# 背景

自 2016
年以来，为了支撑在线推荐的存储需求而诞生的——字节跳动自研高可用 KV
存储
Abase，逐步发展成支撑包
括推荐、广告、搜索、抖音、西瓜、飞书、游戏等公司内几乎所有业务线的
90% 以上的 KV 存储场景，已成为公司内使用最广泛的在线存储系统之一。

Abase 作为一款由字节跳动完全自研的高性能、大容量、高可用的 KV 存储系统，支撑了业务不断快速增长的需求。但随着公司的持续发展，业务数量、规模持续快速增长，我们业务对系统也提出了更高的要求，比如：

- 极致高可用：
  相对于一致
  性，信息流等业务对可
  用性要求更高，希望消除宕机
  选主造成的短时间不可用，和慢节点问题；
- 全球部署：无论是边缘机房还是不同地域的机房，同一个 Abase2
  集群的用户都可以就近访问，获取极快的响应延迟；
- CRDT
  支持：确保多写架构下的数据能自动解决冲突问题，达成最终一致；
- 更低成本：
  通过资源池化解决不同用户资源使用不均衡，造成资源利用率不足问题
  ，降低成本；
- 极致高性能：
  相同的资源使用下，要求提供尽可能高的写/读吞吐，和较低的访问延
  迟。适配 IO 设备和 CPU 性能发展速度不匹配趋势，极致高效对 CPU
  的使用；
- 兼容 Redis 协议：为了让 Redis 用户可以无障碍的接入
  Abase，以满足更大容量的存储需求，我们需要完全兼容 Redis
  协议。

在此背景下，Abase 团队于 2019 年年底开始孵化第二代 Abase 系统。结合业界的先进架构方案及公司内部实践过程中的积累和思考，团队推出了资源池化，支持多租户、多写、CRDT 的软硬件一体化设计的新一代 NoSQL 数据库
—— Abase2。

# 架构概览

# 数据模型

Abase 支持 Redis 的几种主要数据结构与相应接口：

- String: 支持
  Set、Append、IncrBy，是字节线上使用最为广泛的数据模型；
- Hash/Set：使用率仅次于
  String，在部分更新/查询的结构化数据存取场景中广泛使用；
- ZSet: 广泛应用于榜单拉链等在线业务场景，区别于直接使用
  String+Scan 方式进行包装，Abase 在 ZSet
  结构中做了大量优化，从设计上避免了大量 ZIncrBy
  造成的读性能退化；
- List/TTLQueue: 队列接口语义使业务在对应场景下非常方便地接入。

# 架构视图

图 1：Abase2 整体架构图

Abase2
的整体架构主要如上图所示，在用户、管控面、数据面三种视角下主要包含 5 组核心模块。

# RootServer

线上一个集群的规模大约为数千台机器，为管理各个集群，我们研发了 RootServer 这个轻量级组件。顾名思义，RootServer 拥有全集群视角，它可以更
好地协调各个
集群之间的资源配比，支持
租户在不同集群之间的数据迁移，提供容灾视图并合理控制爆炸半径。

# MetaServer

Abase2 是多租户中心化架构，而 MetaServer

则是整个架构的总管理员，它主要包括以下核心功能：

- 管理元信息的逻辑视图：包括 Namespace，Table，Partition，Replica 等状态和配置信息以及之间的关系；
- 管理元信息的物理视图：包括 IDC，Pod，Rack，DataNode，Disk，Core 的分布和 Replica 的位置关系；
- 多租户 QoS 总控，在异构机器的场景下根据各个租户与机器的负载进行副本 Balance 调度；
- 故障检测，节点的生命管理，数据可靠性跟踪，在此基础上进行节点的下线和数据修复。

图 2: 集群物理视图

图 3: 集群逻辑视图

# DataNode

DataNode 是数据存储节点。部署时，可以每台机器或者每块盘部署一个 DataNode，为方便隔离磁盘故障，线上实际采用每块盘部署一个 DataNode 的方式。

DataNode 的最小资源单位是 CPU Core（后简称 Core），每个 Core 都拥有一个独立的 Busy Polling 协程框架，多个 Core 共享一块盘的空间与 IO 资源。

图 4：DataNode 资源视角

一个 Core 包含多个 Replica，每个 Replica 的请求只会在一个 Core 上 Run-to-Complete，可以有效地避免传统多线程模式中上下文切换带来的性能损耗。

Replica 核心模块如下图所示，整个 Partition 为 3 层结构：

- 数据模型层：如上文提到的 String, Hash 等 Redis 生态中的各类数据结构接口。
- 一致性协议层：在多主架构下，多点写入势必会造成数据不一致，Anti-Entropy 一方面会及时合并冲突，另一方面将协调冲突合并后的数据下刷至引擎持久化层并协调 WAL GC。
- 数据引擎层：数据引擎层首先有一层轻量级数据暂存层（或称 Conflict Resolver）用于存储未达成一致的数据；下层为数据数据引擎持久化层，为满足不同用户多样性需求，Abase2 引设计上采用引擎可插拔模式。对于有顺序要求的用户可以采用 RocksDB，TerarkDB 这类 LSM 引擎，对于无顺序要求点查类用户采用延迟更稳定的 LSH 引擎。

图 5: Replica 分层架构

# Client/Proxy/SDK

Client 模块是用户侧视角下的核心组件，向上提供各类数据结构的接口，向下一方面通过 MetaSync 与 MetaServer 节点通信获取租户 Partition 的路由信息，另一方面通过路由信息与存储节点 DataNode 进行数据交互。此外，为了进一步提高服务质量，我们在 Client 的 IO 链路上集成了重试、Backup Request、热 Key 承载、流控、鉴权等重要 QoS 功能。

结合字节各类编程语言生态丰富的现状，团队基于 Client 封装了 Proxy 组件，对外提供 Redis 协议（RESP2)与 Thrift 协议，用户可根据自身偏好选择接入方式。此外，为了满足对延迟更敏感的重度用户，我们也提供了重型 SDK 来跳过 Proxy 层，它是 Client 的简单封装。

# DTS (Data Transfer Service)

DTS 主导了 Abase 生态系统的发展，在一二代 透明迁移、备份回滚、Dump 、订阅等诸多业务场景中起到了非常核心的作用，由于篇幅限制，本文不做更多的详细设计叙述。

# 关键技术

# 一致性策略

我们知道，分布式系统难以同时满足
强一致性、高可用性和正确处理网络故障（CAP ）这三种特性，因此系统设计者们不得不做出权衡，以牺牲某些特性来满足系统主要需求和目标。比如大多数数据库系统都采用牺牲极端情况下系统可用性的方式来满足数据更高的一致性和可靠性需求。

Abase2 目前支持两种同步协议来支持不同一致性的需求：

多主模式（Multi-Leader）：相对于数据强一致性，Abase
的大多数使用者们则对系统可用性有着更高的需求，Abase2 主要通过多主技术实现系统高可用目标。在多主模式下，分片的任一副本都可以接受和处理读写请求，以确保分片只要有任一副本存活，即可对外提供服务。同时，为了避免多主架构按序同步带来的一些可用性降低问题， 我们结合了无主架构的优势，在网络分区、进程重启等异常恢复后，并发同步最新数据和老数据。此外，对于既要求写成功的数据要立即读到，又不能容忍主从切换带来的秒级别不可用的用户，我们提供无更新场景下的写后读一致性给用户进行选择。实现方式是通过 Client 配置 Quorum 读写（W+R>N），通常的配置为
W=3，R=3，N=5。

单主模式（Leader&Followers）：Abase2 支持与一代系统一样的主从模式，并且，半同步适合于对一致性有高要求，但可以忍受一定程度上可用性降低的使用场景。与 MySQL
半同步类似。系统将选择唯一主副本，来处理用户的读写请求，保证至少 2 个副本完成同步后，才会通知用户写入成功。以保证读写请求的强一致性，并在单节点故障后，新的主节点仍然有全量数据。

未来也会提供更多的一致性选择，来满足用户的不同需求。

# 读写流程

下面我们将详细介绍在多主模型下 Abase
的数据读写流程以及数据最终一致的实现方案。

对于读请求，Proxy 首先根据元信息计算出请求对应的分片，再根据地理位置等信息将请求转发到该分片某一个合适的 Replica 上，Replica Coordinator 根据一致性策略查询本地或远端存储引擎后将结果按照冲突解决规则合并后返

回给 Proxy，Proxy 根据对应协议将结果组装后返回给用户。

对于写请求，Proxy 将请求转发到合适的 Replica 上，Replica Coordinator 将写请求序列化后并发地发送至所有
Replica，并根据一致性策略决定请求成功所需要的最少成功响应数
W。可用性与 W 成反比，W=1 时可获得最大的写可用性。

如图 6 所示，假设分片副本数 N=3，当用户写请求到达 Proxy 后，Proxy 根据地理位置等信息将请求转发到分片的某一个副本（Replica B），Replica B 的 Coordinator 负责将请求写入到本地，且并发地将请求 forward 到其他 Replica，当收到成功写入的响应数大于等于用户配置的 W 时(允许不包括本地副本)，即可认为请求成功，若在一定时间内（请求超时时间）未满足上述条件，则认为请求失败。

在单个副本内，数据首先写入到 WAL 内，保证数据的持久化，然后提交到引擎数据暂存层。引擎在达到一定条件后将缓存数据下刷到持久化存储，然后 WAL 对应数据即可被 GC。

一个 Core 内所有 Replica 共享一个 WAL，可以尽量合并不同 Replica 的碎片化提交，减少 IO 次数。引擎层则由 Replica 独占，方便根据不同业务场景对引擎层做精细化配置，同时也便于数据查询、GC 等操作。

图 6: 写流程示意图

用户可以根据一致性、可用性、可靠性与性能综合考虑 NWR 的配比，W（R）为 1 时可获得最大的写（读）可用性与性能；调大 W/R 则可在数据一致性和可靠性方面取得更好的表现。

## Anti-Entropy

由上述写流程可以看到，当 W<N 时，部分副本写入成功即可认为请求成功，而由于网络抖动等原因数据可能并未在所有副本上达成一致状态，我们通过 Anti-Entropy 机制异步地完成数据一致性修复。

为了便于检测分片各个 Replica 间的数据差异，我们在 WAL 之上又构建了一层 ReplicaLog（索引），每个 Replica 都对应一个由自己负责的 ReplicaLog，并会在其他 Replica 上创建该 ReplicaLog 的副本，不同 Replica 接收的写请求将写到对应的 ReplicaLog

内，并分配唯一严格递增的 LogID，我们称为 Seqno。

每个 Replica 的后台 Anti-Entropy 任务将定期检查自身与其他 Replica 的 ReplicaLog 的进度，以确定自身是否已经拥有全部数据。流程如下：

1. 获取自身 ReplicaLog 进度向量[Seqno1, Seqno2..., SeqnoN]；
2. 与其他 Replica 通信，获取其他 Replica 的进度向量；
3. 比对自身与其他 Replica 进度向量，是否有 ReplicaLog 落后于其他 Replica，如果是则进入第 4 步，否则进入第 5 步；
4. 向其他 Replica 发起数据同步请求，从其他 Replica 拉取缺少的 ReplicaLog 数据，并提交到引擎层
5. 若已就某 ReplicaLog 在 SeqnoX 之前已达成一致，回收 SeqnoX 之前的 ReplicaLog 数据。

另外，正常情况下副本间数据能做到秒级达成一致，因此 ReplicaLog 通常只需要构建在内存中，消耗极少的内存，即可达到数据一致的目的。在极端情况下（如网络分区），ReplicaLog 将被 dump 到持久化存储以避免 ReplicaLog 占用过多内存。

与 DynamoDB、Cassandra 等通过扫描引擎层构建 merkle tree 来完成一致性检测相比，Abase
通过额外消耗少量内存的方式，能更高效的完成数据一致性检测和修复。

## 冲突解决

多点写入带来可用性提升的同时，也带来一个问题，相同数据在不同 Replica 上的写入可能产生冲突，检测并解决冲突是多写系统必须要处理的问题。

为了解决冲突，我们将所有写入数据版本化，为每次写入的数据分配一个唯一可比较的版本号，形成一个不可变的数据版本。

Abase 基于 Hybrid Logical Clock 算法生成全局唯一时间戳，称为 HLC timestamp，并使用 HLC timestamp
作为数据的版本号，使得不同版本与时间相关联，且可比较。

通过业务调研，我们发现在发生数据冲突时，大部分业务希望保留最新写入的数据，部分业务自身也无法判断哪个版本数据更有意义（复杂的上下游关系），反而保留最新版本数据更简洁也更有意义，因此 Abase 决定采用 Last Write Wins 策略来解决写入冲突。

在引擎层面，最初我们采用 RocksDB 直接存储多版本数据，将 key 与版本号一起编码，使得相同 key 的版本连续存储在一起；查询时通过 seek 方式找到最新版本返回；同时通过后台版本合并任务和 compaction filter 将过期版本回收。

在实践中我们发现，上述方式存在几个问题：

1. 多版本数据通常能在短时间内（秒级）决定哪个版本最终有效，而直接将所有版本写入 RocksDB，使得即使已经确定了最终有效数据，也无法及时回收无效的版本数据；同时，使用 seek 查询相比 get 消耗更高，性能更低。
2. 需要后台任务扫描所有版本数据完成无效数据的回收，消耗额外的 CPU 和 IO 资源。
3. 引擎层与多版本耦合，使得引擎层无法方便地做到插件化，根据业务场景做性能优化。

为了解决以上问题，我们把引擎层拆分为数据暂存层与通用引擎层，数据多版本将在暂存层完成冲突解决和合并，只将最终结果写入到底层通用引擎层中。

得益于 Multi-Leader 与 Anti-Entropy 机制，在正常情况下，多版本数据能在很小的时间窗口内决定最终有效数据，因此数据暂存层通常只需要将这个时间窗口内的数据缓存在内存中即可。Abase 基于 SkipList 作为数据暂存层的数据结构（实践中直接使用 RocksDB memtable），周期性地将冲突数据合并后写入底层。

图 7：数据暂存层基本结构示意图

# CRDTs

对于幂等类命令如 Set，LWW 能简单有效地解决数据冲突问题，但 Redis String 还需要考虑 Append, Incrby 等非幂等操作的兼容，并且，其它例如 Hash, ZSet 等数据结构则更为复杂。于是，我们引入了 CRDT 支持，实现了 Redis 常见数据结构的 CRDT，包括 String/Hash/Zset/List，并且保持语义完全兼容 Redis。

以 IncrBy 为例，由于 IncrBy 与 Set 会产生冲突，我们发现实际上难以通过 State-based 的 CRDT 来解决问题， 故而我们选用 Operation-based 方案，并结合定期合并 Operation 来满足性能要求。

为了完全兼容 Redis 语义，我们的做法如下：

1. 给所有 Operation 分配全球唯一的 HLC timestamp，作为操作的全排序依据；
2. 记录写入的 Operation 日志（上文 ReplicaLog），每个 key 的最终值等于这些 Operation 日志按照时间戳排序后合并的结果。副本间只要 Operation 日志达成一致，最终状态必然完全一致；
3. 为了防止 Operation 日志过多引发的空间和性能问题，我们定期做 Checkpoint，将达成一致的时间戳之前的操作合并成单一结果；
4. 为了避免每次查询都需要合并 Operation 日志带来的性能开销，我们结合内存缓存，设计了高效的查询机制，将最终结果缓存在 Cache 中，保证查询过程不需要访问这些 Operation 日志。

图 8：Operation-based CRDT 数据合并示意图

完整 CRDT 的实现算法和工程优化细节我们将在后续 Abase2 介绍文章中详细说明。

## 全球部署

结合多主模式，系统可以天然支持全球部署，同时，为了避免网状同步造成的带宽浪费，Abase2 在每个地域都可以设置一个 Main Replicator，由它来主导和
其它地域
间的数据同步。典型的应用场景有多中心数据同步场景以及边缘计算场景。

图 9: 多数据中心部署

图 10: 边缘-中心机房部署

## 多租户 QoS

为了实现资源池化，避免不同租户间资源独占造成浪费，Abase2 采用大集群多租户的部署模式。同时，为了兼顾不同场景优先级的资源隔离需求，我们在

集群内部划分了 3 类资源池，按照不同服务等级进行部署。如图：

图 11：资源池分类示意图

在资源池内的多租户混部要解决两个关键问题：

## 1、DataNode 的 QoS 保障

DataNode 将请求进行分类量化：

- 用户的请求主要归为 3 类：读、写、Scan，三类请求优先级各不相同；
- 不同数据大小的请求会被分别计算其成本，例如一个读请求的数据量每 4KB 会被归一化成 1 个读取单位。

所有的用户请求都会通过这两个条件计算出 Normalized Request Cost(NRC)。基于 NRC 我们构建了 Quota 限制加 WFQ 双层结构的服务质量控制模块。

图 12：IO 路径上的 QoS 示意图

如上图所示，用户请求在抵达租户服务层之前需要迈过两道关卡：

1. Tenant Quota Gate: 如果请求 NRC 已经超过了租户对应的配额，DataNode 将会拒绝该请求，保证 DataNode 不会被打垮；
2. 分级 Weight Fair Queue: 根据请求类型分发至各个 WFQ，保证各个租户的请求尽可能地被合理调度。

图 13(1)：正常状态延迟

图 13(2)：突增流量涌入后延迟

如图 13(2)所示，部分租户突增流量涌入后（蓝绿线）并未对其它租户造成较大影响。流量突增的租户请求延迟受到了一定影响，并且出现请求被 Tenant

Quota Gate
拦截的现象，而其它租户的请求调度却基本不受影响，延迟基本保持稳定。

### 2、多租户的负载均衡

负载均衡是所有分布式系统都需要的重要能力之一。资源负载实际上有多个维度， 包括磁盘空间、IO 负载， CPU
负载等。我们希望调度策略能高效满足如下目标：

- 同一个租户的 Replica 尽量分散，确保租户 Quota 可快速扩容；
- 不会因为个别慢节点阻塞整体均衡流程；
- 最终让每个机器的各个维度的资源负载百分比接近。

负载均衡流程的概要主要分为 3 个步骤：

1. 根据近期的 QPS 与磁盘空间使用率的最大值，为每个 Core 构建二维负载向量；
2. 计算全局最优二维负载向量，即资源池中所有 Core 负载向量在两个维度上的平均值；
3. 将高负载 Core 上的 Replica 调度到低负载 Core 上，使高、低负载的 Core 在执行 Replica 调度后，Core 的负载向量与最优负载向量距离变小。

图 14(1): 某集群均衡调度前的负载分布

图 14(2): 某集群均衡调度后的负载分布

上图是线上负载均衡前后各的负载分布散点图，其中：红点是最优负载向量，横纵分别表示 Core 负载向量的第一和第二维度，每个点对应一个 Core。从图可以看出，各个 Core
的负载向量基本以最优负载向量为中心分布。

## 现状与规划

目前 Abase2 正在逐渐完成对第一代 Abase 系统的数据迁移和升级，使用 Abase2 的原生多租户能力，我们预计可提升
50%的资源使用率。通过对异地多活架构的改造，我们将为 Abase

用户提供
更加准确、快速的多地域数据同步功能。同时，我们也在为火山引擎上推出 Abase 标准产品做准备，以满足公有云上用户的大容量、低成本 Redis 场景需求。

未来的 Abase2 会持续向着下面几个方向努力，我们的追求是

技术先进性：在自研多写架构上做更多探索，通过支持 RDMA/io_uring/ZNS SSD/PMEM 等新硬件新技术，让 Abase2 的各项指标更上一个台阶。

易用性：建设标准的云化产品，提供 Serverless 服务，和更自动的冷热沉降，更完善的 Redis 协议兼容，更高鲁棒性的 dump/bulkload 等功能。

极致稳定：在多租户的 QoS 实践和自动化运维等方面不断追求极致。我们的目标是成为像水和电那样，让用户感觉不到存在的基架产品。

## 结语

随着字节跳动的持续发展，业务数量和场景快速增加，业务对 KV 在线存储系统的可用性与性能的要求也越来越高。在此背景下，团队从初期的拿来主义演进到较为成熟与完善的 Abase 一代架构。秉持着追求极致的字节范儿，团队没有止步于此，我们向着更高可用与更高性能的目标继续演进 Abase2。由于篇幅限制，更多的细节、优化将在后续文章中重点分期讲述。

## 团队介绍

NoSQL 团队为公司提供稳定可靠的在线存储服务。目前已经覆盖公司几乎所有业务线，支持百亿级请求处理能力。团队依靠公司业务的快速发展浪潮，背靠基础架构的综合技术力量支持，结合最新硬件/技术发展趋势，致力于做用户喜爱的、技术领先的、追求极致的 KV 存储标杆产品。欢迎更多志同道合的同学加入我们：

- 联系邮箱：bytebase@bytedance.com
- 岗位描述：https://job.toutiao.com/s/FUTyXhw