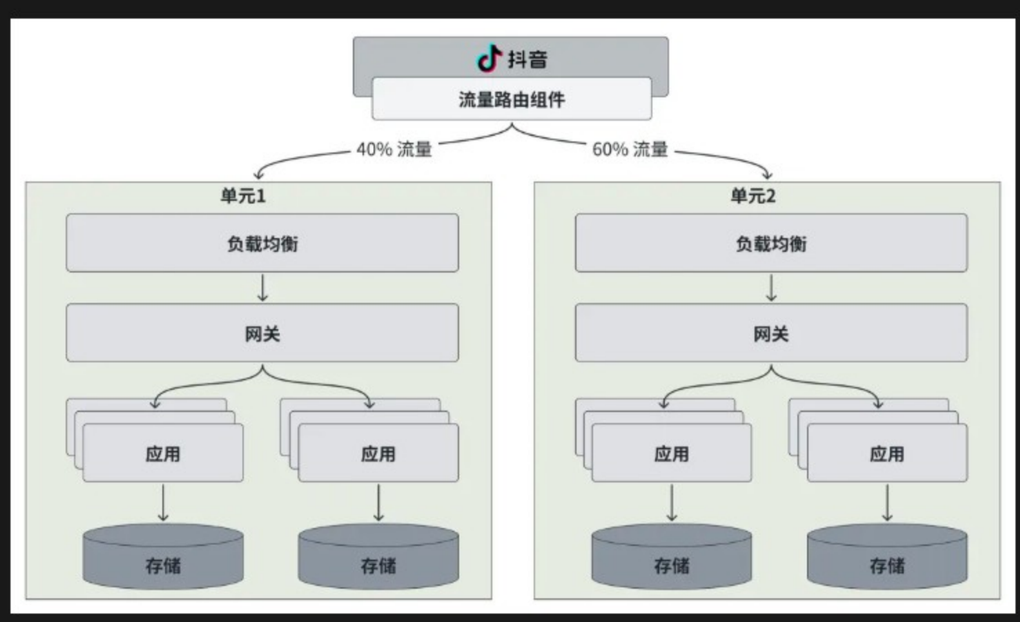
单元化架构在字节跳动的落地实践

InfoQ 2024年10月26日 10:15 新加坡

作者丨谢志旺，字节跳动 - 业务架构团队

什么是单元化

单元化的核心理念是将业务按照某种维度划分成一个个单元，理想情况下每个单元内部都是完成所有业务操作的自包含集合，能独立处理业务流程，各个单元均有其中一部分数据，所有单元的数据组合起来是完整的数据（各企业实际落地过程中会结合实际业务和基建情况做一些折中）。流量按照某种分区维度（例如流量所属用户）Sharding 到不同的单元，调度上按照流量携带的分区信息进行调度，保证同一时刻该分区的数据写入都在同一个单元，简化版示意图如下：



为什么要做单元化

业界各企业演进到单元化一般主要都是出于下面几个原因：

- **资源限制**：单机房受物理资源上限限制，同城多机房受地区的能评和供电等限制，无法做到机房的无限扩展，随着业务规模的扩大，长期一定会面临多地数据中心的布局；
- **合规要求**：全球化产品通常会面临不同地区的合规要求（例如欧盟的 GDPR），会有当地用户数据只能存储在当地的要求，业务天然需要考虑围绕不同的合规区域构建单元；
- **容灾考虑**：核心业务有城市级异地容灾需求，通过单元化方式可以构建异地单元，每个单元都有常态真实流量，流量可以灵活地在单元间进行调度。

除了上述最关键的问题外，建设单元化还能有其他方面的收益：

- **业务体验提升**：通过结合就近调度，能够将用户流量调度到最近的单元，从而降低请求耗时，提升用户体验；
- **成本方面**：相比于异地冷备，两地三中心等传统容灾架构，各个单元都能直接承载流量，减少资源冗余；
- **隔离方面**：在最小的单元范围内去做各种技术演进，能够有效控制风险半径。

异地单元化的主要挑战

机房延迟问题：一般同城机房之间物理距离 <200KM，网络延迟和机房内差别不大，大部分业务场景无需过多考虑。但跨城异地机房之间的延迟会大很多，例如北京和上海之间 RTT P99 达 40 毫秒，此时跨机房请求耗时需要慎重考虑，特别是单个请求如果出现多次跨机房，增加的请求耗时可能是几百毫秒甚至更大的。

数据同步问题：

- 容灾单元之间的数据需要互通，以支持一个单元故障时能够切流到另一个单元进行容灾，因此需要在单元间进行数据同步；

- 数据同步需要考虑不同的存储引擎（数据库、缓存、消息队列等），不同引擎特性不一，实现成本和复杂度巨大；
- 跨城机房间的延迟大而且是弱网环境，网络的质量很难保证，进一步导致数据同步质量保证难度大。

流量路由的问题：所谓流量路由也就是如何将流量调度到正确的单元问题，需要考虑在请求链路哪个环节（客户端、流量入口、内网 RPC、存储层等）、根据请求什么信息（用户 ID、地理位置等）进行用户归属单元的识别，以及如何进行走错单元流量的纠偏。

数据正确性问题：

- 例如归属 单元 1 的用户 A 评论了归属 单元 2 的用户 B 的抖音短视频，系统在 单元 1 给 B 发了一个通知，但 B 查看评论的流量被按 B 的单元归属调度到了 单元 2，由于数据同步延迟问题，B 打开抖音后看不到评论。业务上需要感知这类同步延迟带来的正确性问题；
- 另外两个单元的数据库构建了双向数据同步后，如果同一个用户短时间在两个单元读写同一份数据，可能会出现数据冲突问题。

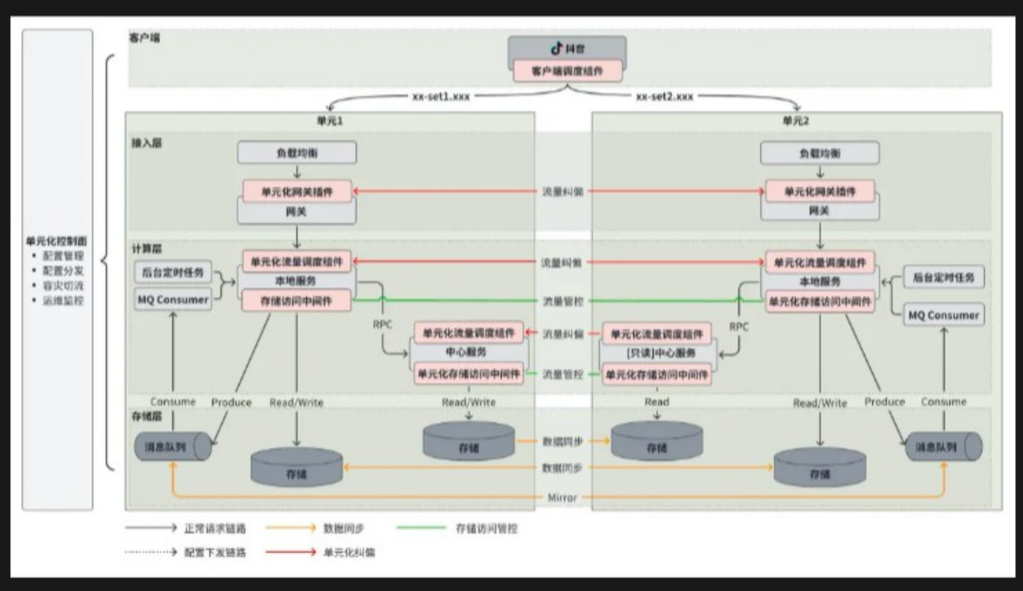
成本问题：

- 每个单元都需要有完整的业务资源以及支撑这些业务资源（计算、存储、网络）的基础设施（运维平台、观测平台等），需要综合考虑对成本的影响；
- 异地单元化的改造落地涉及到包括基建、架构、业务多方的配合支持，需要考虑人力上的成本。

管理复杂度问题：随着单元的增加，不同的单元都需要管理对应的服务和基建，管理和运维的复杂性会有较大增加。

字节跳动异地单元化架构

在本文中，我们仅对字节跳动在**中国大陆**的单元化架构做讨论，目前我们的单元化部署架构如下图所示：



我们围绕客户端选路、接入层纠偏、计算层纠偏、存储访问层管控四个维度构建了单元化流量调度和管控能力，通过技术手段确保单元化流量调度的正确性（将流量调度到正确的单元）和数据访问的正确性（数据不写脏），具体来说：

- **客户端**：在客户端通过调度组件将用户流量从第一跳开始就调度到正确的单元，减少在内网的跨单元流量；
- **接入层**：在网关通过网关的开放能力实现调度插件，按需对客户端调度出错的流量进行路由信息计算和纠偏；
- **计算层**：计算层通过研发框架和 Mesh 的开放能力实现流量切面，确保对未经过网关的内部 RPC 流量的路由信息计算和纠偏；
- **存储层**：通过存储访问中间件和 Mesh 的开放能力实现流量切面，确保对存储层路由出错 / 异常单元化流量进行审计和拦截。

目前包括抖音、抖音电商、抖音支付、抖音直播、抖音本地生活等业务均启动了异地单元化改造落地，生产环境已接入数千个核心微服务、超过 100 万实例。

单元化架构落地的关键问题

如何选择单元的维度

单元维度的选择很大程度上决定了单元化架构的水平扩展能力、运维成本和容灾方式，需要结合业务推进单元化架构想要解决的核心问题（资源、合规、容灾）和业务特性来选型。业界实践大部分是围绕物理维度（例如 Region、机房）构建单元，也有少部分是逻辑维度的。

结合我们面临的业务特性：

- 同时有包括社交、电商、本地生活、直播、支付等多种类型的业务，不同业务在单元化架构上有不同考虑；
- 存在大量中台，各中台同时支持不同类型的业务，这些中台需要适配好不同上游业务的单元化流量和数据；
- 业务之间有比较复杂的依赖，例如电商和本地生活有从抖音入口进来的流量，也有自己独立入口的流量。

综合考虑，我们认为字节跳动的单元应该是物理维度的，单元的建设随着基建的规划和建设去演进，而不是业务自行构建，这种物理意义上的唯一调度依据能够保证不同的业务长期演进中能够在一个单元内闭环，避免调度混乱。

同时考虑我们目前最核心要解决的问题是资源和容灾问题，因此我们最终选择是 Region 维度构建单元，形成当前的 同城容灾 + 异地多活 容灾架构。

如何选择分区维度

分区维度决定了数据和流量在单元间的划分标准，选择分区维度的时候需要重点考虑：

- 每个分区对应的数据互相不能重叠，以保证同一时刻同一个分区维度的数据写只在一个单元；
- 分区的粒度需要能支撑流量调度的灵活性要求，确保流量能按预期分布到各个单元；
- 路由信息的计算需要足够轻量；
- 确保单元内调用逻辑尽可能闭环，避免产生跨单元调用。

一般 ToC 类业务最常见的就是以用户作为分区维度，我们目前大部分业务选择用户（UserID）作为分区维度（抖音、电商等业务），少部分选择 Region 维度（搜索等业务）。

如何进行流量的单元化调度

管理分区和单元的映射

分区和单元的映射也就是对于一个请求我们找到它应该调度到哪个单元的依据，需要综合业务上对管理成本和灵活性的要求来考虑，我们通过以下两种方式结合来管理：

- 映射表：通过 UserID -> Set 的映射表管理，面向 QA 测试、高价值用户群灰度等明确指定归属单元场景；
- 表达式：通过流量分片的方式，按比例将流量在单元间分配，以支持比较低的管理和计算成本，例如 0 <= Hash(UserID) % 100000 < 70000 -> Set1 、70000 <= Hash(UserID) % 100000 < 100000 -> Set2。

计算路由信息和纠偏

路由信息的计算逻辑一般不复杂，更关键的是决策需要在哪些环节计算以及怎么纠偏（把走错单元的流量调度到正确单元），一般来说，整个请求链路包括客户端、接入层、计算层和存储层四层，需要结合公司基建情况和业务要求来决策落地方式，以下是我们对于各层实现纠偏能力的必要性和实现方式上的思考和建议。

客户端：

- 必要性分析：在客户端直接计算流量归属单元并调度，可以减少在内网出现跨单元访问的情况，减少跨单元导致的耗时影响；
- 实现方式：可以在客户端集成调度组件来实现，具体调度逻辑需要结合单元的维度来设计，例如最简单可以不同单元对应不同域名，按 UserID 映射到对应域名。

接入层（负载均衡、网关）：

- 必要性分析：客户端由于改造依赖用户升级、同时网络环境复杂导致配置变更时生效时间无法保证，因此存在无法 100% 保证流量调度正确的情况，需要在接入层兜底计算路由信息并对走错单元的流量进行纠偏；
- 实现方式：在流量入口，结合 LB / 网关的开放能力，实现路由信息的计算和流量纠偏能力。

计算层：

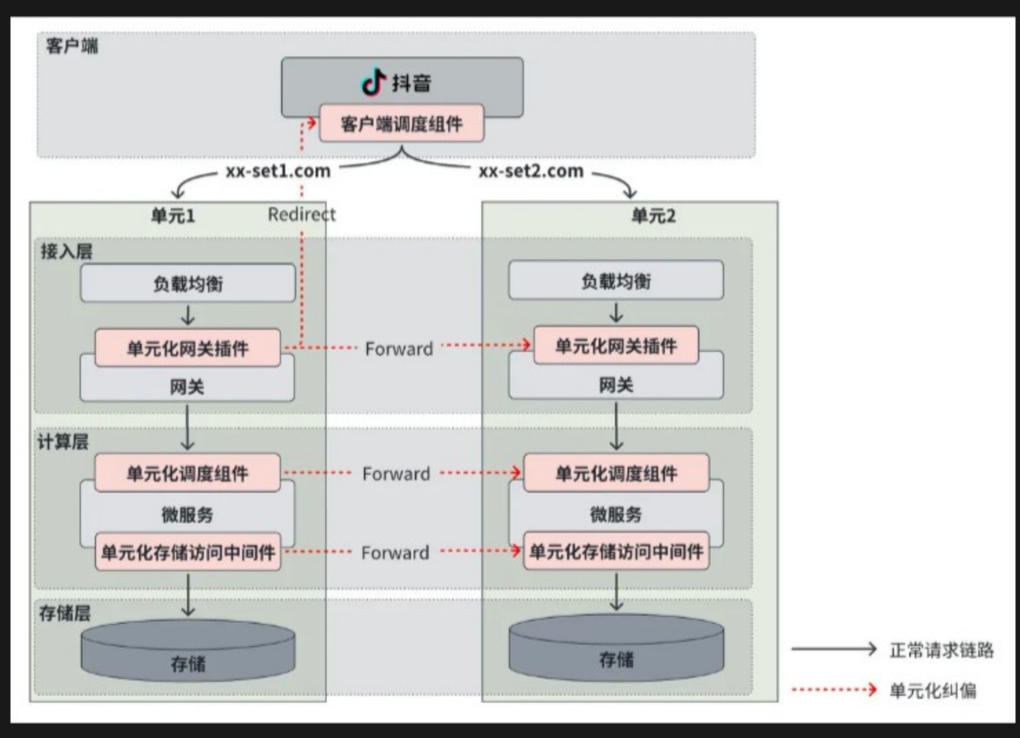
- 必要性分析：实际业务上，存在很多内部的 RPC 流量，例如消息队列的 Consumer 发起请求、后台定时任务发起的请求等，这种流量不经过接入层，需要在计算层 RPC 接口兜底进行路由信息计算并对走错单元的流量进行纠偏；
- 实现方式：可以结合研发框架或者 Service Mesh 的开放能力实现。

存储层：

- 必要性分析：业务上会存在例如一个用户去读写另一个用户的数据（例如社交场景的点赞、评论）这类不经过 RPC 接口调度的场景，需要在存储访问的时候兜底计算路由信息并对走错单元的流量进行纠偏；

- 实现方式：一般可以结合存储访问中间件或者 ServiceMesh 开放能力实现。实际落地的时候，考虑到存储访问层实现纠偏成本和风险偏高（例如连接数风险），实际业务落地上可以考虑只做拦截，推动业务进行改造。

一个比较完整的分层示例：



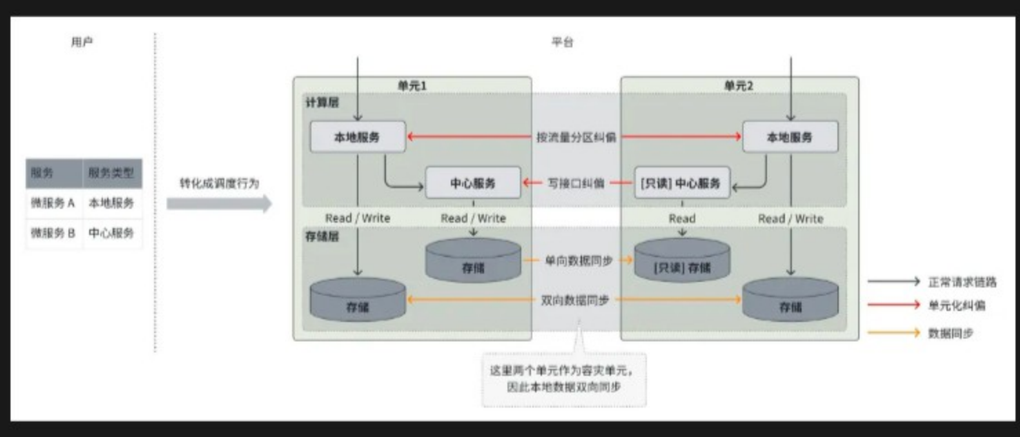
如何适配复杂的业务调度场景

理想的单元化架构是所有服务都能在单元内闭环，不出现跨单元的流量，但是在实际业务场景下很难满足理想的单元化架构。以电商场景为例，如果用买家作为分区维度，那商家的数据必然会被多单元读写，因此一定存在部分服务的流量不能单元化调度的情况，单元化架构需要能做好适配。和业界基本思路类似，我们对服务类型做了区分：

- **本地服务 /LocalService**：单元化拆分后能够本地闭环读写的业务服务，比如在电商场景的买家业务相关服务。业务的大部分微服务应当都是 LocalService 类似，否则倾向于部署单 vRegion 部署；

- **中心服务 /CentralService**：无法进行单元化拆分的业务，固定在某个单元部署，通常是一些对于数据一致性要求非常高的服务，例如电商的商家、商品库存等服务。

服务类型决定了流量调度方式和服务访问的数据的同步方式。基于此，研发核心需要关注业务上服务类型的定义即可，其他包括数据同步管理、流量调度的细节都可以由框架 / 平台内部收敛解决，极大降低业务上的理解和管理复杂度：



> 实际业务落地过程中，甚至会出现同一个服务不同接口的访问行为不一致的情况，需要细化到接口维度区分类型，和服务类型类似设计即可，这里不赘述。

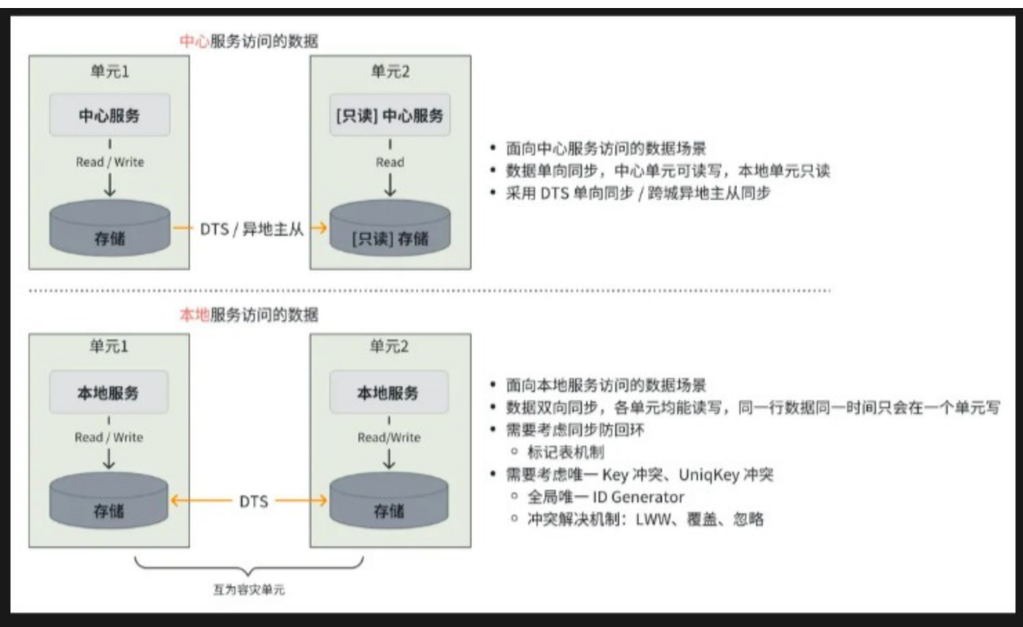
如何进行多单元数据管理

单元间数据同步

单元化架构下的数据同步有两种场景：

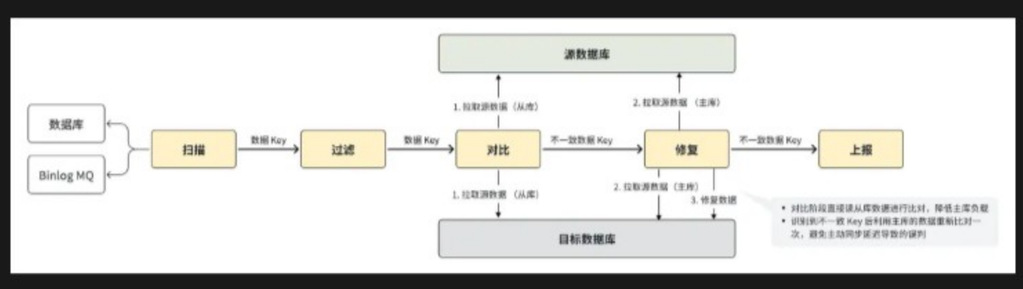
- 中心服务访问的数据在单元间单向同步，支持延迟敏感且接受一定程度数据不一致的业务场景本地只读；

- 本地服务访问的数据在容灾单元间的双向同步，支持一个单元出现故障的时候能够将流量切到容灾单元进行容灾，数据同步需要考虑好防回环和唯一 Key 冲突处理。

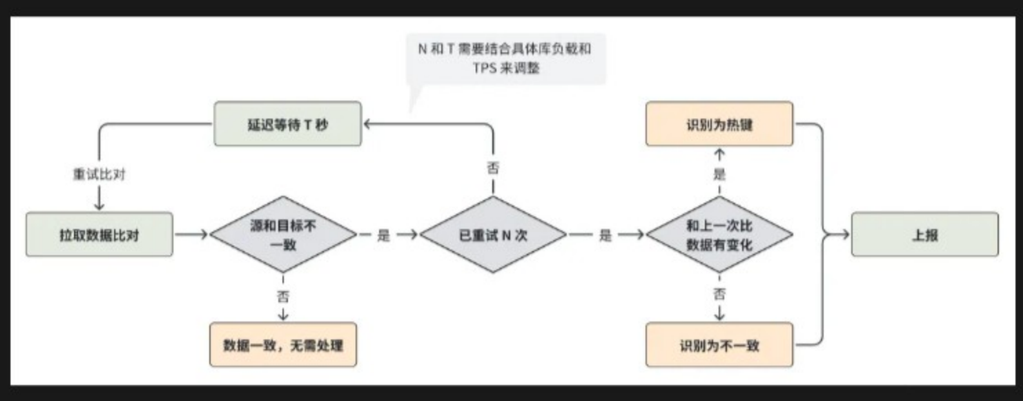


数据同步一致性比对

数据对账一般是全增结合，实时增量比对确保 Diff 发现的实时性，但写入 TPS 高时会有误差，周期全量比对兜底保证整体一致率，常见的一致性比对流程如下：



设计增量数据一致性比对能力的时候，需要重点考虑对热键的检测和处理，部分热键一直在 Update 的话需要能识别出来，否则会导致持续有不一致误报警，一个简单的基于重试比对的实现如下：



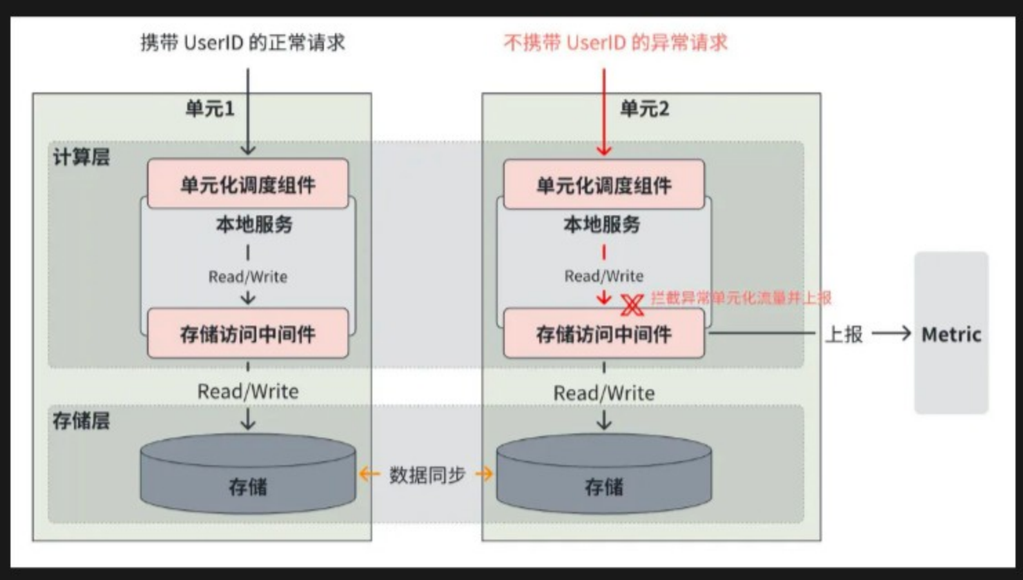
如何保证数据多活的正确性

日常态：异常单元化流量的识别和拦截

以 UserID 作为单元化分区维度的话，在实际业务场景可能出现两种异常情况：

- 从请求里面无法解析到正常的 UserID 或者未接入流量调度组件导致未正常计算路由信息；
- 一个用户的流量在代码逻辑层直接访问数据库，写了另一个非归属本单元的用户的数据。

这两种情况我们都认为是异常单元化流量，在存储访问层通过管控中间件支持了识别和拦截能力，兜底保障数据不被写脏：

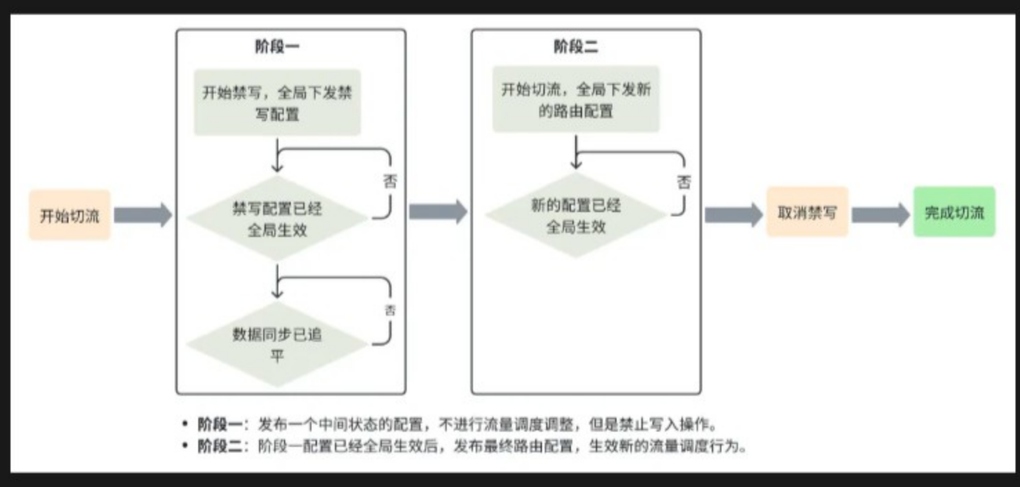


切流态：避免数据同步延迟导致脏写

在做单元间切流的时候，由于下面两个问题，可能导致数据出现脏写：

- 跨城异地单元之间的数据同步延迟一般接近或达到秒级，用户流量从一个单元切换到另一个单元的时候可能由于数据还未同步完成从而出现脏写；
- 单元间切流本质上是分区和单元映射配置的变化和重新下发的过程，在大规模分布式架构下，这份配置需要下发到多个独立的实例上去生效，这些不同的实例生效时间无法完全一致，就导致同一个用户的流量短时间可能进入不同单元从而导致数据脏写。

我们对切流过程引入 两阶段配置变更 + 存储访问禁写 来解决上述数据脏写问题，整体切流流程如下：



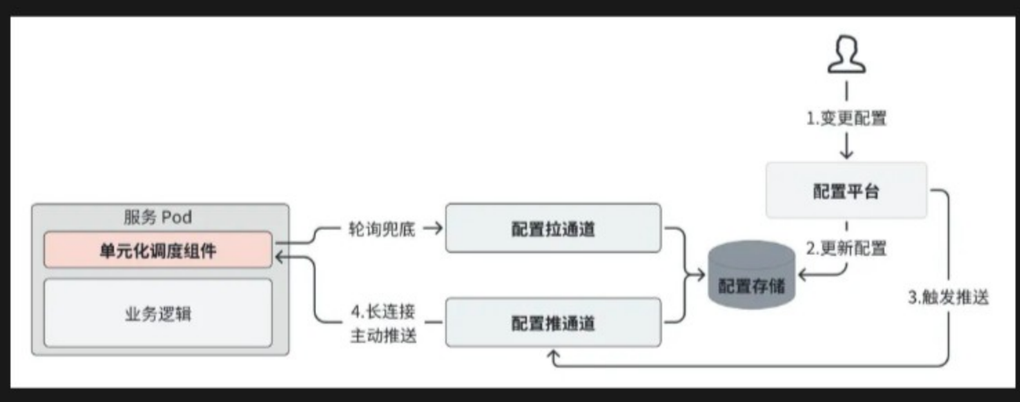
得益于我们整体单元化调度和存储访问中间件设计，我们的禁写是 UserID 维度的，能够控制到每次切流仅禁写切流中的用户，将切流的影响做的尽可能小。

如何确保切流的可靠性和风险控制

优化配置下发时效

结合前面关于切流期间通过 两阶段配置变更 + 存储访问禁写 来避免数据出现脏写的说明，可以看到配置下发生效的时间对于禁写时长是一个非常大的影响因素，而禁写生效时长直接影响切流那部分用户的使用体验，因此我们需要尽可能提升配置下发时效。

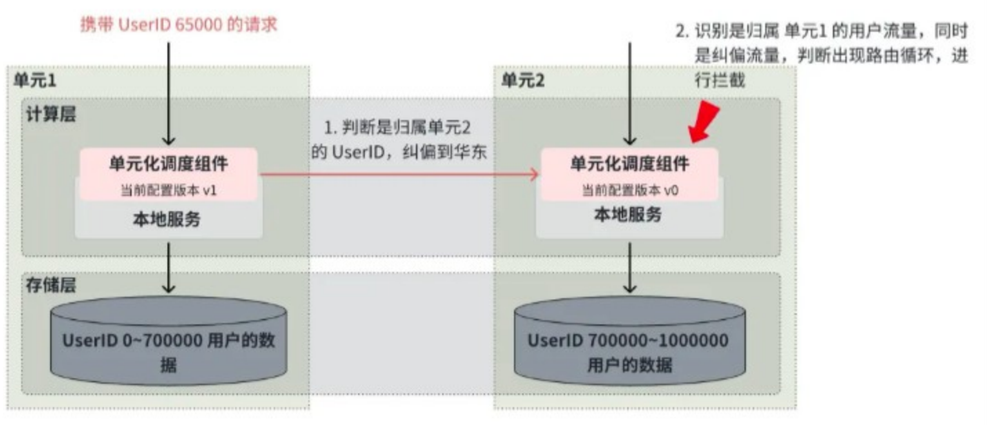
目前字节接入单元化的 Pod 数已超过 100 万，在这么大规模的实例数下快速下发调度配置的挑战是非常大的，我们通过 长连接主动推送 + 定时轮询拉取 的方式进行配置分发，提高配置收敛速度，减少配置不一致中间态的时间：



防止路由死循环

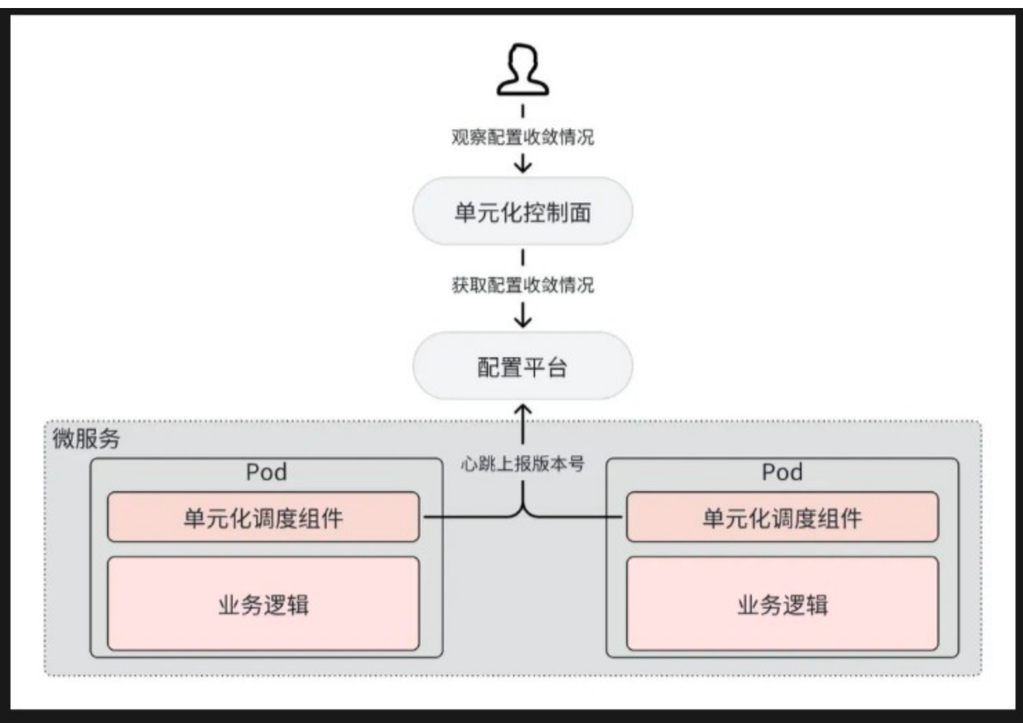
配置发布过程中，同一个 PSM 不同单元实例的生效时间不一样，可能导致流量纠偏到目标单元后由于目标单元重新计算纠偏回来，出现路由死循环的情况。我们通过单元化调度组件在流量上下文中记录并传递纠偏次数，拦截重复纠偏的流量，及时阻断回环流量：

配置版本	单元1 UserID 分片范围	单元2 UserID 分片范围
变更前 v0 版本	0~70000	70000~100000
变更后 v1 版本	0~60000	60000 ~ 100000

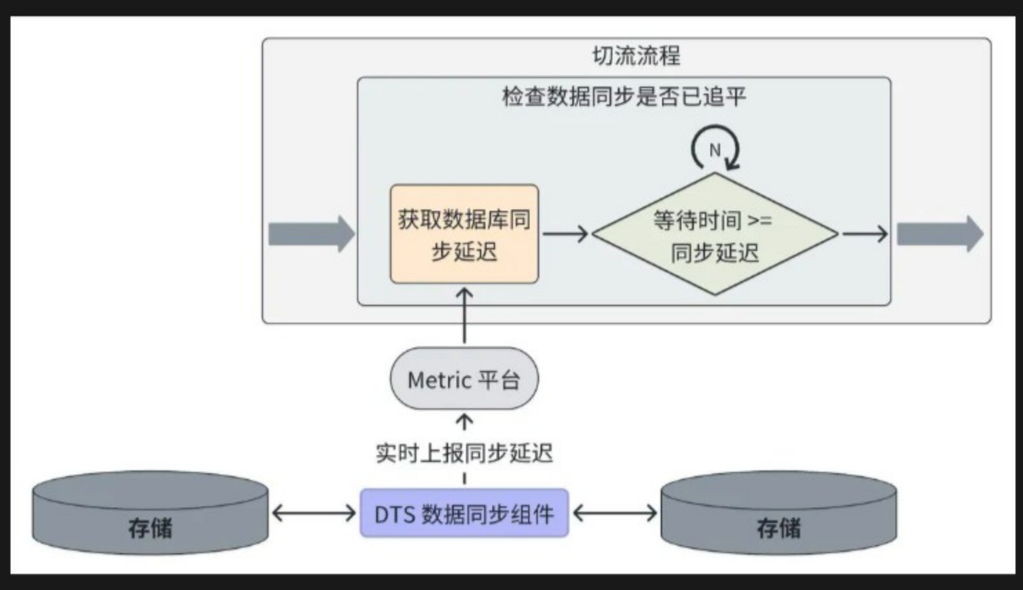


切流过程业务架构各层状态检查

- **多实例配置生效版本观测检查**：每次配置发布会分配一个递增的版本号，数据面组件通过长连接上报 Pod 当前配置版本号给配置中心，从而支持单元化控制面查询和观测配置收敛情况：



- **数据同步延迟观测检查**：由于我们的切流禁写是用户维度的，不是整个数据库禁写，禁写过程中数据库并不是完全没有新的写流量，因此我们是根据 数据实时的同步延迟 + 禁写等待时间 结合来判断切流范围的用户禁写后的数据是否已经同步：

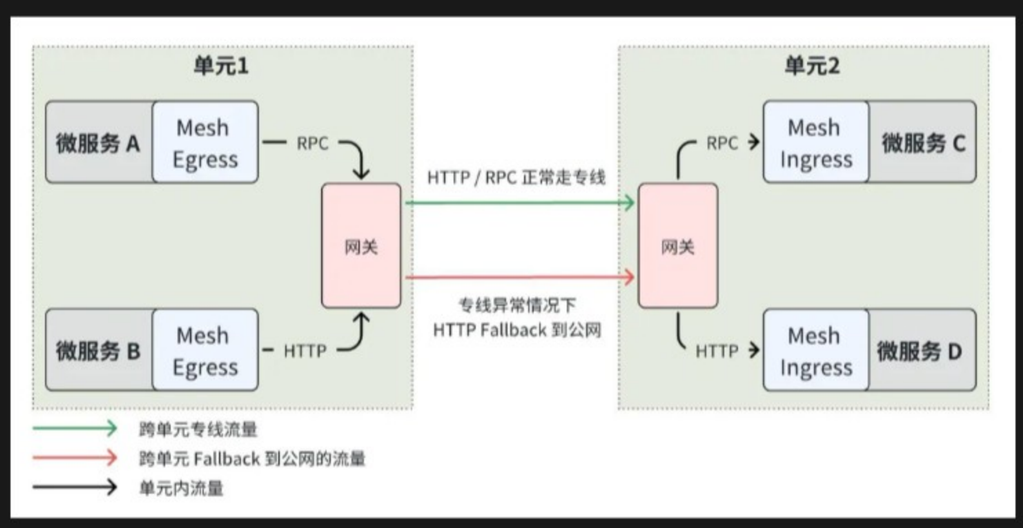


业务核心指标观测检查：切流过程中对业务成功率、负载、延迟等指标的观测。

如何保证跨地区 RPC 质量

单元之间通常物理距离较远，网络专线建设难度大成本高，网络质量（延迟、丢包、可用带宽）也差于同城机房间。前面介绍的各种单元路由纠偏都会产生跨单元 RPC，相比单元内 RPC 其成功率 / 耗时指标有所劣化。为了提升跨单元 RPC 质量，除了网络专线本身的稳定性建设（属于网络基建范畴，此处不展开），也可以在架构设计和带宽使用策略上进行针对性的优化。

跨单元 RPC 通道收敛：在单元化流量路由场景，某个流量很大的 RPC 服务可能仅小比例命中跨单元纠偏，此时由于下游实例数量很多可能导致连接复用效果较差。此时如果让所有跨单元 RPC 先统一经过本地边界网关，然后传输到其他单元的边界网关，最后再转发给实际下游，则建连过程拆分成了 3 段，中间段（网关之间）被所有跨单元 RPC 共用，可以保持较高的链接复用率； 另外 2 段为本地建连，即使连接复用率低，带来的耗时增加也较少。此外收敛到网关通道也更容易集中对跨单元 RPC 进行其他优化，比如按接口等级进行 QoS 管控、在专线完全故障情况下对重要的控制信令做加密后 Fallback 到公网传输等：



跨单元网络分级 QoS 管控：长距离专线建设成本高，带宽有限，也更容易因为各类异常导致可用带宽变少，无法满足所有场景的跨单元网络传输的需求，因此需要对不同类型的跨单元流量进行分级 QoS 管控。跨机房 RPC 失败要么直接影响用户体验，要么导致故障无法操作止损，应在跨单元网络传输中给予更高的优先级；离线数据传输容易短时间大量占用带宽，异常对用户感知相对不明显，应给以较低优先级，严格限制上限，必要时还应让出带宽：

流量类型		数据量	传输异常的影响	优先级
RPC调用	内部平台控制指令	很小	不直接影响用户请求，但是内部平台不能做变更发布，故障时无法止损	最高
	业务请求RPC	较大	请求处理失败，用户感知明显，或者耗时增加导致体验变差	
在线存储数据同步		较大	写入后其他单元读不到或者读到旧版本，部分场景导致业务逻辑异常，部分场景用户感知相对不明显。通常有MQ缓冲，短时间的网络波动可以自行恢复	次高
离线模型/训练任务同步		最大	离线训练任务暂停，模型效果变差，但通常用户感知不太明显	最低

未来演进思考

多单元研发成本和效率优化

字节跳动从原本的单 Region 内同城容灾架构演进到多 Region 异地单元化架构周期比较短（一年半左右），基础设施对多 Region 视角的支持还比较不足，对业务的整体研发和业务管理成本偏高，需要将多 Region 的研发和业务管理成本打平到单 Region。

极致的成本优化

从计算资源成本视角：在原来三机房同城容灾模式下，每个机房需要预留 50% 的 Buffer 用于机房故障容灾，演进到异地单元化架构后，基于两个容灾单元间的六个机房，部分业务机房故障可以将流量分摊到其他五个机房，此时各机房仅需 20% 的 Buffer。

从存储资源成本视角：我们现在是 同城容灾 + 异地多活 的容灾模式，各单元都支持同城容灾，因此部分业务可以直接进行数据的单元化拆分，单元内各自只有一部分数据（加起来是全量数据），理想情况下存储成本减少一半。

更复杂的单元化架构演进

未来字节跳动在国内会有更多的区域，不同业务在各单元的排布模型会越来越复杂，结合我们复杂的业务依赖关系，这里的流量调度模型、数据单元化和同步模型都需要演进。

未来区域增多后，业务随着发展机房排布会调整，可能会需要在非容灾单元之间调整流量，此时存在数据单元化拆分和用户维度数据单元间搬迁能力，需要解决用户维度数据的识别和低成本搬迁问题。

更完善的数据多活能力

字节跳动目前的存储对 AP 场景更友好（侧重抖音这种社交类场景），主要围绕单 Region 构建，在多单元场景下对于电商、支付类（对数据一致要求非常高）的业务支持较弱，在异地单元化架构下强依赖数据同步能力来支持多单元数据多活能力，业务上的限制偏大（例如写只能统一在一个单元），有跨 Region 强一致数据库的需求。

今日好文推荐

太古可口可乐：将经验转化为数据，探索行业专有模型

字节澄清大模型遭攻击，实习生嫁祸背刺同事？IBM老员工被无端解雇后举报董事长；年薪40万美元因滥用25美元餐补被裁！| Q资讯

国产编程语言MoonBit发布原生后端，比Java快15倍，拥抱 RISC-V

37signals"下云"计划完美收官：成本节约比当初估算的还要多，5年狂省千万美元