connection error: Connection reset by peer
Connection error: Connection reset by peer
Connection error: Connection reset by peer
connection dropped.
Connection error: Connection reset by peer
Connection error: Connection reset by peer
Connection error: Connection reset by peer
Connection error: Connection reset by peer
connection dropped.
connection dropped.
[RUN #1 50%, 191 secs]  0 threads:     9570078 ops,       0 (avg:   50072) ops/sec, 0.00KB/sec (avg: 49.77MB/sec),  -nan (avg:  2.75) msec latency
```

```
1555:M 18 Nov 2021 22:17:38.163 * Ready to accept connections
1555:M 18 Nov 2021 22:18:25.063 * 10000 changes in 60 seconds. Saving...
1555:M 18 Nov 2021 22:18:25.069 * Background saving started by pid 1636
1555:M 18 Nov 2021 22:18:32.309 # Background saving terminated by signal 9
1555:M 18 Nov 2021 22:18:32.414 * 10000 changes in 60 seconds. Saving...
1555:M 18 Nov 2021 22:18:32.423 * Background saving started by pid 1637
1637:C 18 Nov 2021 22:18:52.552 * DB saved on disk
1637:C 18 Nov 2021 22:18:52.555 * RDB: 228 MB of memory used by copy-on-write
1555:M 18 Nov 2021 22:18:52.576 * Background saving terminated with success
1555:M 18 Nov 2021 22:20:02.312 * 10000 changes in 60 seconds. Saving...
1555:M 18 Nov 2021 22:20:04.336 # Can't save in background: fork: Cannot allocate memory
1555:M 18 Nov 2021 22:20:57.308 * 10000 changes in 60 seconds. Saving...
Killed
[root@ecs-8601-f49b redis-5.0.14]#
```

```
Nov 18 22:18:32 ecs-8601-f49b kernel: Out of memory: Kill process 1555 (redis-server) score 657 or sacrifice child
Nov 18 22:18:32 ecs-8601-f49b kernel: Killed process 1636 (redis-server), UID 0, total-vm:11649572kB, anon-rss:10982024kB, file-rss:56kB, shmem-rss:0kB
Nov 18 22:20:59 ecs-8601-f49b kernel: uniagent invoked oom-killer: gfp_mask=0x201da, order=0, oom_score_adj=0
Nov 18 22:20:59 ecs-8601-f49b kernel: uniagent cpuset=/ mems_allowed=0
```

没办法，只能再次升级云服务器 ECS 配置到 32G，升级期间 Redis 服务再次不可用。这次升级后终于使用 memtier_benchmark 成功的插入了数据。

```
[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 100%, 172 secs]  0 threads:    19200000 ops,  140449 (avg:  111540) ops/sec, 139.41MB/sec (avg: 110.72MB/sec),  1.37 (avg:  1.70) msec latency

[root@ecs-8601 ~]# redis-cli -h 192.       -a       -p 6379
(integer) 16622382
# Memory
used_memory:18085128424
used_memory_human:16.84G
used_memory_rss:18685403136
used_memory_rss_human:17.40G
used_memory_peak:18096042280
used_memory_peak_human:16.85G
used_memory_peak_perc:99.94%
used_memory_overhead:799967880
```

五、 数据淘汰问题

下面我们来看高压力下导致数据写满的场景，直观对比双方的表现。

- **插入数据到 GaussDB(for Redis)**

  memtier_benchmark 参数设置如下，全部为写入操作，set 的单个 value 长度 50k 字节，12 个线程，每个线程 16 个客户端，每个客户端发出请求数 10000 次请求。折算下来 总的插入的 key 约为 192 万，数据量约 96G，远大于实例的规格了。

```
memtier_benchmark -s 192.XXX.XXX.XXX  -a XXXXXXX -p 8635 -c 16 -t 12  -n 10000 --random-data --randomize --distinct-client-seed -d 50000 --key-maximum=65000000 --key-minimum=1 --key-prefix= --ratio=1:0 --out-file=./result_small_6G_set.log
```

运行了一段时间后，从监控上看到 GaussDB(for Redis)磁盘空间 100%，并且实例进入只读模式拒绝新数据的写入。检查发现共导入数据 194954 条。


```
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
server 192.168.0.57:8635 handle error response: -ERR Disable write requests in read-only mode.
```

```
[root@ecs-7257-0001 ~]# redis-cli  -h 192.1[redacted]  -a [redacted]d -p 8635
192.1[redacted]:8635> dbsize
(integer) 194954
```

```
[root@ecs-7257-0001 ~]# memtier_benchmark -s 192[redacted]  -a Re[redacted]w0rd -p 8635 -c 16 -t 12  -n 10000 --random-data --randomize --distinct-client-seed -d 50000 --key-maximum=65000000 --key-minimum=1 --key-prefix= --ratio=1:0 --out-file=./result_small_6G_set.log
Writing results to ./result_small_6G_set.log...
[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 2%,  41 secs] 12 threads:      36997 ops,     762 (avg:     902) ops/sec, 36.41MB/sec (avg: 43.06MB/sec), 280.89 (avg: 211.71) msec latency
```

对于 GaussDB(for Redis)来说，当容量接近写满的时候，用户会收到告警通知，此时只需在控制台点击“磁盘扩容”，即可秒级完成扩容，对业务没有影响。


```
[root@ecs-7257-0001 ~]# redis-cli  -h 192.168.  .  -a Redi  d -p 8635
192.168.  .  :8635> dbsize
(integer) 264446
```

- **插入数据到原生 Redis**

原生 Redis 通过配置限制了内存大小为 8G，同样执行以下命令导入数据

```
memtier_benchmark -s 192.XXX.XXX.XXX  -a XXXXXXX -p 8635 -c 16 -t 12  -n 10000 --random-data --randomize --distinct-client-seed -d 50000 --key-maximum=65000000 --key-minimum=1 --key-prefix= --ratio=1:0 --out-file=./result_small_6G_set.log
```

运行一段时间后报错。

```
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
server 192.168.0.168:6379 handle error response: -OOM command not allowed when used memory > 'maxmemory'.
```

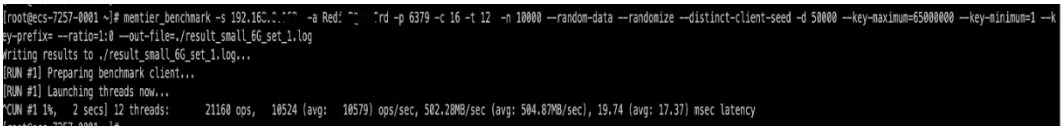
登录 redis 查看内存已写满

```
192.[redacted]:6379> config get maxmem*
1) "maxmemory"
2) "8589934592"
3) "maxmemory-samples"
4) "5"
5) "maxmemory-policy"
6) "allkeys-lru"
```

也可以通过配置 maxmemory-policy 设置数据淘汰策略保障数据写入，如图我们将淘汰策略设置成 allkeys-lru，即淘汰最近最少使用的 key 满足插入数据的内存需求；

```
192.[redacted]:6379> info  memory
# Memory
used_memory:8583831728
used_memory_human:7.99G
used_memory_rss:8964931584
used_memory_rss_human:8.35G
used_memory_peak:8608539832
used_memory_peak_human:8.02G
used_memory_peak_perc:99.71%
used_memory_overhead:8918022
used_memory_startup:791456
used_memory_dataset:8574913706
used_memory_dataset_perc:99.91%
allocator_allocated:8584281336
allocator_active:9197101056
allocator_resident:9380802560
```

修改配置后 插入正常



综上，**GaussDB(for Redis)更加看重数据安全，将“保障用户数据不丢”作为最高优先级**。当数据写满后自动进入只读模式，确保实例中数据的安全。通过控制台可以做到快速的扩容，最大可能降低对业务的影响。 原生 Redis 提供了数据淘汰参数，用户可自主选择策略当数据写满后淘汰符合条件的数据，设计思想更偏向于缓存的用途“数据可随意丢弃”。如使用在重要的业务场景，不希望数据丢失，建议选择 GaussDB(for Redis)。

六、 测试总结

本次我们使用 memtier_benchmark 分别对 GaussDB(for Redis) 和原生 Redis 进行 set 操作的测试，8G 规格的 GaussDB(for Redis) 很顺利的完成了数据加载的操作，原生 Redis 出现 OOM 异常导致数据加载失败。原生 Redis 通过 fork 进程 copy-on-write 的方式拷贝数据，在 RDB 快照、aof 重写以及新增从库等操作时容易出现 OOM 异常。**反观 GaussDB(for Redis) 由于摒弃了 fork 机制，使得架构更健壮，服务的可用性更强。**

在后续的扩容操作中 GaussDB(for Redis)能够快速完成且对业务 RW 操作无影响，而原生 Redis 扩容需停服，期间业务无法正常使用。**GaussDB(for Redis)快速扩容的特性非常适合生产环境中需要紧急扩容的场景，如游戏开服、电商抢购的火爆程度远超预期时。从测试的情况看，扩容几乎达到了秒级完成，且扩容过程中对业务的读写完全没有影响。**

另外更重要的原生 Redis 无论采用 RDB 还是 aof 方式进行数据持久化，都有数据丢失的风险，而 GaussDB(for Redis)支持全量数据落盘，GaussDB 基础组件服务提供底层数据三副本冗余保存，能够保证数据零丢失。

如果使用场景既要满足 KV 查询的高性能，又希望数据得到重视能够不丢，建议从原生 Redis 迁移到 GaussDB(for Redis) 。

GaussDB(for Redis)免费体验：企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，**点击体验**

【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专

业技能！**点击前往**

大 Key 来袭，GaussDB(for Redis)处惊不变

在前一篇文章《现身说法：GaussDB(for Redis)的大肚量与真稳定》中，我们使用多线程压测工具 memtier_benchmark 对华为 GaussDB(for Redis)和原生 Redis 进行了对比压测，发现原生 Redis 容易出现 OOM 故障，且扩容操作会很慢，给运维带来很大压力。反观华为 GaussDB(for Redis)不仅性能稳定，还具备在压测过程中秒级扩容的能力，扩容操作对业务读写无影响。

华为 GaussDB(for Redis)支持全量数据落盘，GaussDB 基础组件服务提供底层数据三副本冗余保存，能够保证数据零丢失。如果使用场景既要满足 KV 查询的高性能，又希望数据得到重视能够不丢，则华为 GaussDB(for Redis)是合适的选型。

我们大多在实际生产环境中都遇到过 big key 对 Redis 性能的严重影响。接下来我们通过几个简单的实验，测试下对于大 key 这一“性能杀手”，GaussDB(for Redis)的表现怎样，和原生 Redis 相比在性能上有哪些改进？

一、 访问大 key

首先分别在 GaussDB(for Redis)和原生 redis 中创建一个大的 hash 类型的 key。

编辑一个简单的 lua 脚本，向一个 hash key 中插入一千万条数据。

```
# vim redis-bigkey.lua
local result
for var=1,10000000,1 do
redis.call('hset',KEYS[1],var,var)
redis.call('lpush',KEYS[2],var)
redis.call('sadd',KEYS[3],var)
end
return result
```

向 GaussDB(for Redis)【192.168.0.226:8635】中插入大 key

```
# redis-cli  -h  192.168.0.226 -a XXXXXXX -p 8635    --eval /root/redis-bigkey.lua 3 hset_test list_test set_test
(nil)

# redis-cli  -h  192.168.0.226 -a XXXXXXX -p 8635 hlen hset_test
(integer) 10000000
# redis-cli  -h  192.168.0.226 -a XXXXXXX -p 8635 scard sadd_test
(integer) 10000000
# redis-cli  -h  192.168.0.226 -a XXXXXXX -p 8635 llen lpush_test
(integer) 10000000
```

向原生 Redis【192.168.0.135】中插入大 key

```
# redis-cli  -h  192.168.0.135 -a XXXXXXX -p 6379    --eval /root/redis-bigkey.lua 3 hset_test list_test set_test
(nil)

# redis-cli  -h  192.168.0.135 -a XXXXXXX -p 6379 hlen hset_test
(integer) 10000000
# redis-cli  -h  192.168.0.135 -a XXXXXXX -p 6379 scard sadd_test
(integer) 10000000
# redis-cli  -h  192.168.0.135 -a XXXXXXX -p 6379 llen lpush_test
(integer) 10000000
```

使用 memtier_benchmark 模拟业务压力的同时 对大 key 进行访问，观察对业务 qps 的影响。

编辑一个简单 shell 脚本，对大 key 进行频繁的访问

```
#!/bin/bash
while true
do
redis-cli  -h  192.168.0.135 -a XXXXXXX -p 6379 hget hset_test $RANDOM
redis-cli  -h  192.168.0.226 -a XXXXXXX -p 8635 hget hset_test $RANDOM
redis-cli  -h  192.168.0.135 -a XXXXXXX -p 6379 LREM lpush_test 0 $RANDOM
redis-cli  -h  192.168.0.226 -a XXXXXXX -p 8635 LREM lpush_test 0 $RANDOM
redis-cli  -h  192.168.0.135 -a XXXXXXX -p 6379 SISMEMBER sadd_test $RANDOM
redis-cli  -h  192.168.0.226 -a XXXXXXX -p 8635 SISMEMBER sadd_test $RANDOM
done
```

使用 memtier_benchmark 进行压测，读写混合场景。通过反馈的性能数据可以看到 GaussDB(for Redis) 和 Redis 每秒的操作数 ops/sec 分别为 14 万和 13 万，差别不大。

```
#GaussDB(for Redis)
memtier_benchmark -s 192.168.0.226 -a XXXXXXX -p 8635 -c 16 -t 12  -n 100000 --random-data --randomize --distinct-client-seed -d 1000 --key-maximum=65000000 --key-minimum=1 --key-prefix= --ratio=1:1 --out-file=./result_gauss_setget.log
```

```
[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 3%,   4 secs] 12 threads:        582045 ops,   144053 (avg:   145472) ops/sec, 21.16MB/sec (avg: 21.51MB/sec),  1.33 (avg:  1.32) msec latency
#原生 Redis
memtier_benchmark -s 192.168.0.135 -a XXXXXXX -p 6379 -c 16 -t 12  -n 100000 --random-data --randomize --distinct-client-seed -d 1000 --key-maximum=65000000 --key-minimum=1 --key-prefix= --ratio=1:1 --out-file=./result_redis_setget.log

[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 7%,  11 secs] 12 threads:       1430798 ops,   132637 (avg:   130051) ops/sec, 70.51MB/sec (avg: 68.79MB/sec),  1.44 (avg:  1.47) msec latency
```

启动 shell 脚本后再次观察，**发现 GaussDB(for Redis) 的每秒操作数几乎无变化，而原生 Redis 的每秒操作数波动巨大，甚至降低到了 3k 左右。说明大 key 操作对原生 Redis 性能有较大影响，对 GaussDB(for Redis) 的影响可控。**

```
# bash hget_bigkey.sh

#GaussDB(for Redis)
# memtier_benchmark -s 192.168.0.226 -a XXXXXXX -p 8635 -c 16 -t 12  -n 100000 --random-data --randomize --distinct-client-seed -d 1000 --key-maximum=65000000 --key-minimum=1 --key-prefix= --ratio=1:1 --out-file=./result_gauss_setget.log

[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 47%,  64 secs] 12 threads:       9099444 ops,   139186 (avg: 142163) ops/sec, 20.60MB/sec (avg: 20.96MB/sec),  1.38 (avg:  1.35) msec latency
#原生 Redis
# memtier_benchmark -s 192.168.0.135 -a XXXXXXX -p 6379 -c 16 -t 12  -n 100000 --random-data --randomize --distinct-client-seed -d 1000 --key-maximum=65000000 --key-minimum=1 --key-prefix= --ratio=1:1 --out-file=./result_redis_setget.log

[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 29%,  75 secs] 12 threads:       5607700 ops,     3329 (avg: 74759) ops/sec, 1.80MB/sec (avg: 40.08MB/sec), 52.35 (avg:  2.55) msec latencyy
```

二、 删除大 key

继续使用 memtier_benchmark 对 GaussDB(for Redis) 和原生 Redis 进行测试，只读场景。**在 GaussDB(for Redis)中删除大 key 很快就能完成，且对性能几乎无影响。**

```
#GaussDB(for Redis)
# memtier_benchmark -s 192.168.0.226 -a XXXXXXX -p 8635 -c 16 -t 12  -
n 100000 --random-data --randomize --distinct-client-seed -d 1000 --
key-maximum=65000000 --key-minimum=1 --key-prefix= --ratio=0:1 --
out-file=./result_gauss_setget.log
Writing results to ./result_gauss_setget.log...
[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 4%,   5 secs] 12 threads:        719216 ops,   151326 (avg:   143795)
ops/sec, 22.16MB/sec (avg: 21.13MB/sec),  1.27 (avg:  1.33) msec latency
# time redis-cli  -h  192.168.0.226 -a XXXXXXX -p 8635 del sadd_test
(integer) 1

real    0m0.003s
user    0m0.001s
sys     0m0.002s
# memtier_benchmark -s 192.168.0.226 -a XXXXXXX -p 8635 -c 16 -t 12  -
n 100000 --random-data --randomize --distinct-client-seed -d 1000 --
key-maximum=65000000 --key-minimum=1 --key-prefix= --ratio=0:1 --
out-file=./result_gauss_setget.log
Writing results to ./result_gauss_setget.log...
[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 42%,  57 secs] 12 threads:       8031731 ops,   144874 (avg:
140890) ops/sec, 21.46MB/sec (avg: 20.77MB/sec),  1.32 (avg:  1.36) msec
latency
```

反观原生 Redis，删除大 key 耗时 3 秒，且在删除期间对性能影响较大。可以观察到在删除期间 ops/sec 变成 0，也就是说大 key 删除期间操作是没有办法正常响应的。

```
#原生 Redis
# memtier_benchmark -s 192.168.0.135 -a XXXXXXX -p 6379 -c 16 -t 12  -
n 100000 --random-data --randomize --distinct-client-seed -d 1000 --
key-maximum=65000000 --key-minimum=1 --key-prefix= --ratio=0:1 --
out-file=./result_redis_setget.log
Writing results to ./result_redis_setget.log...
[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 6%,   7 secs] 12 threads:       1107132 ops,   157621 (avg:   158125)
ops/sec, 16.07MB/sec (avg: 16.13MB/sec),  1.22 (avg:  1.21) msec latency

# time redis-cli  -h  192.168.0.135 -a XXXXXXX -p 6379 del sadd_test
```

```
(integer) 1

real    0m3.001s
user    0m0.000s
sys     0m0.003s

# memtier_benchmark -s 192.168.0.135 -a XXXXXXX -p 6379 -c 16 -t 12  -
n 100000 --random-data --randomize --distinct-client-seed -d 1000 --
key-maximum=65000000 --key-minimum=1 --key-prefix= --ratio=0:1 --
out-file=./result_redis_setget.log
Writing results to ./result_redis_setget.log...
[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 57%,  68 secs] 12 threads:       1015893 ops,          0 (avg:  126961)
ops/sec, 0.00KB/sec (avg: 12.98MB/sec),  -nan (avg:  1.13) msec latencyy
```

手动删除大 key 对性能的影响差别明显，如果设置大 key 的过期时间交由 Redis 删除过期数据 是否会有性能影响呢？下面简单测试下

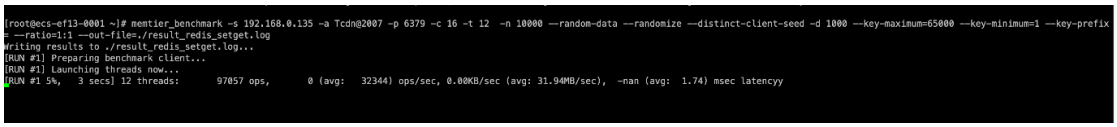
手动设置大 key 的过期时间，并启动 memtier_benchmark 读写混合测试，查看对性能的影响。**通过测试发现大 key 的过期对于 GaussDB(for Redis)的性能几乎没有影响。**

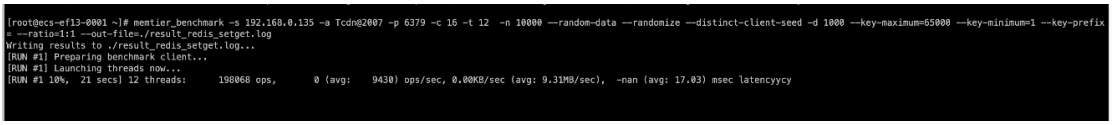
```
#GaussDB(for Redis)
[root@ecs-ef13-0001 ~]#  redis-cli  -h  192.168.0.226 -a XXXXXXX -p
8635 EXPIRE sadd_test 8 && redis-cli  -h  192.168.0.226 -a XXXXXXX -p
8635 EXPIRE sadd_test1 12
(integer) 1
(integer) 1

[root@ecs-ef13-0001 ~]# memtier_benchmark -s 192.168.0.226 -a
XXXXXXX -p 8635 -c 16 -t 12  -n 10000 --random-data --randomize --
distinct-client-seed -d 1000 --key-maximum=65000 --key-minimum=1 --
key-prefix= --ratio=1:1 --out-file=./result_gauss_setget.log
Writing results to ./result_gauss_setget.log...
[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 100%,  17 secs]  0 threads:       1920000 ops,  106367 (avg:
109940) ops/sec, 105.02MB/sec (avg: 108.55MB/sec),  1.74 (avg:  1.74)
msec latency
```

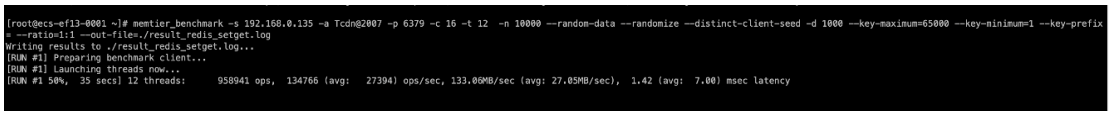
在对原生 Redis 测试时，我们发现大 key 过期操作几乎阻塞了正常的读写，在 memtier_benchmark 测试时 ops/sec 指标为 0，只有当大 key 过期操作结束后才恢复正常。

```
#原生 Redis
```

```
[root@ecs-ef13-0001 ~]# redis-cli  -h  192.168.0.135 -a XXXXXXX -p 6379
EXPIRE sadd_test 8 && redis-cli  -h  192.168.0.135 -a XXXXXXX -p 6379
EXPIRE sadd_test1 12
(integer) 1
(integer) 1
```








```
[root@ecs-ef13-0001 ~]# memtier_benchmark -s 192.168.0.135 -a
XXXXXXX -p 6379 -c 16 -t 12  -n 10000 --random-data --randomize --
distinct-client-seed -d 1000 --key-maximum=65000 --key-minimum=1 --
key-prefix= --ratio=1:1 --out-file=./result_redis_setget.log
Writing results to ./result_redis_setget.log...
[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 100%,  42 secs]  0 threads:  1920000 ops,  134502 (avg:
45551) ops/sec, 132.80MB/sec (avg: 44.98MB/sec),  1.43 (avg:  4.21) msec
latency
```

开源 redis 的过期虽然也支持异步，但需要用户手动配置策略；删除操作则需用 UNLINK 替换常规的 DEL，具体对性能的影响可能会有所降低，本次不做深入验证。华为 GaussDB(for Redis)的删除和过期对性能 0 影响。

三、 大 key 对 GaussDB(for Redis)扩容操作的影响

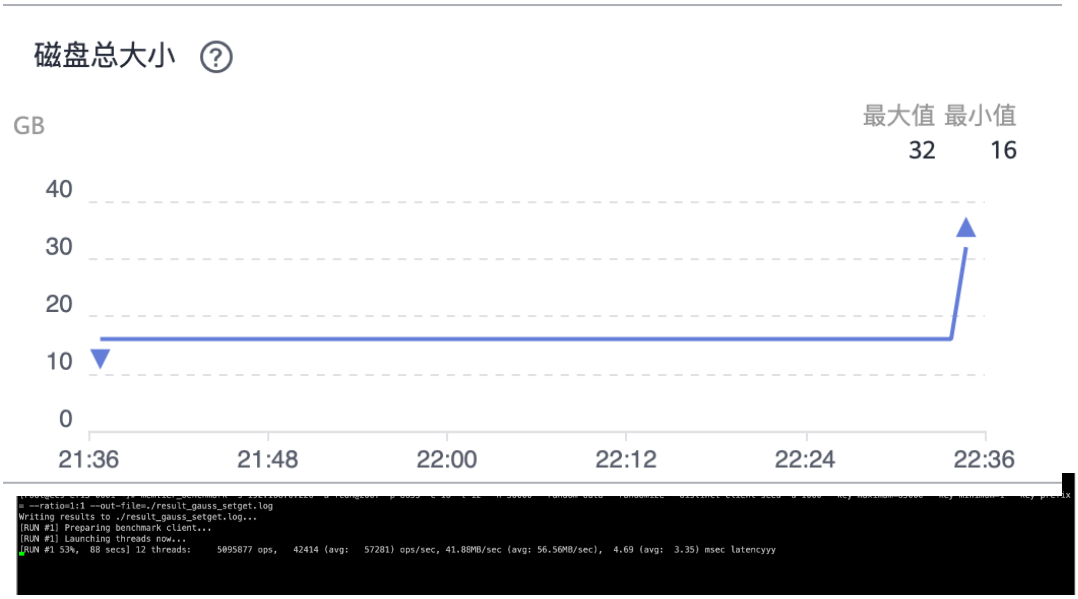
在上一篇文章中我们测试了 GaussDB(for Redis)的在线扩容功能，经测试 GaussDB(for Redis)可以在不影响业务读写的前提下实现秒级的扩容。这次我们增加一些“难度”，看看存在大 key 的情况下 GaussDB(for Redis)扩容操作是否还能做到秒级和业务零感知。

预先在 GaussDB(for Redis)中插入大 key

```
[root@ecs-ef13-0001 ~]# redis-cli  -h  192.168.0.226 -a XXXXXXX -p
8635 scard sadd_test4
(integer) 10000000
[root@ecs-ef13-0001 ~]# redis-cli  -h  192.168.0.226 -a XXXXXXX -p
8635 scard sadd_test5
(integer) 10000000
[root@ecs-ef13-0001 ~]# redis-cli  -h  192.168.0.226 -a XXXXXXX -p
8635 scard sadd_test6
(integer) 10000000
[root@ecs-ef13-0001 ~]# redis-cli  -h  192.168.0.226 -a XXXXXXX -p
8635 scard sadd_test7
(integer) 10000000
[root@ecs-ef13-0001 ~]# redis-cli  -h  192.168.0.226 -a XXXXXXX -p
8635 scard sadd_test8
(integer) 10000000
```

使用 memtier_benchmark 模拟读写请求，同时在控制台上进行扩容操作。同之前的测试效果一样，**GaussDB(for Redis)同样实现了对业务零感知的秒级扩容**

```
[root@ecs-ef13-0001 ~]# memtier_benchmark -s 192.168.0.226 -a
XXXXXXX -p 8635 -c 16 -t 12  -n 50000 --random-data --randomize --
distinct-client-seed -d 1000 --key-maximum=65000 --key-minimum=1 --
key-prefix= --ratio=0:1 --out-file=./result_gauss_setget.log
Writing results to ./result_gauss_setget.log...
[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 9%,  20 secs] 12 threads:        902361 ops,   42634 (avg:   45112)
ops/sec, 41.99MB/sec (avg: 44.44MB/sec),  4.53 (avg:  4.23) msec
latencycy
```



四、 总结

原生 Redis 对大 key 的访问，删除等操作会严重阻塞业务的正常访问，这是由 Redis 自身单线程处理请求的架构决定的。使用原生 Redis 时需要严格限制大 key 的使用，一旦出现大 key 对系统的性能影响通常是“致命”的。

反观 GaussDB(for Redis)由于采用多线程架构，对大 key 的访问、删除，以及存在大 key 情况下的扩容操作，对性能的影响都是可控的。

1) 大 key 访问场景中，由于 GaussDB(for Redis)采用的多线程的架构，不易阻塞其他业务操作。
2) 大 key 删除的场景中，由于 GaussDB(for Redis)实现的逻辑不同，删除操作能够快速完成，对业务无影响。
3) 扩容场景中，GaussDB(for Redis)不涉及 key 迁移，大 key 对扩容更是 0 影响。

综上，**虽然一般推荐业务设计避免大 key，但在一些需要操作少量大 key 的业务场景，华为云 GaussDB(for Redis)表现更佳。**

此外，从业务开发角度看，当多业务共用一个实例时，使用 GaussDB(for Redis)的话，即使其他业务引入大 key，自己的业务也不至于受太大影响。

GaussDB(for Redis)免费体验： 企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，**点击体验**

【 数据库论坛 】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！**点击前往**

应用篇

华为云 GaussDB NoSQL 云原生多模数据库的超融合实践

引言

本届中国数据库技术大会（DTCC），不管是公有云数据库厂商，还是传统数据库厂商，都在关注云原生的技术实践，分享了很多存算分离、DBaaS、Kubernetes 等与数据库深度结合的案例。

在 NoSQL 专题会场，热度最高的一场演讲来自华为云数据库团队，由广入深，干货满满。

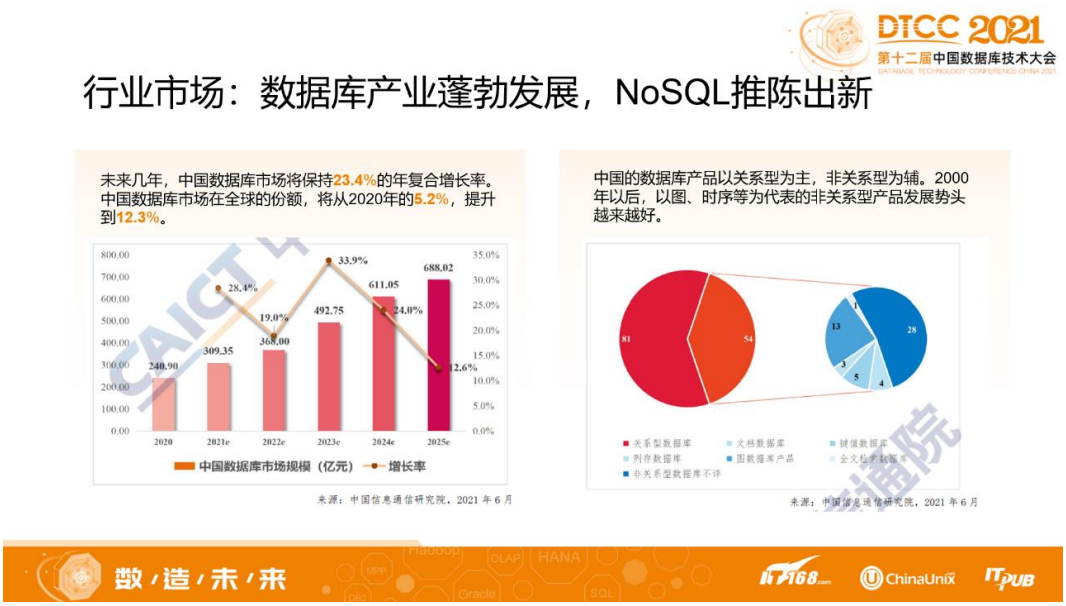
本文整理自华为云数据库 NoSQL 架构师余汶龙的专题分享——云原生多模数据库 GaussDB NoSQL 架构设计，总结了当前数据库的发展趋势、GaussDB NoSQL 关键技术解密以及核心竞争力。

数据库发展趋势

一、 行业市场

中国信通院最新研究透露出两个重要信息：

- 未来几年，中国数据库市场将保持 23.4%的年复合增长率，中国数据库市场在全球的份额，将从 2020 年的 5.2%提升到 12.3%；
- 中国的国产数据库产品虽然以关系型为主，非关系型为辅，但从 2000 年以后，以图、时序等为代表的非关系型产品发展势头越来越好，截止 2020 年底，国产 NoSQL 数据库厂商已经占到了 40%。



二、 行业趋势

受大环境的影响，国内金融、电信、政企等行业为防止潜在的供应链风险，技术层面存在国产化需求，这使得我们的国产数据库产业进入蓬勃发展的初期。

但我国数据库行业发展还面临 2 个核心问题：

- 如何缩小“**高要求的存量数据应用**”与“**仍处于发展初期阶段的供给能力**”之间的差距；
- 如何匹配“**创新型数据应用**”与“**数据库技术演进**”的合理映射关系。

如何回答上述两个问题，可以从中国信通院最新的趋势总结里找到答案：“多模实现一库多用，简化开发运维成本”、“云原生降低硬件依赖，更方便的享受新兴技术红利”。

因此，为了更好的兼容历史数据应用（比如原先用 Redis），并支持好未来的创新应用（新增 Influx），可以在多模与云原生领域提前做好相关布局。

三、 云原生数据库演进方向

数据库的发展，按传统物理机部署到云化，大概经历了三代。

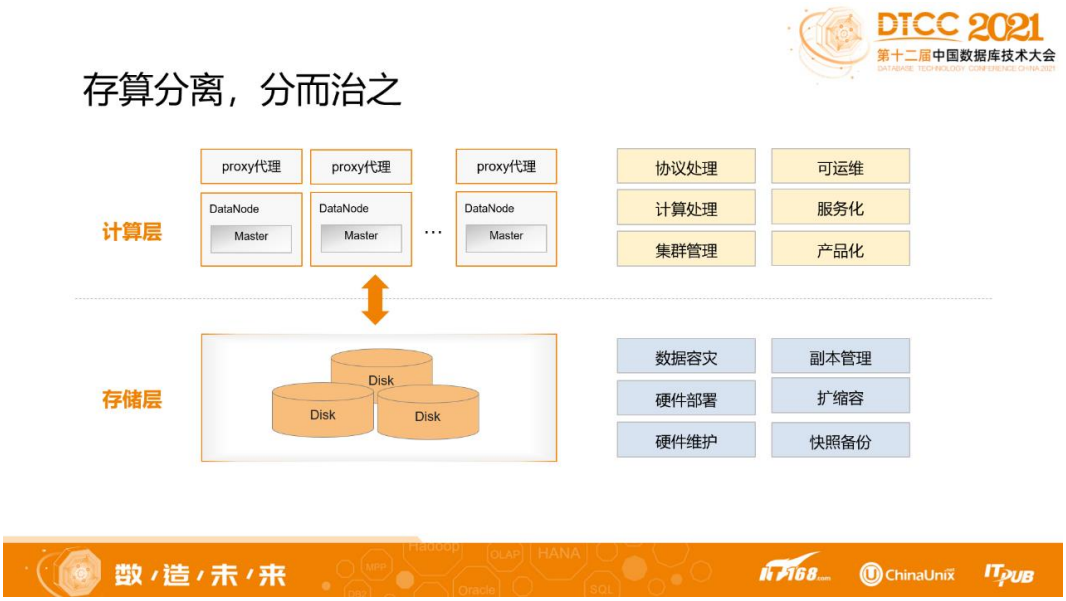
- 第一代是纯物理机、裸硬盘部署，从业人员必须关心硬件的各种细节，包括机型、系统、硬盘、组网等等；
- 第二代是云化的初级阶段，从业人员把数据库部署从物理机，迁移到虚拟机 VM，把物理硬盘，换成了云盘 EVS。但这一代有个明显的缺点，EVS 是个 3 副本可靠的服务，再加上数据库自身的高可用，那么存储成本就放大了 3 倍；并且备机其实是资源浪费的，没有提供服务；
- 第三代是云化的高级阶段，这个阶段将数据库的资源，彻底分成存储和计算两层，其中计算资源部署在更轻量级的容器之上，而存储资源部署在分布式存储池之上。很显然，这是与云原生结合更彻底的方式，充分享受了架构的弹性、便捷，而且轻松实现了多点读写的全负荷分担能力。

四、 存算分离，分而治之

云原生数据库有两个重要的特点。首先是存算分离。

存算分离是一种分层的设计思想：

- 从逻辑到功能进行明确的划分，让计算层更聚焦服务、产品、协议处理等事件；
- 存储层更聚焦数据本身的复制、安全、扩缩容等等。

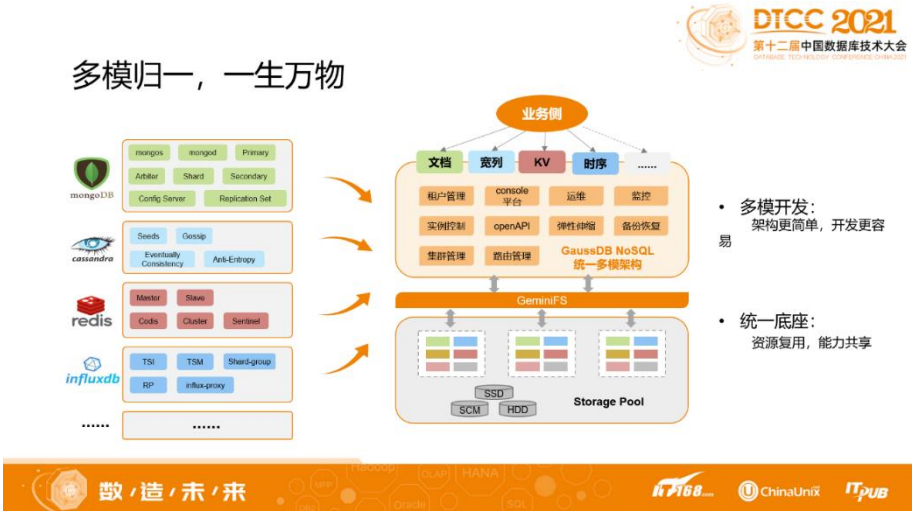


五、 多模归一，一生万物

云原生数据库第二个重要的特点，是多模。

多模实际上是一种“归一”，也是一种“派生”。以大家熟悉的 NoSQL 为例，MongoDB 是有 Mongod/Mongos/Config 等组件，而对应的 Cassandra 其实也有 Coordinate Node/Data Node 等组件。虽然这些组件名字不同，但背后做的事情是一样的，即：集群管理、副本管理、扩缩容管理、以及管控等功能。

其实，完全可以把这些功能抽象成统一的架构，即“多模归一”。在这套统一架构之上，我们再新增别的引擎就很容易了。可以快速复用当前的成熟架构，提供不同的协议接口即可，即“一生万物”。



六、 GaussDB NoSQL 概况

接下来介绍这次分享的主角——云原生多模数据库 GaussDB NoSQL。

当前 GaussDB NoSQL 已经支持 MongoDB、Cassandra、Redis、InfluxDB 等 4 款引擎；全球客户 1000+，足迹遍布金融、政府、电信、互联网等行业；总数据量超过 10PB，每日新增超过 10TB。

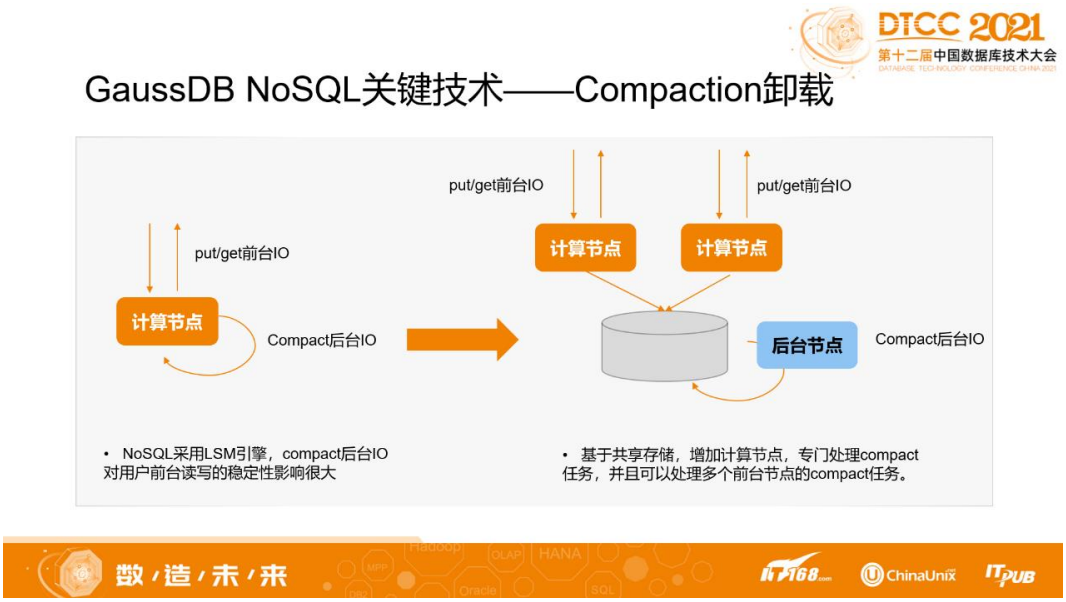
七、 GaussDB NoSQL 关键技术

1. Compaction 卸载

GaussDB NoSQL 采用 LSM 做存储引擎，正常情况下，前台的读写会受到后台的 Compaction 任务的影响，从而导致时延抖动。

因此，我们设计了单独的 Compaction 任务节点，通过共享的方式，访问用户的数据并进行 Compact，再将 Compact 的结果应用到用户的可见版本中。这样做的话，就将用

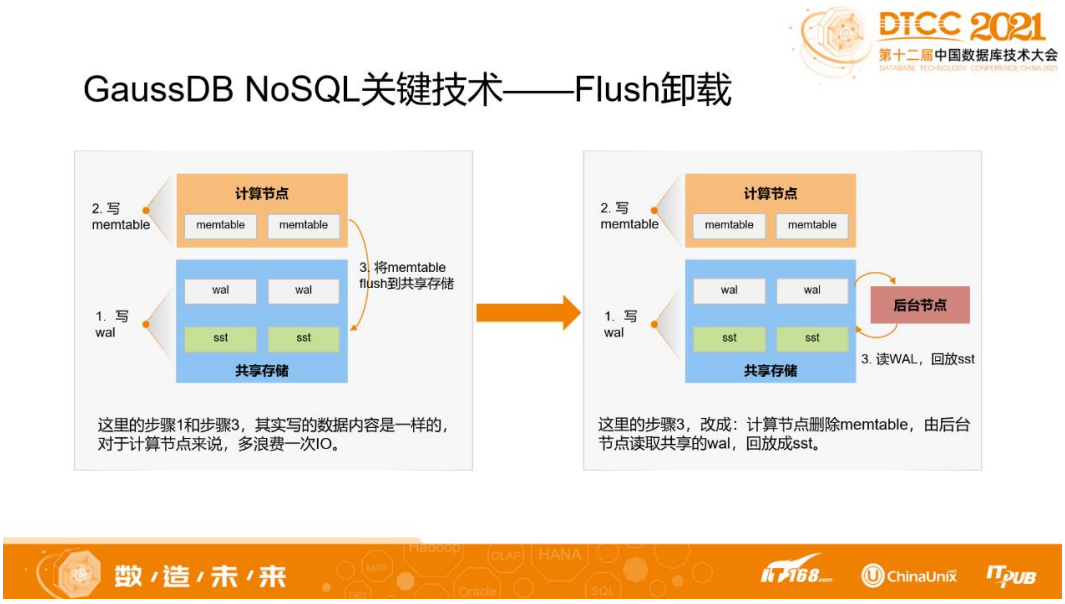
户前台的 IO 和后台 IO 分离，解决了时延抖动问题。



2. Flush 卸载

根据 LSM 引擎的写入流程，可以知道，一个数据要写入 DB 中，需要经历两次 IO：

- 写 WAL
- flush memtable



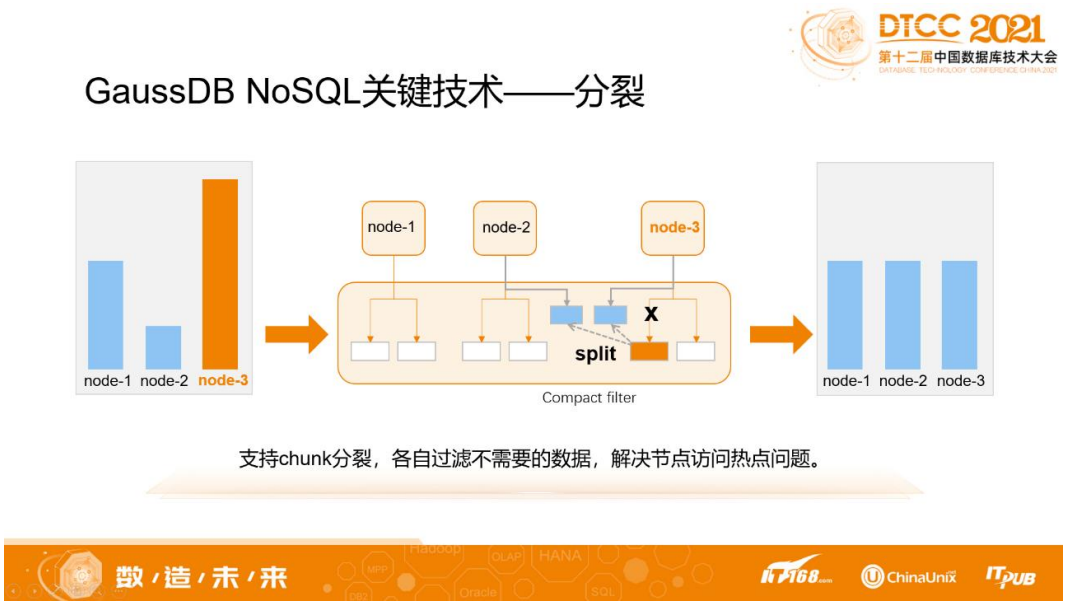
而这两次 IO 写的其实是相同数据，完全可以省掉一次。因此，我们借助共享存储的能

力，独立出一个后台任务节点。当用户前台节点需要 flush memtable 的时候，由后台任务节点读取 WAL，并转化成 L0 层的 SST，再应用版本，并通知前台删除 memtable。这样就极大节省了用户前台的 IO 开销。

3. 分裂

GaussDB NoSQL 在分片策略上，采取的是 Hash + Range 的结合方式，因此扩容或处理热点的时候会很灵活。

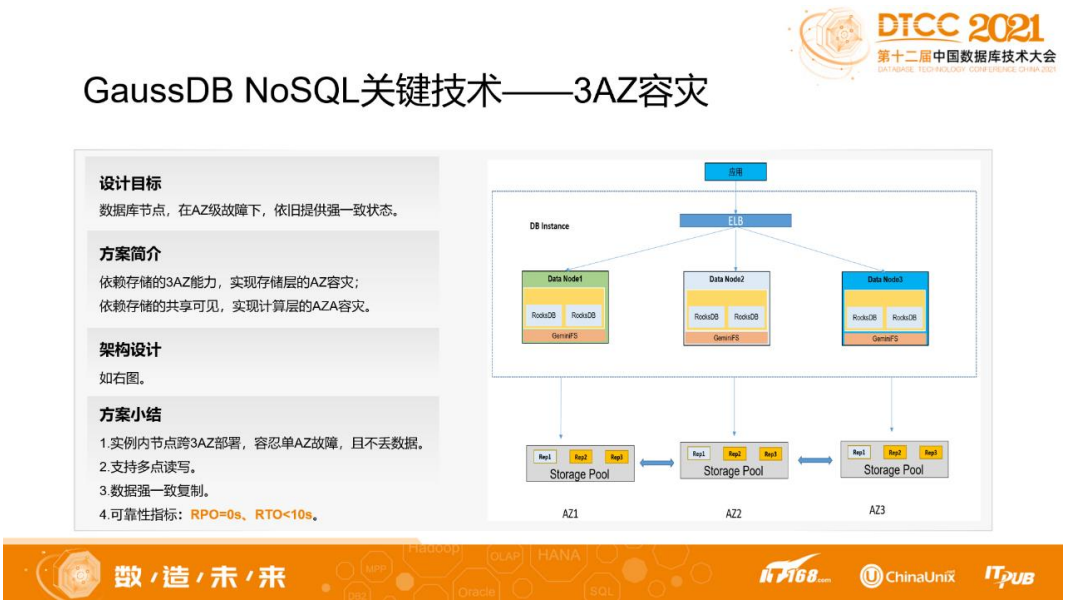
比如，当 chunk 数量足够多时，只需要移动 chunk 就可以扩容；而当某个 chunk 成为访问热点时，对它做分裂就可以解决局部热点问题。



3AZ 容灾

作为数据库产品，容灾特性是很重要的，它可以避免极端情况给用户业务带来的灾难性损失。

GaussDB NoSQL 有统一的容灾设计，即存储和计算可以实现 3AZ 部署，同时存储层数据实现 3 副本强一致复制。因此在任意时间，挂掉了任意机房的存储，都不会丢数据；而挂掉计算，也会被其他 AZ 的计算节点接管元数据，不会让访问完全中断。



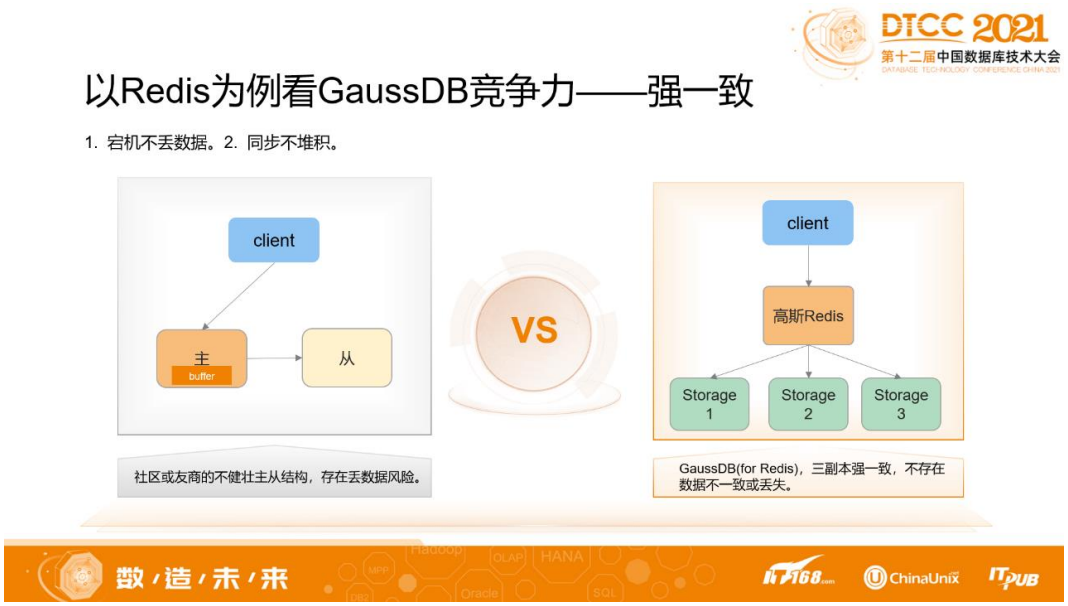
八、 以 Redis 为例看 GaussDB 竞争力

接下来，以使用最广泛的 NoSQL 引擎 Redis 为例，具体介绍 GaussDB NoSQL 的优势。

1. 强一致

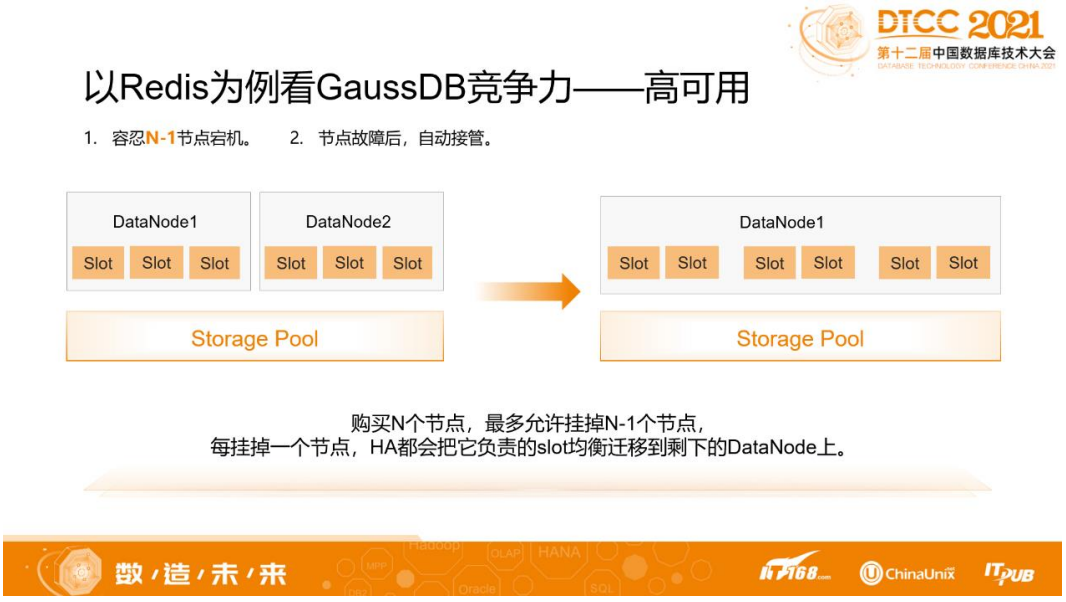
社区版 Redis，主从复制是异步的，容易造成数据堆积，也有宕机丢数据风险。

GaussDB(for Redis)（下文简称高斯 Redis）则是采用强一致同步的，当用户的数据写入高斯 Redis 并收到返回 OK，这意味着高斯 Redis 已经实现了强一致的复制，数据的安全性很高。当然，这里的复制过程采用了组提交、用户态文件系统、RDMA 等技术来降低同步复制的时延。



2. 高可用

高斯 Redis 的数据存储是共享的，即 Shared Everything，因此可以容忍最多 N−1 个节点故障，而不影响数据的访问。



3. 弹性伸缩

高斯 Redis 实现了分层弹性，将资源准确的划分成计算资源、存储资源，真正做到了按需扩容：

- 当用户的计算不足时，只需要扩展计算节点；

- 当存储空间不够时，只需要扩展存储空间即可。

同时，扩容过程也足够流畅：

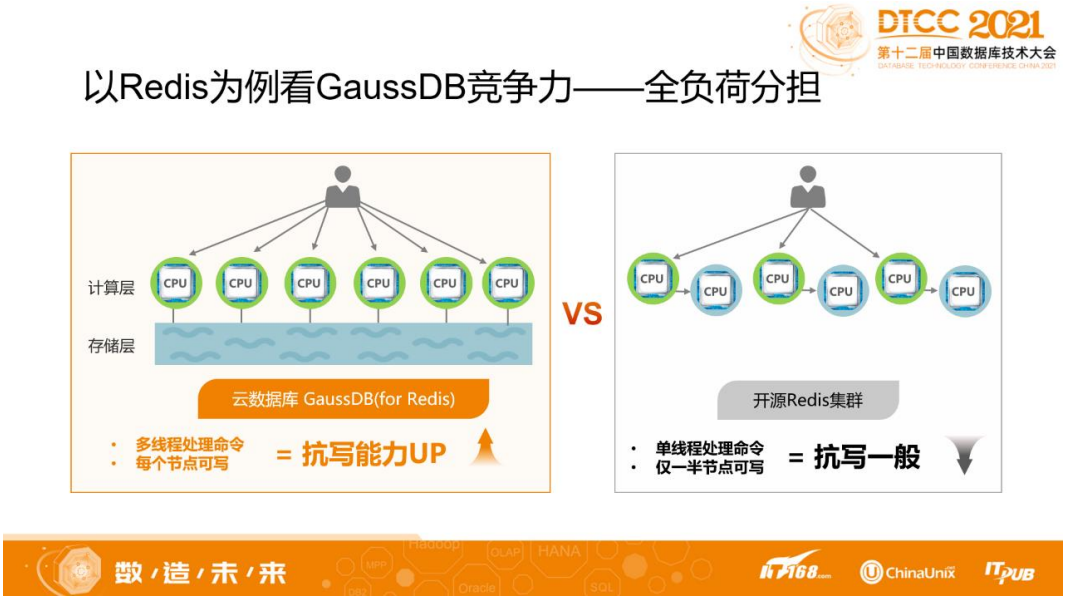
- 计算扩容的过程，不需要拷贝数据，只需要修改路由映射即可，对业务侧的影响很小；
- 存储扩容更简单，只需要修改配额即可，对业务侧零影响。

所以计算、存储的扩容都足够轻量级，可极速完成且对业务干扰极小。

4. 全负荷分担

存算分离的设计，让我们把数据复制交给了存储，计算层则完全解放。

每个节点都可以承担用户的读写请求，这跟开源 Redis 的主上读写来比较，实现了 2 倍扩展。



GaussDB(for Redis)免费体验：企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，[点击体验]

九、 总结

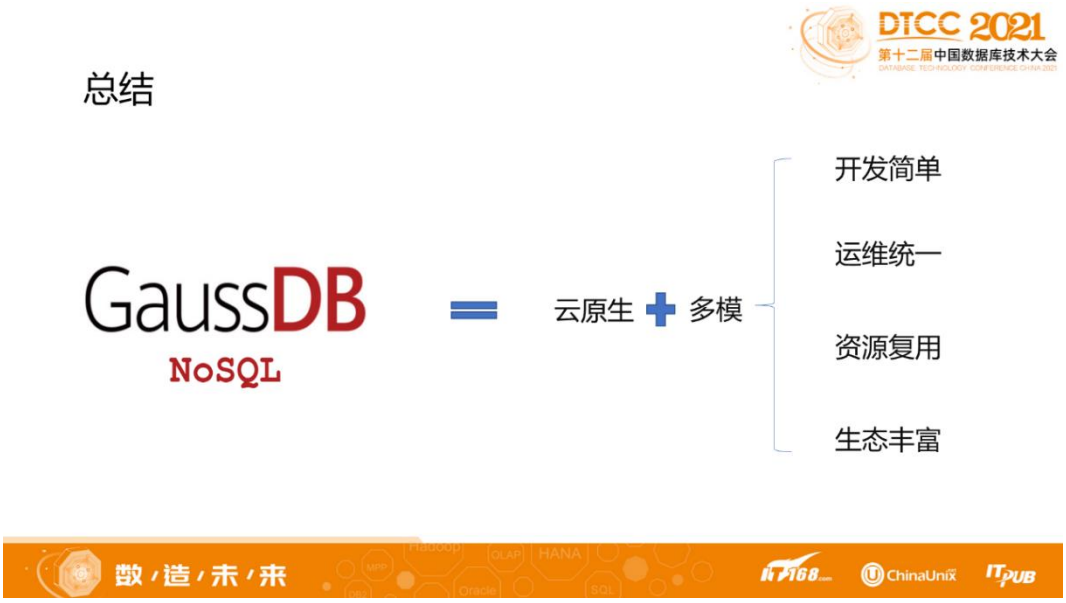
- **云原生是技术趋势**

云原生是大势所趋，越来越多厂商和从业者都在提倡云原生，而华为云 GaussDB NoSQL 不仅仅基于云原生，还实现了多模架构，实现了多副本强一致、高可用、弹性伸缩、高性能等能力，以及具备资源复用、开发运维统一等好处。

- **华为云 GaussDB NoSQL 提供超融合数字化解决方案**

华为云 GaussDB NoSQL 的多模特性，提供高并发、低时延的 Redis，助力秒杀、推荐、热搜等场景；提供大容量、高频写的 Cassandra，助力海量存储以及检索等场景；提供非结构化、灵活扩展的 MongoDB，助力大数据分析、交易等场景；提供时序特征的 InfluxDB，助力边缘计算、工业生产、实时监控等场景。

以上场景涵盖数字工业的方方面面，提供了完整的一体化解决方案，方便用户一站式使用。



【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！[点击前往]

华为云企业级 Redis：助力 VMALL 打造先进特征平台

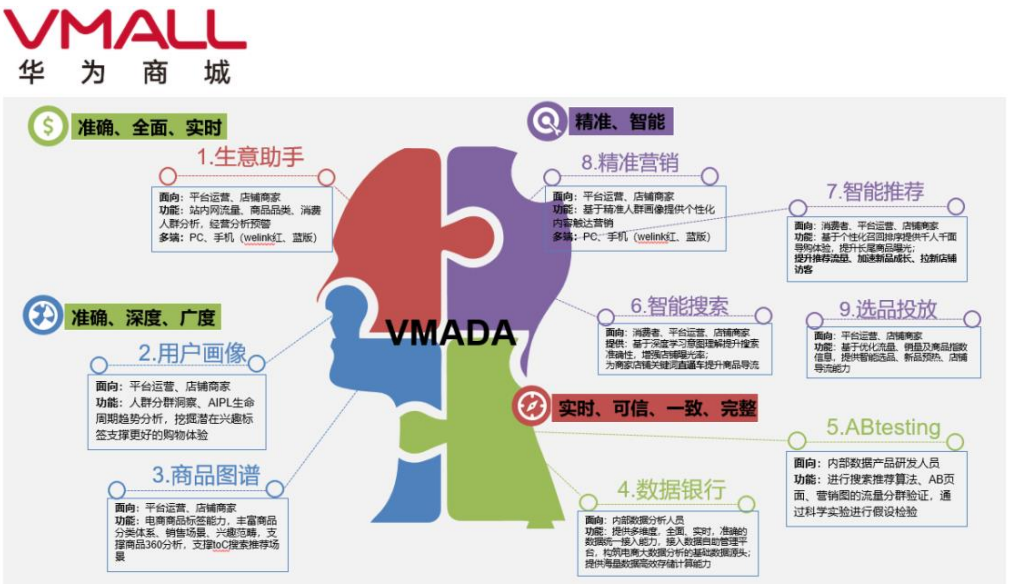
一、 客户介绍

华为商城（VMALL）是华为公司旗下自营及精选好物的官方电商平台，本着“智慧生活、精选好物”的理念，为消费者提供最齐全的华为品牌产品及鸿蒙生态产品，覆盖了办公、出行、居家、运动、娱乐等生活需求，致力于将全场景智慧生活带给更多的消费者。

云数据库 GaussDB(for Redis)作为华为云旗下企业级 Redis，致力于为客户提供稳定可靠、超高并发，且能够极速弹性扩容的 KV 存储服务。GaussDB(for Redis)在 VMALL 特征工程平台建设中，起到了关键作用。

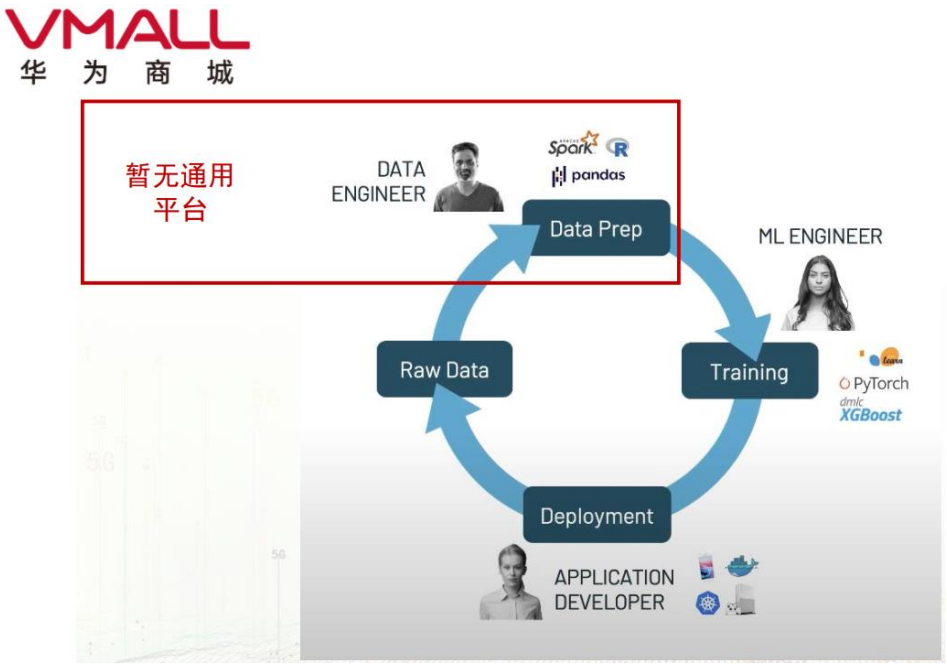
二、 业务痛点

VMALL 使用了大量的 AI 和大数据技术，用来支撑智能推荐、精准营销、智能搜索、选品投放等业务的高效开展。



随着业务的快速发展，系统对 AI 算法模型的需求日益增多。当前的 AI 开发流程中，“模型训练”和“模型部署”阶段都已经有成熟的平台支撑，唯独“特征数据准备”阶段缺乏通用平台，导致了“线上推理和线下训练的特征数据不一致”、“各算法模型独立开发，特征生产重复造轮子”、“特征工程投入时间多（占据算法开发耗时的 60%-70%）”3 个关键问题，严重影响了研发效率，阻碍业务发展。

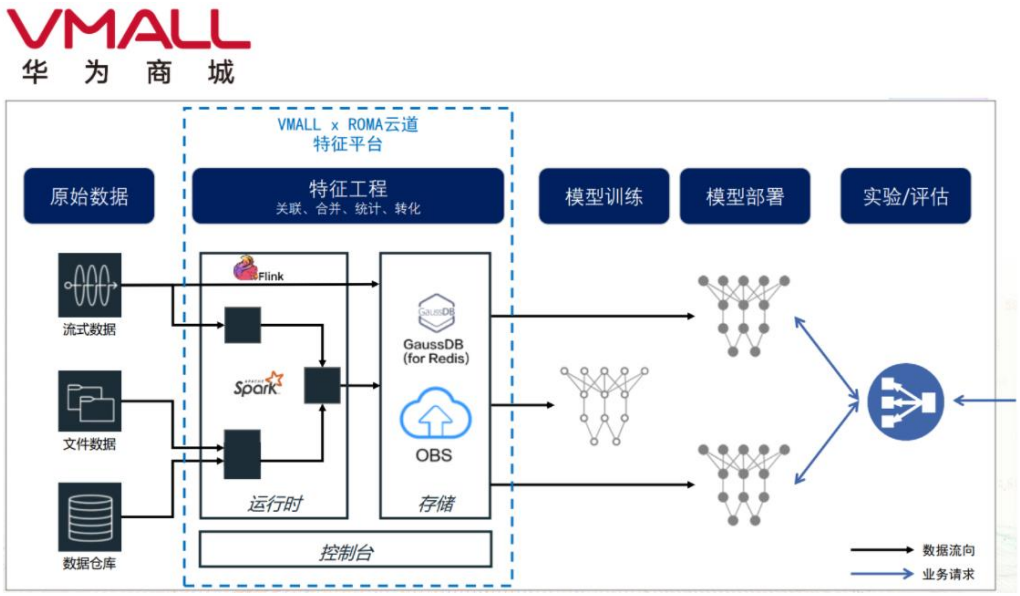
为解决此问题，VMALL 大数据团队开始着手建设统一的特征平台。



特征平台的核心部件是特征存储数据库，只有通过统一的特征数据存储，才能改变原有的“数据孤岛”窘境，彻底解决“不一致”、“难共享”、“效率低”3 大问题。

但也正是由于特征数据库需要承担打通线上/线下多个场景，对接批式/流式多种数据源，满足训练/推理多样消费需求，对特征数据库的选型提出了非常高的要求：需要找到一款数据存储服务，既能提供低成本的海量数据存储并方便扩容，又能保证数据的绝对可靠和服务的高可用；既要满足低时延的线上推理，又要满足高吞吐的线下训练；既能提供简洁的 KV 接口供下游轻松消费，又要兼容主流的批式/流式处理引擎（Spark/Flink 等）供上游快速接入。

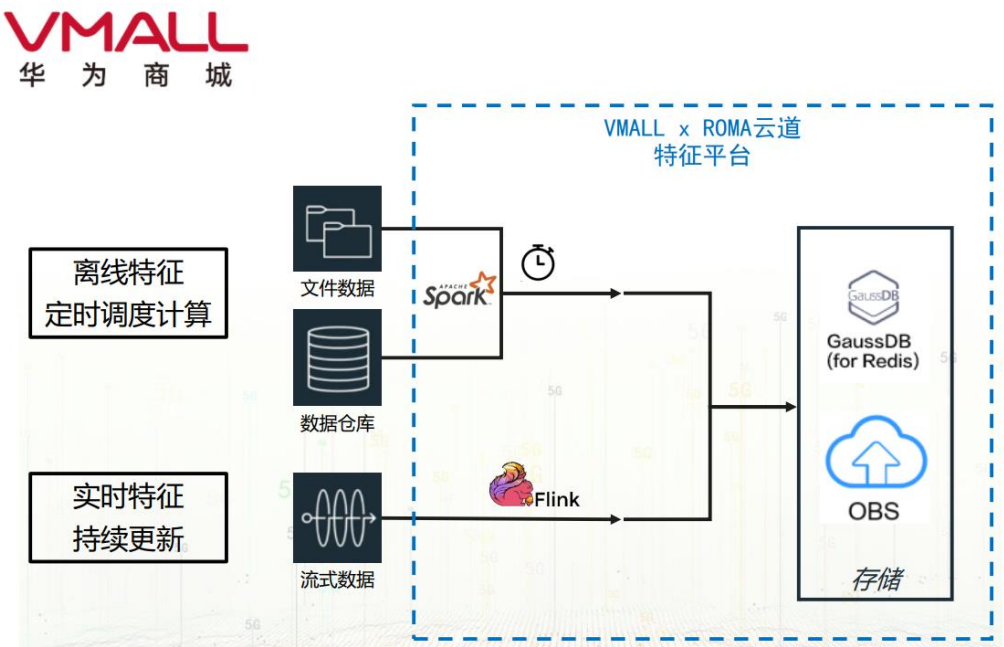
经过深入调研，VMALL 大数据团队最终选择了 GaussDB(for Redis)作为特征数据库，下面就让我们详细看看 GaussDB(for Redis)是如何满足上述苛刻要求的。



三、 解决方案

1.特征平台使用 GaussDB(for Redis)的主要流程

a) **特征生产（抽取、处理、存储）**



- 离线特征（静态特征）:

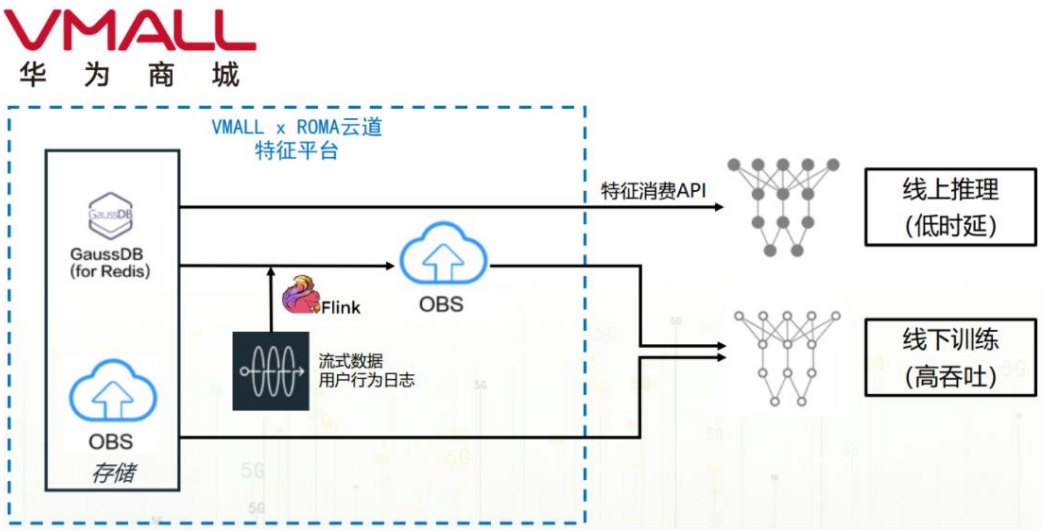
定时调度 Spark 作业，从各种数据仓库、数据湖中提取数据，进行特征工程处理后，存入

GaussDB(for Redis)。

- 实时特征（动态特征）:

Flink 消费 Kafka，或流式存储中的数据，持续更新到 GaussDB(for Redis)中。

b) **特征消费**



- 线上推理:

模型已经部署到生产，开始承接业务，需要低时延，高并发的消费数据，从GaussDB(for Redis)中读取数据。

- 线下训练:

GaussDB(for Redis)存有最新的特征数据，OBS 中存有全量的特征数据。

1) 对于使用静态特征的较为简单的模型，可以直接从 OBS 中获取特征使用。
2) 对于使用实时特征的场景（如实时推荐系统），由 Flink 从 Kafka 中实时取得用户请求记录，并从 GaussDB(for Redis) 查询取得特征，将记录和特征拼接成训练样本，存储到文件中，供线下训练使用。

2.特征平台对 GaussDB(for Redis)的核心诉求

结合上述业务场景，总结特征平台对 GaussDB(for Redis)的核心诉求如下:

序号	类别	诉求点
1	业务接口	支持简洁的 KV 接口（不需要范围查询），支持和 Spark/Flink 快速对接，提升业务开发效率
2	稳定性	作为电商应用的关键支撑系统，需要具备企业级应用的稳定性。如社区 Redis 存在的 fork 抖动，oom 等问题，需要解决
3	可靠性	特征数据决定了最终给客户的推荐效果，需要确保数据零丢失，强一致
4	成本	特征数据体量庞大，希望采用内存+磁盘混合存储，降低使用成本
5	性能	抗写能力必须强，满足特征数据批量灌库的要求；读取总体要低时延，但非缓存场景，可接受非活跃用户的时延毛刺
6	可扩展性	随着业务发展，特征数据上量很快；扩容要简单，快速，业务影响尽量小

3.GaussDB(for Redis)满足特征平台诉求的关键方案

a） **业务接口**

GaussDB(for Redis)兼容社区 Redis5.0 接口，支持和 Spark/Flink Connector 的对接，很好的满足了业务的使用需求

b） **稳定性**

GaussDB(for Redis)采用自研内核，解决了社区 Redis 的 fork，oom 等老大难问题，具备了企业级应用的稳定性

c） 可靠性

数据零丢失：逐条命令实时落盘，底层三副本冗余存储，无数据丢失风险

数据强一致：基于 GaussDB 公共的共享存储部件 DFV，实现三副本强一致，多点访问无脏读风险

d） **成本**

GaussDB(for Redis)实现数据的自动冷热分离，采用内存+SSD 的混合存储方案，大幅降低了客户的使用成本。按照 VMALL 的特征体量测算，亿级用户，每个用户的特征数量是数 K-数 10K，GaussDB(for Redis)一年的费用仅 3W 出头，如果选用社区 Redis，费用在 20W+

e） **性能**

GaussDB(for Redis)采用多线程架构，并且所有节点可以同时支持写入，因此可以较好满足批量灌库的高吞吐写需求。读方面，基于冷热分离方案，热数据常驻内存提供稳定低时延；冷数据读涉及 IO 交换，存在一定长尾，但可满足 VMALL 业务要求（目前 VMALL 线上 GaussDB(for Redis)实例读时延平均 0.16ms，P99 0.4ms，P9999 1.5ms）

f) **可扩展性**

基于计算存储分离架构，底层数据可被任一节点访问，扩容过程不发生数据拷贝搬迁，因此速度极快；计算节点扩容分钟级完成，存储扩容秒级完成，RTO < 10 秒

综上，与社区 Redis 相比，GaussDB(for Redis)提供了更稳定的使用体验，更可靠的数据存储，更低廉的使用成本和更便捷的扩展能力，是更适合像 VMALL 特征平台这样大规模电商大数据应用的企业级 Redis 服务。因此，VMALL 特征平台最终选择 GaussDB(for Redis)作为特征数据的存储服务。

四、 上线后效果

目前 VMALL 已完成一期的特征数据迁移，包括“特征生产”业务中的“Spark 离线特征生产”，以及“特征消费”业务中的“线下训练 Flink 特征查询”，已迁移到 GaussDB(for Redis)。

当前 GaussDB(for Redis)运行平稳，业务高峰时段时延稳定，能够满足 VMALL 当前业务要求。其中，读平均时延 0.2ms（p99<0.4ms），写入平均时延 0.6ms（P99<2ms）。

VMALL 当前已启动二期的特征数据迁移，计划完成包括“Flink 在线特征生成”，“线上推理”等核心业务的接入。

五、 总结

本文介绍了华为商城（VMALL）在建设特征平台过程中，对特征数据存储服务的选型和应用。由此可见，华为云 GaussDB(for Redis)服务在成本，可靠性，可扩展性等方面具有优势，可作为特征数据存储的理想方案，提供企业级的稳定可靠的 Redis 服务能力。

GaussDB(for Redis)免费体验：企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，**点击体验**

【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专

业技能！**点击前往**

高斯 Redis Geo 为地理位置信息存储场景提供更优解决方案

一、背景

LBS（Location Based Service，基于位置的服务）有非常广泛的应用场景，最常见的应用就是 POI（Point of Interest）的查询，例如用户查找附近的人，附近的餐厅，附近的外卖商家等等。LBS 的实现需要数据库存储地理位置信息，开源 Redis 是一个功能强、效率高、使用方便的缓存数据库，实现了地理位置存储的功能，可以用于 LBS 的数据存储。

开源 Redis 3.2 以上版本的 Geo 功能支持了地理位置信息存储管理，但是内存限制导致没有大规模应用。GaussDB(for Redis)（下文简称**高斯 Redis**）兼容开源 Redis 的 Geo 功能，使用磁盘替代内存，突破了开源 Redis 的内存限制，可以完美解决 Geo 的大规模应用问题。

二、开源 Redis Geo 介绍

Redis 的 Geo 功能支持如下 6 个 Geo 的相关操作：

- geoadd：添加某个地理位置的坐标。

```
# geoadd 用于存储指定的地理空间位置，可以将一个或多个经度(longitude)、纬度(latitude)、
# 位置名称(member)添加到指定的 key 中。
# geoadd 语法格式如下：
# GEOADD key longitude latitude member [longitude latitude member ...]
# 添加member(ShengZhen)的地理位置坐标(经度120,纬度30)到key (GuangDong)中
127.0.0.1:6318> geoadd GuangDong 120 30 ShengZhen
(integer) 1
```

华为云社区

- geopos：获取某个地理位置的坐标。

```
# geopos 用于从给定的 key 里返回所有指定名称(member)的位置（经度和纬度）
# geopos 语法格式如下
# GEOPOS key member [member ...]
# 获取key(GuangDong)中member(ShengZhen)和member(DongGuang)的经纬度
127.0.0.1:6318> geopos GuangDong ShengZhen DongGuang
1) 1) "120.00000089406967"
   2) "30.000000249977013"
2) 1) "100.00000208616257"
   2) "24.799999513823884"
```

华为云社区

- geodist：获取两个地理位置的距离。

```
# geodist 用于返回两个给定位置之间的距离
# geodist 语法格式如下：
# GEODIST key member1 member2 [m|km|ft|mi]
# member1 member2 为两个地理位置。
# 最后一个距离单位参数说明：
# m ：米，默认单位；km ：千米； mi ：英里；ft ：英尺。
# 获取key (GuangDong)中member(ShengZhen)与member(DongGuang)的距离。
127.0.0.1:6318> geodist GuangDong ShengZhen DongGuang km
"2054.6799"
```

华为云社区

- geohash：获取某个地理位置的 geohash 值。

```
# geohash 用于获取一个或多个位置元素的 geohash 值
# geohash 语法格式如下：
# GEOHASH key member [member ...]
# 获取key(GuangDong)中member(ShengZhen)和member(DongGuang)的geohash值
127.0.0.1:6318> geohash GuangDong ShengZhen DongGuang
1) "wtm6dtm6dt0"
2) "whrj5d9y2y0"
```

华为云社区

- georadius：根据给定地理位置坐标获取指定范围内的地理位置集合。

```
# georadius 以给定的经纬度为中心，返回键包含的位置元素当中，与中心的距离不超过给定最大距离的所有位置元素
# georadius 语法格式如下：
# GEORADIUS key longitude latitude radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH]
[COUNT count] # [ASC|DESC] [STORE key] [STOREDIST key]
# 命令参数说明：
# m ：米，默认单位；km ：千米； mi ：英里；ft ：英尺。
# WITHDIST：在返回位置元素的同时， 将位置元素离中心节点位置的距离也一并返回。
# WITHCOORD：将位置元素的经度和维度也一并返回。
# WITHHASH：以 52 位有符号整数的形式，返回位置元素经过原始 geohash 编码的有序集合分值，这个选项主要用
于底层应用或者调试。
# COUNT： 指定返回位置元素的数量。
# ASC：返回位置元素按照离中心节点的距离做升序排列。
# DESC：返回位置元素按照离中心节点的距离做降序排列。
# STORE：将返回位置元素的地理位置信息保存到指定key。
# STOREDIST：将返回位置元素离中心节点的距离保存到指定key。
# 获取key (GuangDong)中距离地理位置坐标(经度120,纬度30)为2000km的成员
127.0.0.1:6318> georadius GuangDong 120 30 2000 km
1) "GuangZhong"
2) "ShengZhen"
```

华为云社区

- georadiusbymember：根据给定地理位置获取指定范围内的地理位置集合。

```
# georadiusbymember 和 georadius 命令一样，都可以找出位于指定范围内的元素，但是 georadiusbymember
的中心点是由给定的位置元素决定的，而不是使用经度和纬度来决定中心点
# georadiusbymember 语法格式如下:
# GEORADIUSBYMEMBER key member radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT
count] [ASC|DESC] [STORE key] [STOREDIST key]
# Georadiusbymember与georadius命令参数含义相同
127.0.0.1:6318> georadius GuangDong ShengZhen 2000 km
1) "GuangZhong"
2) "ShengZhen"
                                                                    华为云社区
```

Redis Geo 功能的空间索引采用 GeoHash 原理，配合 zset 集合存储，查询效率接近 log(N)。

三、为什么开源 Redis Geo 没有广泛应用?

存储地理位置信息的应用非常广泛，而开源 Redis Geo 功能也可以存储地理位置信息，并且查询效率高，为什么没有得到大规模的应用呢?

分析存储地理位置信息的场景，都有如下特点:

- 数据量大

  大部分场景存储地理位置信息的数据量都是 TB 级以上的，开源 Redis 的数据全部存放在内存中，节点的内存大小固定，要支持大数据量的地理位置信息存储，必须增加节点数，这会造成成本过高、大集群维护困难等问题。

- 数据持续增长

  随着用户的增长，地理位置信息的数据也在持续增长，要求底层存储能够无损扩容。但开源 Redis 扩容需要重新划分 hash 槽进行数据迁移，必定会影响业务。

- 高并发读写

  开源 Redis 主从模式下只有主节点可写，主节点高并发数据写入、高并发数据读出，写入速度过高容易造成主从堆积，数据丢失。

  除此之外，还需要考虑备份恢复，数据一致性，扩容，高可用等数据库系统能力。

- 备份恢复

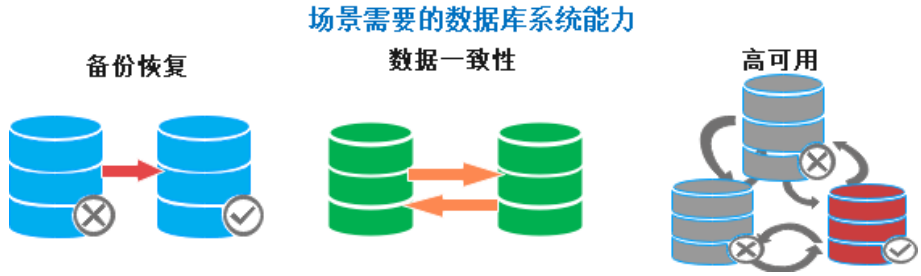
  开源 Redis 提供 RDB 和 AOF 方式备份数据，但当数据规模大时，RDB 方式恢复的数据一致性和完整性较差，AOF 方式数据恢复的效率低。

- 数据一致性

  开源 Redis 的主从采用异步复制，会出现数据不一致的情况。

- 高可用

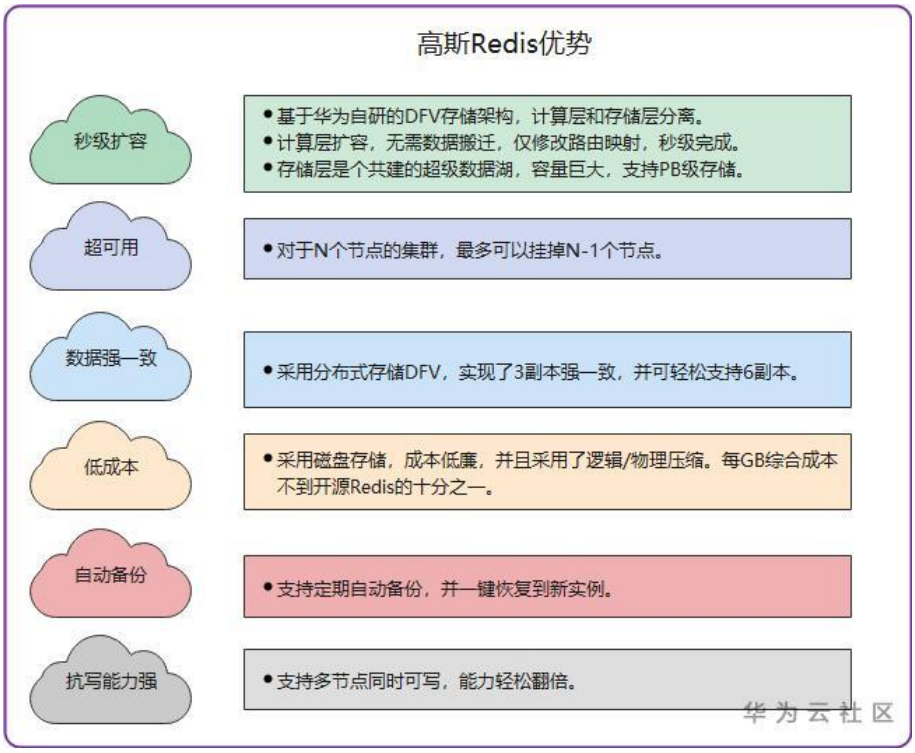
  开源 Redis 如果同时挂掉一对主从节点，部分数据将不可用，容错能力弱。



四、高斯 Redis 为什么合适?

高斯 Redis 基于华为自研分布式存储系统 DFV，支持 PB 级大规模的数据存储。解决了开源 Redis 高成本、存储数据量小、数据不一致等问题，具有秒扩容、超可用、强一致、低成本、自动备份、抗写能力强的优势。

GaussDB(for Redis)免费体验：企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，[点击体验]



五、适用场景

高斯 Redis Geo 功能适用于数据量大、读写频繁的场景。在外卖平台、点评平台、找房平台中，餐馆的数据、外卖骑手的数据、用户的数据、房源的数据这些数据随着用户增长，数据量过亿，对应的地理位置信息的数据量可到数 TB 级别，正是高斯 Redis 适用的场景。下面介绍在不同场景中 Geo 功能的应用。

1. 外卖场景：

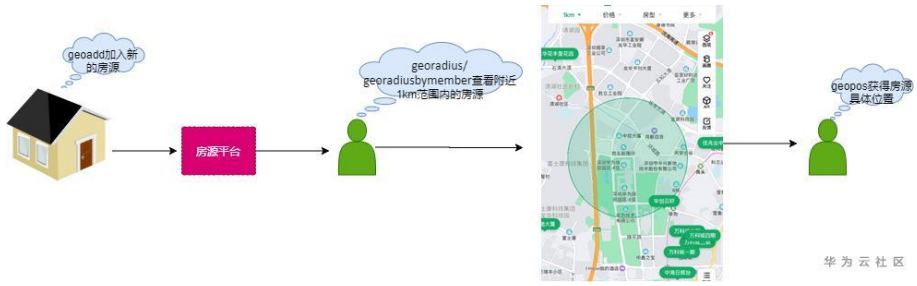
1) 用户下完外卖订单后，使用 geoadd 命令加入骑手的位置。
2) 使用 geopos 命令，用户可获得骑手的具体位置。
3) 使用 georadius/ georadiusbymember 命令骑手查看附近可配送的订单。
4) 使用 geodist 命令用户可获得骑手的距离。

2.点评场景：

1) 新的店铺加入点评平台，使用 geoadd 命令，添加新店铺的位置。
2) 使用 geopos 命令，用户获得店铺的具体位置。
3) 使用 geodist 命令，用户可获得与店铺的距离。
4) 使用 georadius/ georadiusbymember，用户可查找距离 500 米范围的店铺。

3. 找房场景：

1) 新的房源加入房源平台中，使用 geoadd 命令，添加新房源的位置。
2) 使用 geopos 命令，用户可获得房源的具体位置。
3) 使用 geodist 命令，用户可获得与房源的距离。
4) 使用 georadius/ georadiusbymember 命令，用户查找附近 1km 范围内的房源。



六、总结

开源 Redis 的 Geo 功能查询效率高，但存在存储容量小、抗写能力弱、可用性差等明显缺点，导致了其 Geo 功能一直没有广泛应用。高斯 Redis 突破了开源 Redis 的内存限制，以高性能磁盘存储数据，具有秒扩容、超可用、强一致、低成本、自动备份、抗写能力强的特点，因此高斯 Redis 适用于大量地理位置信息存储的场景。

【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！[点击前往]

IM 场景的技术挑战下，GaussDB(for Redis) 的创新应用有哪些?

一、背景

即时通讯（Instant Messaging，简称 IM）是一个实时通信系统，允许两人或多人使用网络实时的传递文字消息、文件、语音与视频。微信、QQ 等 IM 类产品在这个高度信息化的互联网时代已成为生活必备品，IM 系统中最核心的部分是消息系统，消息系统中最核心的功能是消息的同步、存储和检索。

- **消息同步：**

将消息完整的、快速的从发送方发送至接收方。消息同步系统最重要的衡量指标是消息传递的实时性、完整性、顺序性以及支撑的消息规模。

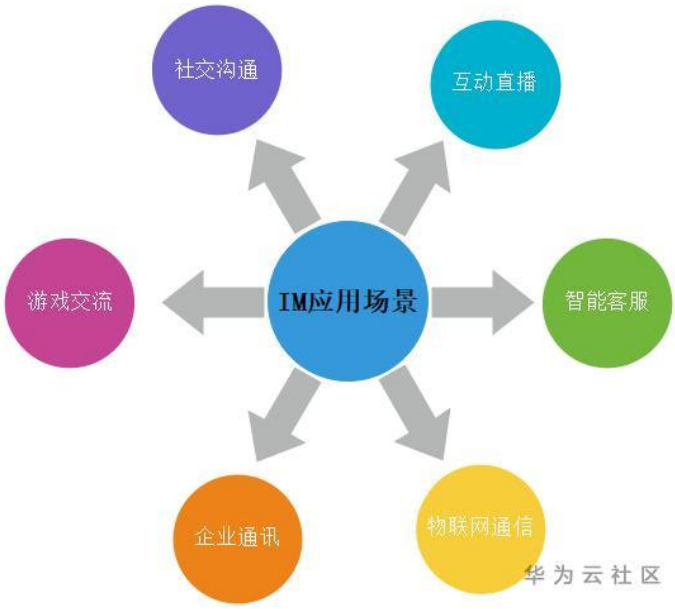
- **消息存储：**

即消息的持久化，传统消息系统通常支持消息在接收端的本地存储，数据基本不具备可靠性。现代消息系统支持消息在云端存储，从而实现消息漫游：账号可在任意客户端登陆查看所有历史消息。

- **消息检索：**

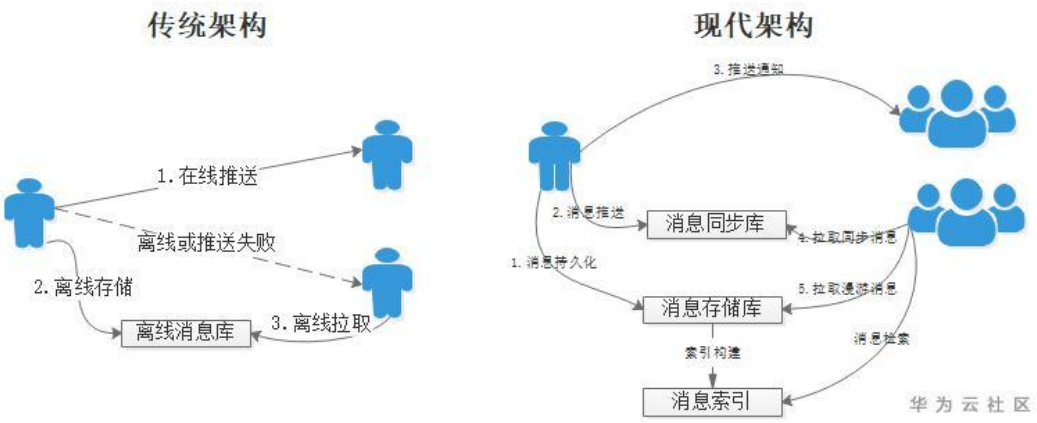
消息一般是文本，所以支持全文检索也是必备的能力之一。传统消息系统通常来说基于本地存储的消息数据来构建索引，支持消息的本地检索。而现代消息系统支持消息的在线存储以及存储过程中构建索引，提供全面的消息检索功能。

二、IM 系统架构设计



上图为 IM 系统的应用场景，可用于聊天，游戏、智能客服等诸多行业。不同行业对 IM 系统的成本、性能、可靠性、时延等指标的需求是不同的，架构设计需要进行平衡。接下来将介绍 IM 系统架构设计所涉及到的一些基本概念。

1. 传统架构 vs 现代架构



传统架构

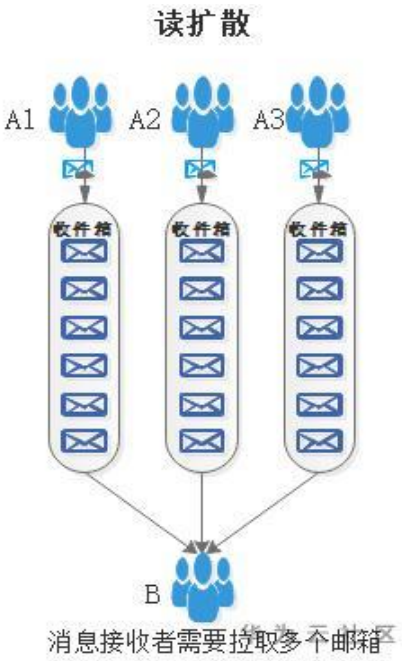
- 先同步后存储。
- 在线消息同步和离线消息缓存。
- 服务端不会对消息进行持久化，无法支持消息漫游。

现代架构

- 先存储后同步。
- 划分消息存储库与消息同步库。消息存储库用于全量保存所有会话消息，主要用于支持消息漫游。消息同步库，主要用于接收方的多端同步。
- 提供消息全文检索能力。

2. 读扩散 vs 写扩散

《2020 微信数据报告》指出，截至 2020 年 9 月，微信月活跃用户数为 10.825 亿，日消息发送次数 450 亿次，日音视频呼叫成功次数 4.1 亿次。面临这么多的消息，如何保证消息传递的可靠性、一致性并且有效的降低服务器或者客户端的压力是十分具有技术挑战的。其中，采用何种读写模型对 IM 系统至关重要，这里介绍两种模型：读扩散和写扩散。



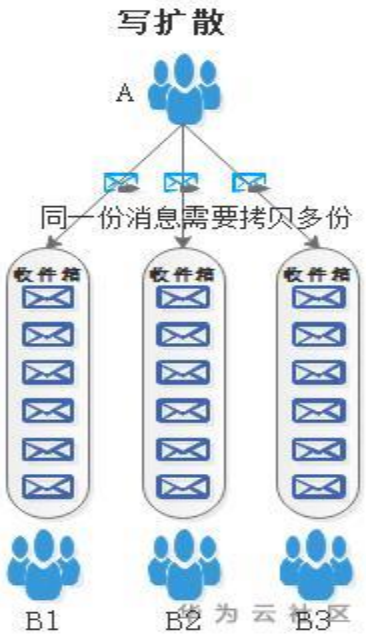
如上图所示，用户 B 与每个聊天的人(A1,A2,A3)都有一个信箱（一种数据结构的抽象，用于存储消息），B 在查看聊天信息时需读取所有有新消息的信箱。IM 系统里的读扩散通常是每两个相关联的人就有一个信箱。

读扩散的优点：

- 写操作(发消息)轻量，不管是单聊还是群聊，只需要往相应的信箱写一次即可；
- 每一个信箱天然就是两个人的聊天记录，可以方便查看和搜索聊天记录。

读扩散的缺点：

- 读操作(读消息)很重，存在读放大效应。



如上图，在写扩散中，用户(B1,B2,B3)都只从自己的信箱里读取消息，但写(发消息)的时候，对于单聊跟群聊处理如下：

- 单聊：

  往自己的信箱跟对方的信箱都写一份消息；同时，如果需要查看两个人的聊天历史记录的话还需要再写一份。

- 群聊：

  发信息时需要针对所有群成员的信箱都写一份消息。群聊使用的是写扩散模型，而写扩散很消耗资源，因此微信群有人数上限(目前是 500)。

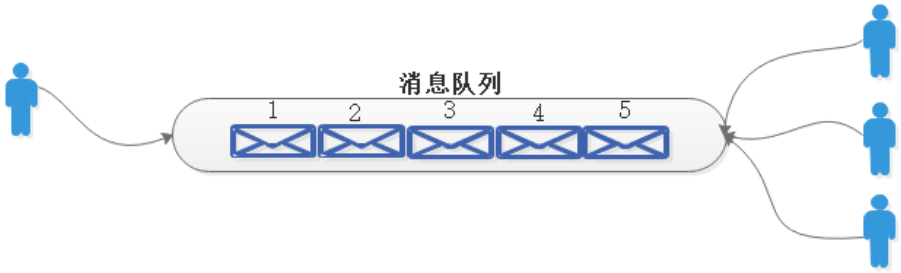
写扩散优点：

- 读操作很轻量，只需要读取自己的邮箱。
- 可以很方便实现消息的多终端同步。

写扩散缺点：

- 写操作很重，尤其是对于群聊来说。

3. 推模式 vs 拉模式 vs 推拉结合模式



在 IM 系统中，消息的获取通常有三种模式：

- **推模式(Push)：**

新消息到达时由服务器主动推送给所有客户端；需要客户端和服务器建立长连接，实时性很高，对客户端来说只需要接收处理消息即可；缺点是服务端不知道客户端处理消息的能力，可能会导致数据积压。

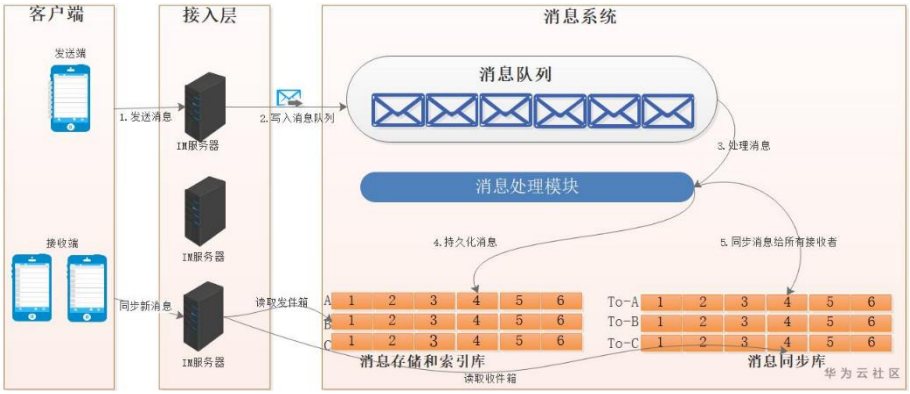
- **拉模式(Pull)：**

由前端主动发起拉取消息的请求，为了保证消息的实时性，一般采用推模式，拉模式一般用于获取历史消息；因客户端拉取新消息的时间间隔不好预设，太短可能会导致大量的连接拉取不到数据，太长导致数据接收不及时。

- **推拉结合模式：**

兼顾 push 和 pull 两种模式的优点。新消息来临时服务器会先推送一个新消息到达的通知给前端，前端接收到通知后就向服务器拉取消息。

三、IM 技术挑战



上图为 IM 系统的总体架构图，Client 双方通信会经过 Server 转发来完成消息传递。其核心为消息存储库和消息同步库。这两种库对存储层的性能有极高的要求。

- **支撑海量数据存储：**

对于消息存储库来说，如果需要消息永久存储，则随着时间的积累，数据规模会越来越大，需存储库支持容量无限扩展以应对日益增长的消息数据。

- **低存储成本：**

消息数据具有明显的冷热特征，大部分查询集中在热数据，冷数据需要一个低成本的存储方式，否则随着时间的积累，数据量不断膨胀，存储成本会不断上升。

- **数据生命周期管理：**

不管是对于消息数据的存储还是同步，数据都需要定义生命周期。存储库是用于在线存储消息数据本身，通常需要设定一个较长周期的保存时间。而同步库是用于写扩散模式的在线或离线推送，通常设定一个较短的保存时间。

- **极高的写入吞吐：**

绝大部分 IM 类场景，通常是采用写扩散模型，写扩散要求底层存储具备极高的写入吞吐能力，从而应对消息洪峰。

- **低延迟的读：**

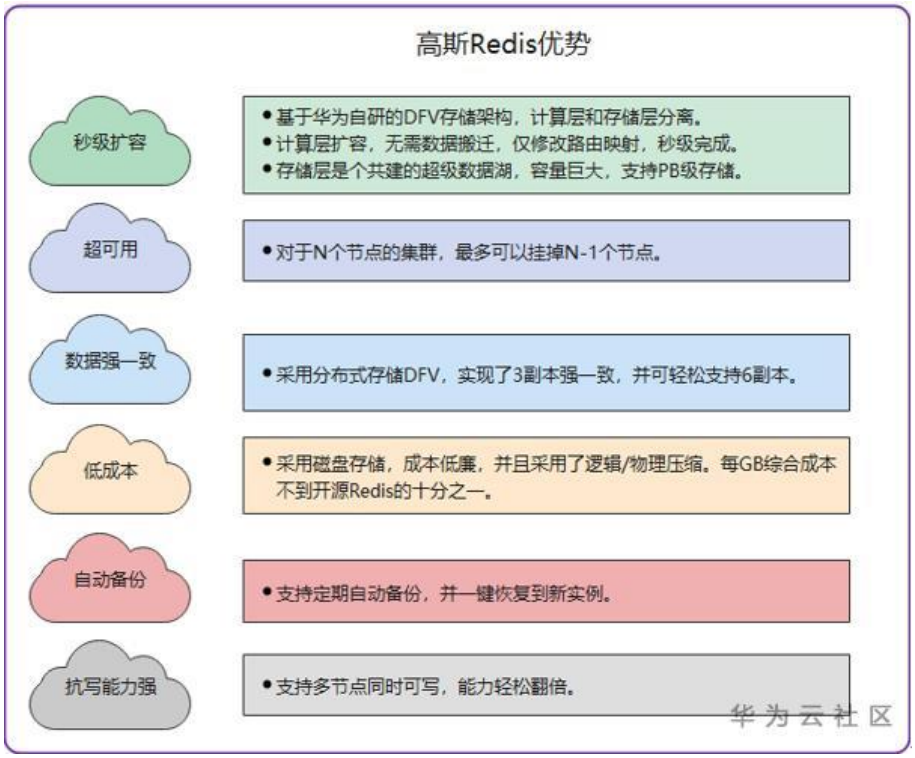
消息系统通常应用于在线场景，具备较高的实时性，读取延迟应尽可能低。

四、高斯 Redis 在 IM 场景中的优势

IM 系统的核心是存储层，其性能差异将直接影响 IM 系统的用户体验。目前存储层可选择的数据库产品有很多，如 HBase、开源 Redis 等等。选择何种数据库，需根据业务规模、成本、性能等指标来进行综合选择。这里介绍一种 NoSQL 数据库：高斯 Redis，在性能和规模上，可以满足 IM 系统对存储层的严格要求：海量数据存储、低存储成本、生命周期管理、写入吞吐大、读取时延低。

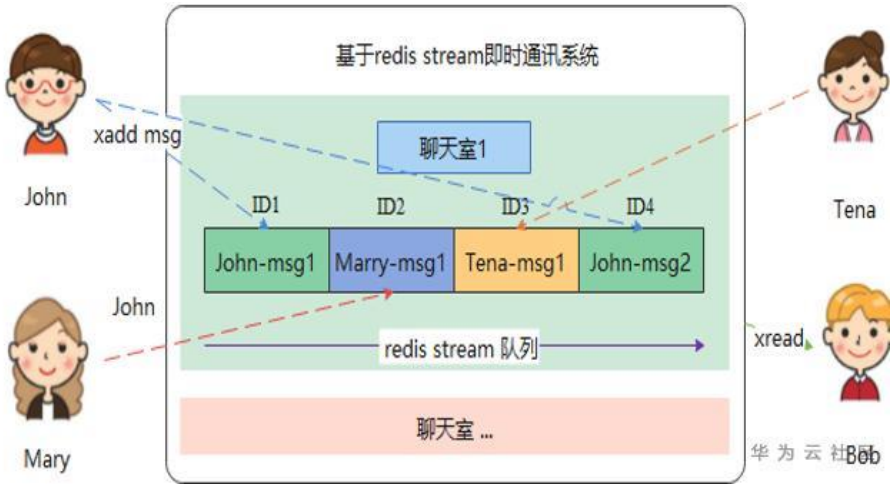
1. 高斯 Redis 简介

高斯 Redis 是华为云数据库团队自主研发且兼容 Redis5.0 协议的云原生数据库，采用计算存储分离架构。存储侧使用自研的存储系统，容量无限扩展、强一致、高可靠。计算侧基于 LSM 存储引擎实现，通过将大量的随机写转换为顺序写，从而极大的提升了数据写入性能，同时也通过读缓存、bloom filter 等极大优化了读取性能。下图是高斯 Redis 在 IM 场景的优势介绍。



2. 基于高斯 Redis 的 IM 应用案例：

下图是基于高斯 Redis 的 IM 系统模型图，这里我们使用 stream 作为基本数据结构。Redis stream 不仅可以作为消息存储容器，还实现了生产者、消费者等基本模型，具有 IM 系统的基本功能，如消息订阅，分发、增加消费者等，用户可基于高斯 Redis 快速构建一套 IM 系统。创建一个群聊时，在 Redis 中对应地为该群聊创建一个 stream 队列。在发送消息时，每个用户都将消息按照时间顺序添加到 stream 队列中，保证了消息的有序性。stream 是一个持久化的队列，可保证信息不丢失。



GaussDB(for Redis)免费体验： 企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，[点击体验]

五、总结

GaussDB(for Redis)通过一系列技术创新实现了读写性能水平扩展，秒级扩容，低成本以及自动备份等功能, 可作为 IM 系统的存储层，其优异的读写性能和高级特性将会极大助力 IM 应用.同时，高斯 Redis 在开源 Redis 的基础之上，较好平衡了性能和成本，能够广泛应用在智慧医疗、流量削峰、计数器等领域。

【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！[点击前往]

朋友圈、抖音等让人沉沦的 feed 流，GaussDB(for Redis)轻松给你设计

一、背景

GaussDB(for Redis)（下文简称高斯 Redis），是华为自研的强一致、持久化 NoSQL 数据库，兼容 Redis5.0 协议。

在互联网时代，我们日常生活充斥着 Feed 流，微信朋友圈、微博、抖音以及头条等等都在使用 Feed 流，将我们关注的好友或感兴趣的内容及时推送给我们，使我们沉沦其中无法自拔，带来商业价值的提升。接下来将和大家一起探讨常见的 Feed 流系统包含的概念、架构和挑战以及如何使用高斯 Redis 设计一个 Feed 流系统。

二、 概念

Feed 流系统从形式上是 Feed 生成者将生产的 Feed 经过存储分发系统传递给 Feed 消费者，最终以某种展现形式。

1. Feed：单次推送的内容。 一条微博就是一个 Feed。

2. Feed 流：由连续的 Feed 组成的信息流。

3. 展现形式：展示形式目前主要有 Timeline 和 Rank。比如微博以 Timeline 展示。头条客户端主要以推荐 Rank 来进行展示。

4. Feed 生产者：对于微博就是每个用户，针对头条就是推荐算法。

5. Feed 消费者：接收 Feed 通知的主体。

6. 同步存储系统：这部分一般又可以分为三部分(具体实现会略有差异)。

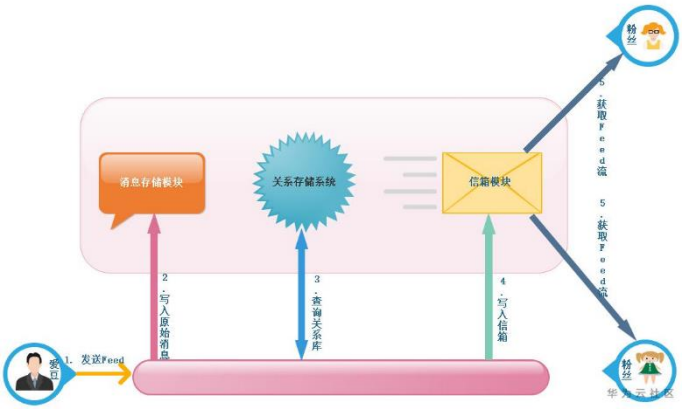
    6.1 内容存储模块：这部分关心 Feed 原始内容如何存储。如你发的一条微博。

    6.2 关联关系存储模块：对于微博存储的是用户的关注人和被关注人，对于头条存储的是人群(根据用户画像对所有用户进行分类

    6.3 信箱模块：一般又可叫做消息传递模块，Feed 消息存储到信箱，才能最终形成 Feed 流。

三、架构设计

通过上述的概念介绍，我们来看 Feed 如何从 Feed 生产者最终流转到 Feed 消费者。



Feed 生产者创作一条内容后发到 Server 端，Server 端先将消息内容存储到消息存储模块，接着根据信箱模块的设计查询关系存储库将通知信息写入信箱模块， Feed 消费者通过查询信箱获取及时消息。

消息存储模块：

Feed 内容一般都是半结构化数据，数据量大，需要持久化内容，逻辑上就是一个 KV 系统，ID 到内容的映射关系。

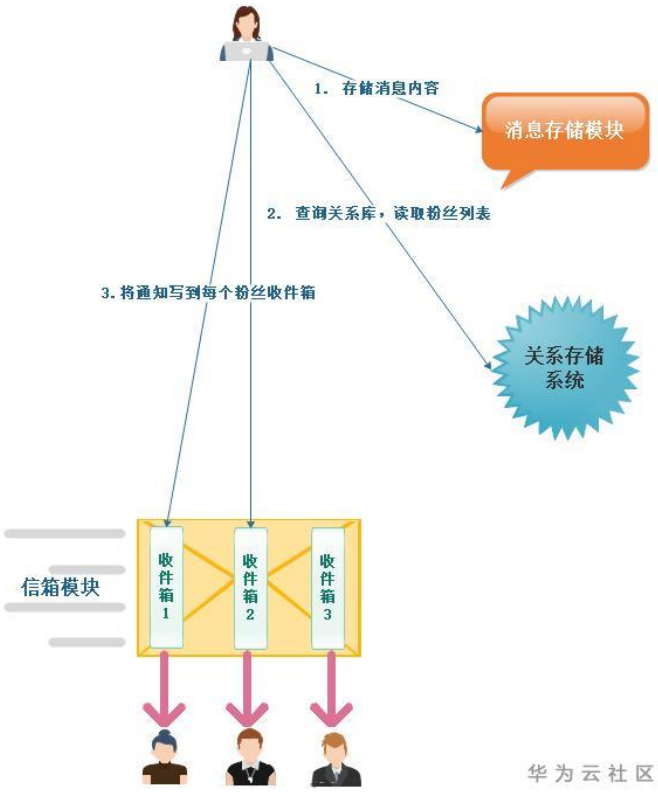
关系存储模块：

关联关系会发生新增和删除，是一个变长的集合，需要能够支持快速的增删查动作，一般不需要支持 join 等复杂操作。因此 NoSQL 数据库比较适合这类数据的存储。

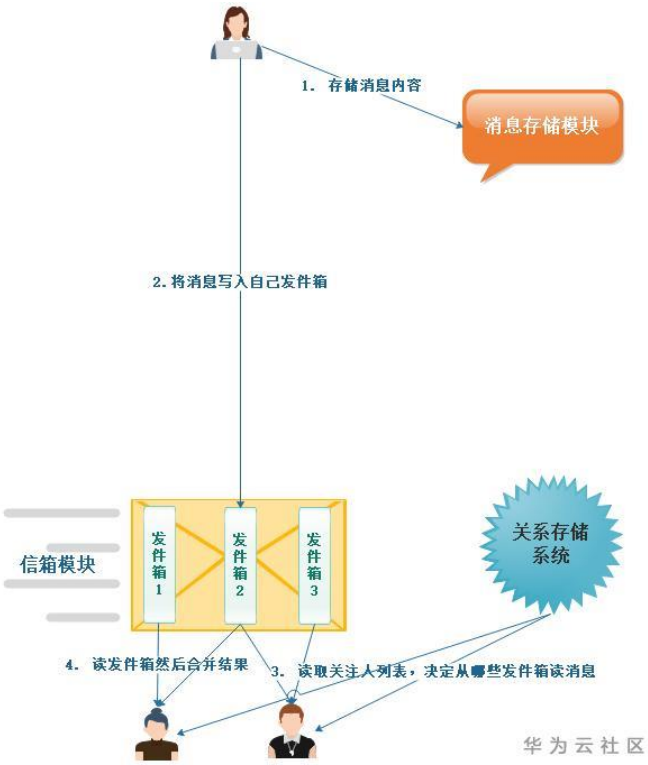
信箱模块：

说到信箱模块一般大家都会讨论是采用推模式、拉模式或推拉模式结合方式。

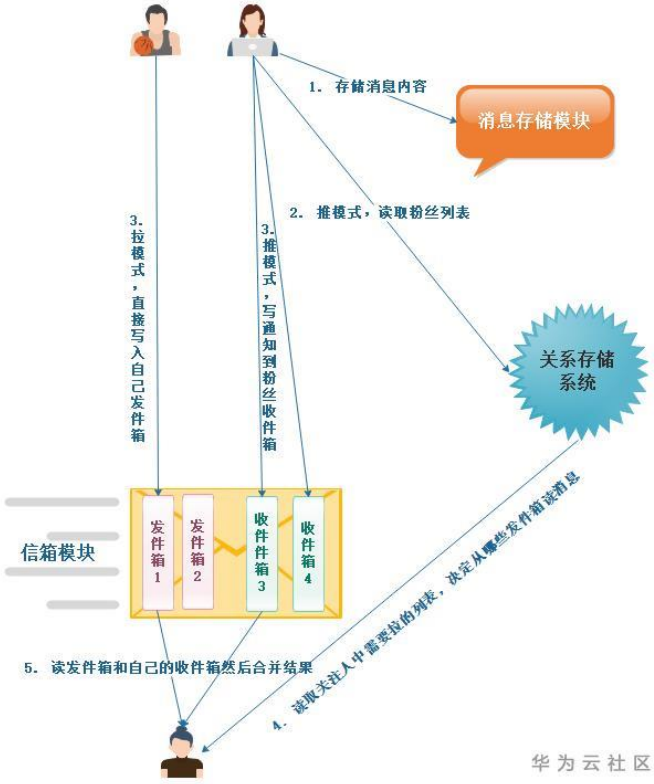
推模式，在查询完关联关系后，将 Feed 通知写入到每个 Feed 消费者的收件箱中，Feed 消费者查询自己的收件箱就能获取完整的 Feed 流，通知会写入每个需要通知的人收件箱，写会放大。



拉模式，将 Feed 通知写入自己的发件箱，Feed 消费者先查询关系库，然后从所有关注人的发件箱中获取 Feed 消息后合并展示，因此收件箱被读次数和被关注人数目有关，读会放大。



推拉模式结合：针对大多数用户的写入用推模式，特定用户采用拉模式，Feed 消费者读取时分别读取自己的收件箱和特定用户的发件箱，合并之后展示。



究竟选用哪种模式，看具体业务场景和要求。

很多业务在具体实现的时候，会先将消息写入消息队列，一方面可以起到流量削峰的作用，另一方面可以实现一些特定的推送优化逻辑，如判断为垃圾广告或者敏感词不进行推送。

四、技术挑战

我们首先从微信朋友圈公布的数据来感受一下。在 2021 年 1 月 19 日，在微信公开课 Pro 上微信创始人张小龙披露微信最新数据：微信每天有 7.8 亿用户进入朋友圈，1.2 亿用户发表朋友圈。平均每人打开浏览十几次，每天 100 亿次浏览量。

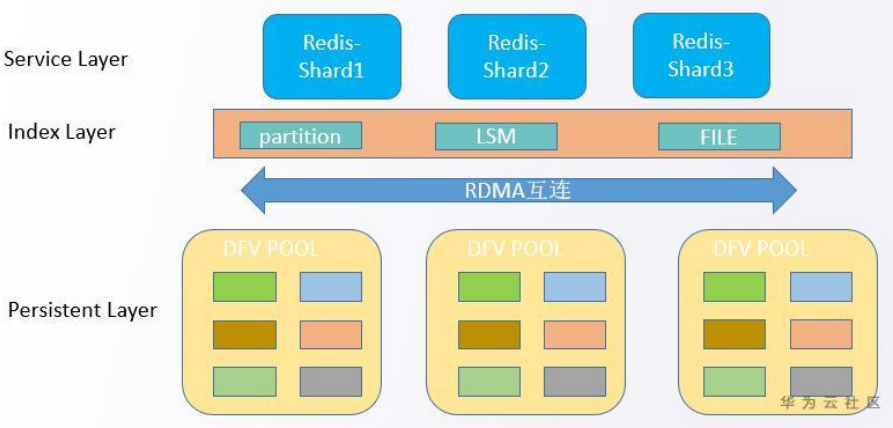
若我们想实现类似的 Feed 流系统会有什么挑战。从存储量上来看，若用户平均每天发送 3 次朋友圈，每条内容 1kB，一年大约 1000 亿条记录，存储容量接近 100TB。

从访问请求次数来看，每天写入和读取 OPS 峰值至少百万级别，用户写入和读取延迟都要有实时性，响应时间至少都要在秒级内，否则用户分分钟关闭 APP。

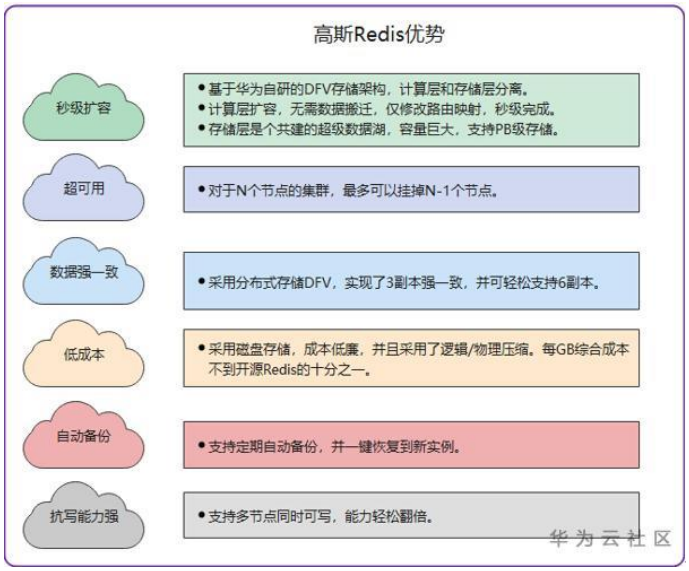
因此我们需要一个持久化、海量存储、高吞吐、易扩展、低延迟、低存储成本的分布式存储系统。

五、 高斯 Redis 的优势

1. 高斯 Redis 简介



高斯 Redis 是华为云数据库团队自主研发且兼容 Redis5.0 协议的云原生数据库，采用计算存储分离架构。存储侧使用自研的存储系统 DFV，容量无限扩展、强一致、高可靠。计算侧基于 LSM 存储引擎实现，具有极佳的写性能和读性能。利用计算分离架构的优势，高斯 Redis 扩容无需进行数据拷贝，实现秒级扩容，充分发挥了云原生的弹性伸缩、资源共享的优势。



2. Feed 流场景如何利用高斯 Redis 优势

针对 Feed 流场景，可以按照如下方式使用高斯 Redis：

a）消息内容存储

利用高斯 Redis 的 KV 结构即可实现，高斯 Redis 采用存储计算分离架构，可以轻松支撑海量数据存储及低延迟访问延迟。

b）关联关系存储

高斯 Redis 集合结构或字典结构可以轻松实现关联关系的增删改查。

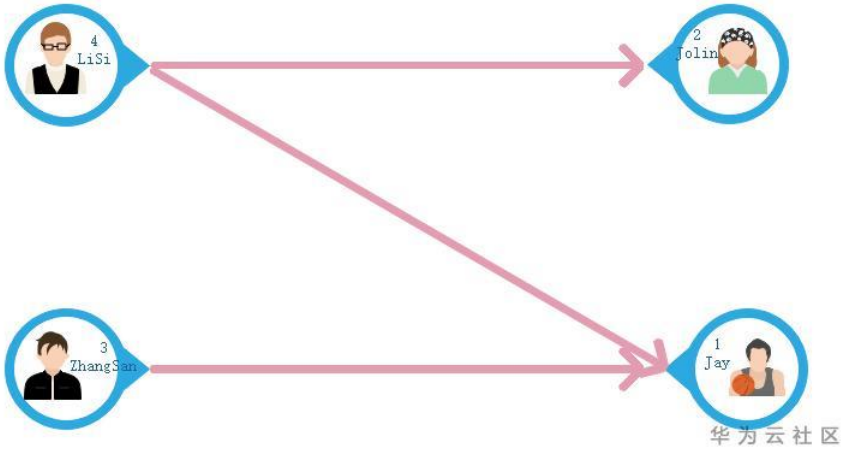
c）信箱存储

信箱从实现上就是一个队列，支持从指定位置消费的能力。高斯 Redis 的 Stream 结构可以实现队列能力，轻松实现 Feed 流消息读取。

3. 高斯 Redis Feed 流实践

以下用高斯 Redis 实现一个简单的微博样例，采用写扩散模型，用以说明可行性。

系统中存在四个用户，Jay、Jolin、ZhangSan、LiSi。其中 ZhangSan、LiSi 关注了 Jay，LiSi 同时关注了 Jolin。



华为云社区

```
# 系统中有四个用户分别是：Jay、Jolin、ZhangSan、LiSi
# Jay的id是1，Jolin id是2，ZhangSan id是3，LiSi id是4
# 我们以userid_{id}作为key存储用户的信息

127.0.0.1:8635> HSET userid_1 name Jay
1
127.0.0.1:8635> HSET userid_2 name Jolin
1
127.0.0.1:8635> HSET userid_3 name 'ZhangSan'
1
127.0.0.1:8635> HSET userid_4 name 'LiSi'
1
# 此时ZhangSan和LiSi 关注了Jay
# 我们以follower_{id}作为key存储用户的粉丝id
127.0.0.1:8635> SADD follower_1 3 4
2
# 此时Lisi又关注了 Jolin
127.0.0.1:8635> SADD follower_2 4
1
```

华为云社区

```
# Jay此时发了一条微博，id是100，我们采用写扩散模型实现推送
# 首先按照message_{id}作为key存储内容
# 接着查询Jay的粉丝id列表，分别往粉丝3,4的收件箱推送消息
# 我们以inbox_{id}作为每个人的收件箱key
127.0.0.1:8635> SET message_100 '我将在鸟巢开演唱会，很迪奥哦'
OK
127.0.0.1:8635> SMEMBERS follower_1
3
4
127.0.0.1:8635> XADD inbox_3 * id message_100
1616243276152-0
127.0.0.1:8635> XADD inbox_4 * id message_100
1616243279902-0
127.0.0.1:8635>

# Jolin突然想在小巨蛋演出，此时她也发了一个微博
#
127.0.0.1:8635> set message_101 '我想在小巨蛋dance一下'
OK
127.0.0.1:8635> SMEMBERS follower_2
4
127.0.0.1:8635> XADD inbox_4 * id message_101
1616243936582-0
127.0.0.1:8635>
```

华为云社区

```
# 此时ZhangSan下班回到家拿起手机刷起微博
# 先查询自己的收件箱，看到有一条消息id message_100
127.0.0.1:8635> XRANGE inbox_3 - + count 10
1616243276152-0
id
message_100
# 接着具体查询内容是啥
127.0.0.1:8635> get message_100
我将在鸟巢开演唱会，很迪奥哦
```

华为云社区

```
# LiSi下课回到寝室也拿起手机刷起微博
# 查询自己的收件箱，发现有两条消息
127.0.0.1:8635> XRANGE inbox_4 - + COUNT 10
1616243279902-0
id
message_100
1616243936582-0
id
message_101
# 分别查询具体两条内容
127.0.0.1:8635> get message_100
我将在鸟巢开演唱会，很迪奥哦
127.0.0.1:8635> get message_101
我想在小巨蛋dance一下
127.0.0.1:8635>                                    华为云社区
```

以上实现了一个简单的微博系统，真实系统会比这个复杂，会涉及业务场景特定处理逻辑。用高斯 Redis 作为 Feed 流存储底座是比较理想的技术选型。

六、总结

GaussDB(for Redis)具有持久化、海量存储、高吞吐、易扩展、低延迟、低存储成本等优点，作为 Feed 流存储底座非常合适，其优异的读写性能和高级特性将会极大简化应用开发。同时，高斯 Redis 在开源 Redis 基础之上，较好平衡了性能和成本，能够广泛应用在智慧医疗、流量削峰、计数器等领域。

GaussDB(for Redis)免费体验：企业级 Redis 存储空间支持秒级扩容，业务 0 感知，8G 存储空间新用户免费试用 1 个月，点击体验

【数据库论坛】

数据库一站式学习平台 ，涵盖数据库理论基础、优质课程、案例实践。交流互助，提升专业技能！点击前往

更多精彩内容，扫码关注交流讨论

华为云开发者社区

华为云开发者社区主页

华为云开发者微信公众号

华为云数据库社区

华为云数据库社区

GaussDB数据库微信公众号

此合集内容均来自于华为云社区用户的原创博文。如需转载，请标注文章的来源（华为云社区）、文章链接、文章作者等基本信息。如涉及版权问题，请联系我们huaweicloud.bbs@huawei.com，经核实后会尽快予以处理。