企业微信客户端中组织架构数据的同步更新方案优化实战

JackJiang Lv.9 9 个月前 只看大图

【概述】：作为企业级的微信，在业务快速发展的背景下，迭代优化的要求也越发急迫。企业微信初版的全量同步方案在快速的业务增长面前已经不堪重负，针对其遇到的问题，怎样做好组织架构同步优化？这是一篇来自微信团队的技术实战。

【作者】：胡腾，腾讯工程师，参与企业微信从无到有的整个过程，目前主要负责企业微信移动端组织架构和外部联系人等模块的开发工作。

1、写在前面

企业微信在快速发展过程中，陆续有大企业加入使用，企业微信初版采用全量同步方案，该方案在大企业下存在流量和性能两方面的问题，每次同步消耗大量流量，且在 iPhone 5s 上拉取 10w+ 成员架构包解压时会提示 memory warning 而应用崩溃。

全量同步方案难以支撑业务的快速发展，优化同步方案越来越有必要。本文针对全量同步方案遇到的问题进行分析，提出组织架构增量同步方案，并对移动端实现增量同步方案的思路和重难点进行了讲解。

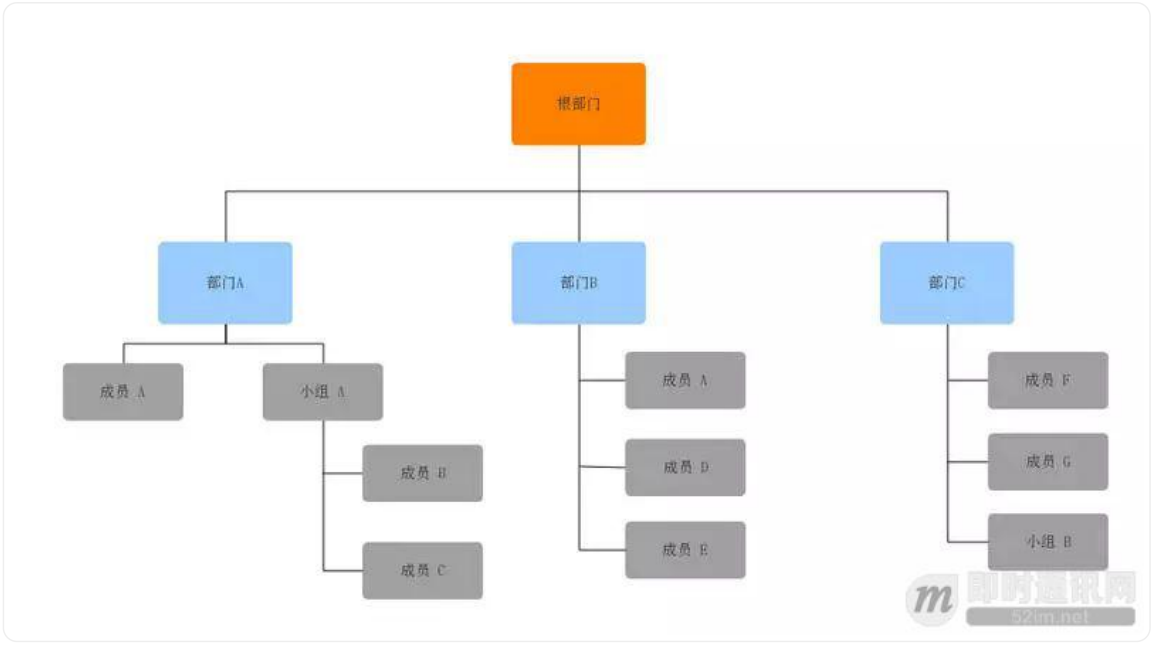
2、企业微信业务背景

在企业微信中，组织架构是非常重要的模块，用户可以在首页的 tab 上选择"通讯录"查看到本公司的组织架构，并且可以通过"通讯录"找到本公司的所有成员，并与其发起会话或者视频语音通话。

组织架构是非常重要且敏感的信息，企业微信作为企业级产品，始终把用户隐私和安全放在重要位置。针对组织架构信息，企业管理员具有高粒度隐私保护操作权限，不仅支持个人信息隐藏，也支持通讯录查看权限等操作。

在企业微信中，组织架构特征有：

- **1）多叉树结构：** 叶子节点代表成员，非叶子节点代表部门。部门最多只有一个父部门，但成员可属于多个部门：



- **2）架构隐藏操作：** 企业管理员可以在管理后台设置白名单和黑名单，白名单可以查看完整的组织架构，其他成员在组织架构里看不到他们。黑名单的成员只能看到自己所在小组和其所有的父部门，其余人可以看到黑名单的成员；
- **3）组织架构操作：** 企业管理员可以在 web 端和 app 端添加 / 删除部门，添加 / 删除 / 移动 / 编辑成员等操作，并且操作结果会及时同步给本公司所有成员。

3、全量同步方案的问题

本节大致讲解下全量同步方案实现以及遇到的问题。

3.1 全量同步方案原理

企业微信在 1.0 时代，从稳定性以及快速迭代的角度考虑，延用了企业邮通讯录同步方案，采取了全量架构同步方案。

核心思想为服务端下发全量节点，客户端对比本地数据找出变更节点。此处节点可以是用户，也可以是部门，将组织架构视为二叉树结构体，其下的用户与部门均为节点，若同一个用户存在多个部门下，被视为多个节点。

全量同步方案分为首次同步与非首次同步：

- 首次同步服务端会下发全量的节点信息的压缩包，客户端解压后得到全量的架构树并展示；
- 非首次同步分为两步：
  - 服务端下发全量节点的 hash 值。客户端对比本地数据找到删除的节点保存在内存中，对比找到新增的节点待请求具体信息；
  - 客户端请求新增节点的具体信息。请求具体信息成功后，再进行本地数据库的插入 / 更新 / 删除处理，保证同步流程的原子性。

3.2 用户反馈

初版上线后，收到了大量的组织架构相关的 bug 投诉，主要集中在：

- 流量消耗过大；
- 客户端架构与 web 端架构不一致；
- 组织架构同步不及时。

这些问题在大企业下更明显。

3.3 问题剖析

深究全量同步方案难以支撑大企业同步的背后原因，皆是因为采取了服务端全量下发 hash 值方案的原因，方案存在以下问题：

- **拉取大量冗余信息：**
  即使只有一个成员信息的变化，服务端也会下发全量的 hash 节点。针对几十万人的大企业，这样的流量消耗是相当大的，因此在大企业要尽可能的减少更新的频率，但是却会导致架构数据更新不及时；
- **大企业拉取信息容易失败：**
  全量同步方案中首次同步架构会一次性拉取全量架构树的压缩包，而超大企业这个包的数据有几十兆，解压后几百兆，对内存不足的低端设备，首次加载架构可能会出现内存不足而 crash。非首次同步在对比出新增的节点，请求具体信息时，可能遇到数据量过大而请求超时的情况；
- **客户端无法过滤无效数据：**
  客户端不理解 hash 值的具体含义，导致在本地对比 hash 值时不能过滤掉无效 hash 的情况，可能出现组织架构展示错误。

优化组织架构同步方案越来越有必要。

4、寻找优化思路

寻求同步方案优化点，我们要找准原来方案的痛点以及不合理的地方，通过方案的调整来避免这个问题。

4.1 组织架构同步难点

准确且耗费最少资源同步组织架构是一件很困难的事情，难点主要在：

- **组织架构架构数据量大：**
  消息 / 联系人同步一次的数据量一般情况不会过百，而企业微信活跃企业中有许多上万甚至几十万节点的企业，意味着架构一次同步的数据量很轻松就会上千上万。移动端的流量消耗是用户非常在乎的，且内存有限，减少流量的消耗以及减少内存使用并保证架构树的完整同步是企业微信追求的目标；
- **架构规则复杂：**
  组织架构必须同步到完整的架构树才能展示，而且企业微信里的涉及到复杂的隐藏规则，为了安全考虑，客户端不应该拿到隐藏的成员；
- **修改频繁且改动大：**
  组织架构的调整存在着新建部门且移动若干成员到新部门的情况，也存在解散某个部门的情况。而员工离职也会通过组织架构同步下来，意味着超大型企业基本上每天都会有改动。

4.2 技术选型 - 提出增量更新方案

上述提到的问题，在大型企业下会变得更明显。

在几轮方案讨论后，我们给原来的方案增加了两个特性来实现增量更新：

- **增量：** 服务端记录组织架构修改的历史，客户端通过版本号来增量同步架构；
- **分片：** 同步组织架构的接口支持传阈值来分片拉取。

在新方案中，服务端针对某个节点的存储结构可简化为：

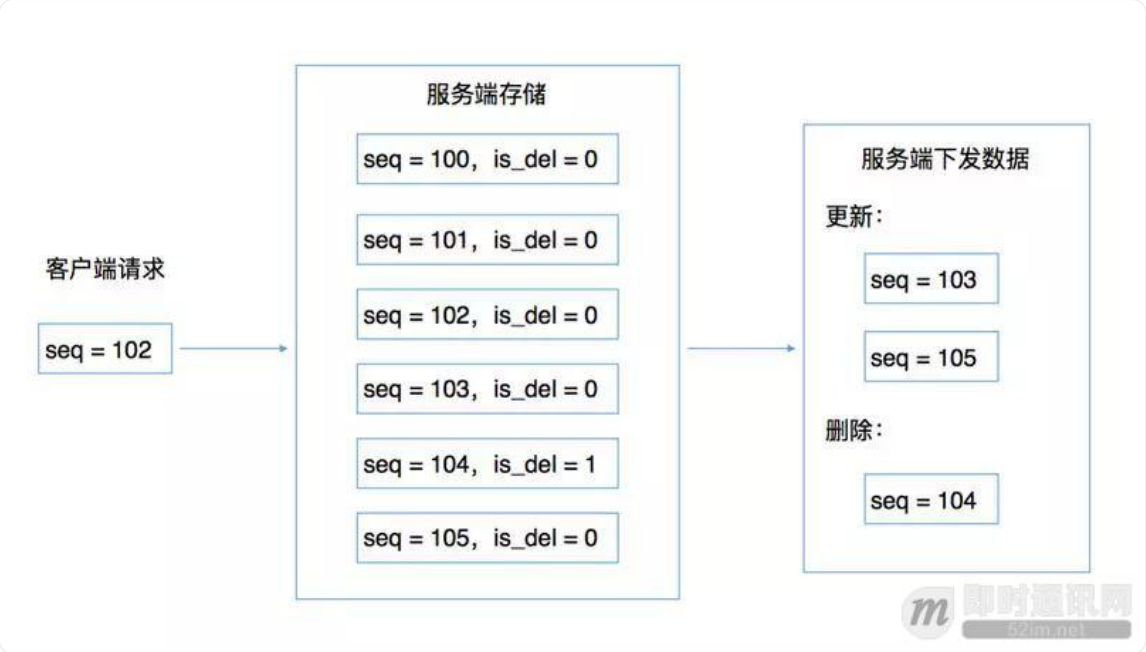
seq	vid	departmentid	is_delete

vid 是指节点用户的唯一标识 id，departmentid 是指节点的部门 id，is_delete 表示该节点是否已被删除：

- 若节点被删除了，服务端不会真正的删除该节点，而将 is_delete 标为 true；
- 若节点被更新了，服务端会增大记录的 seq，下次客户端来进行同步便能同步到。

其中，seq 是自增的值，可以理解成版本号。每次组织架构的节点有更新，服务端增加相应节点的 seq 值。客户端通过一个旧的 seq 向服务器请求，服务端返回这个 seq 和 最新的 seq 之间所有的变更给客户端，完成增量更新。

图示为：



通过提出增量同步方案，我们从技术选型层面解决了问题，但是在实际操作中会遇到许多问题，下文中我们将针对方案原理以及实际操作中遇到的问题进行讲解。

5、增量同步方案

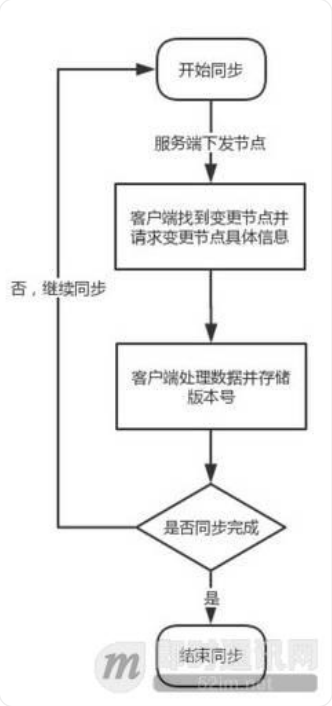
本节主要讲解客户端中增量同步架构方案的原理与实现，以及基础概念讲解。

5.1 增量同步方案原理

企业微信中，增量同步方案核心思想为:
服务端下发增量节点，且支持传阈值来分片拉取增量节点，若服务端计算不出客户端的差量，下发全量节点由客户端来对比差异。

- **增量同步方案可抽象为四步完成：**
- 客户端传入本地版本号，拉取变更节点；
- 客户端找到变更节点并拉取节点的具体信息；
- 客户端处理数据并存储版本号；
- 判断完整架构同步是否完成，若尚未完成，重复步骤 1，若完成了完整组织架构同步，清除掉本地的同步状态。

忽略掉各种边界条件和异常状况，增量同步方案的流程图可以抽象为：



接下来我们再看看增量同步方案中的关键概念以及完整流程是怎样的。

5.2 版本号

同步的版本号是由多个版本号拼接成的字符串，版本号的具体含义对客户端透明，但是对服务端非常重要。

版本号的组成部分为：

版本数据	描述	作用
db_version	同步使用版本号	增量同步使用。该 version 记录的当前客户端的最大 seq，服务端对比数据后将比该 verison 大的数据返回给客户端
filter_version	隐藏规则版本号	版本号回退使用。记录隐藏规则的版本号，若服务端发现后台的隐藏规则版本号比客户端的隐藏规则版本号大，则下发全量节点给客户端
vid_version	用户规则版本号	版本号回退使用。因为存在隐藏规则的情况，用户本身若移动了部门，所见的整个架构都可能会有变化，此时需要提升该版本号。若发现服务端该版本号更大，则下发全量数据
del_version	物理删除的 seq 的版本号	版本号回退使用。若后台物理删除了数据，则记录下删除的最大 seq 是多少，若 db_version 比后台 del_version 数据小，则下发全量节点

5.3 版本号回退

增量同步在实际操作过程中会遇到一些问题：

- 服务端不可能永久存储删除的记录，删除的记录对服务端是毫无意义的而且永久存储会占用大量的硬盘空间。而且无效数据过多也会影响架构读取速度。当 is_delete 节点的数目超过一定的阈值后，服务端会物理删除掉所有的 is_delete 为 true 的节点。此时客户端会重新拉取全量的数据进行本地对比；
- 一旦架构隐藏规则变化后，服务端很难计算出增量节点，此时会下发全量节点由客户端对比出差异。

理想状况下，若服务端下发全量节点，客户端铲掉旧数据，并且去拉全量节点的信息，并且用新数据覆盖即可。但是移动端这样做会消耗大量的用户流量，这样的做法是不可接受的。所以若服务端下发全量节点，客户端需要本地对比出增删改节点，再去拉变更节点的具体信息。

增量同步情况下，若服务端下发全量节点，我们在本文中称这种情况为版本号回退，效果类似于客户端用空版本号去同步架构。从统计结果来看，线上版本的同步中有 4% 的情况会出现版本号回退。

5.4 阈值分片拉取

若客户端的传的 seq 过旧，增量数据可能很大。此时若一次性返回全部的更新数据，客户端请求的数据量会很大，时间会很长，成功率很低。针对这种场景，客户端和服务端需要约定阈值，若请求的更新数据总数超过这个阈值，服务端每次最多返回不超过该阈值的数据。若客户端发现服务端返回的数据数量等于阈值，则再次到服务端请求数据，直到服务端下发的数据数量小于阈值。

5.5 节点结构体优化

在全量同步方案中，节点通过 hash 唯一标示。服务端下发的全量 hash 列表，客户端对比本地存储的全量 hash 列表，若有新的 hash 值则请求节点具体信息，若有删除的 hash 值则客户端删除掉该节点信息。

在全量同步方案中，客户端并不能理解 hash 值的具体含义，并且可能遇到 hash 碰撞这种极端情况导致客户端无法正确处理下发的 hash 列表。

而增量同步方案中，使用 protobuf 结构体代替 hash 值，增量更新中节点的 proto 定义为：

```
message ArchNode
{
  optional ArchNodeType type = 1; // 节点类型
  optional uint64 vid = 2;        // 节点用户 id
  optional uint64 partyid = 3;    // 节点所在的部门 id
  optional uint64 seq = 4;        // 节点版本
}
```


在增量同步方案中，用 vid 和 partyid 来唯一标识节点，完全废弃了 hash 值。这样在增量同步的时候，客户端完全理解了节点的具体含义，而且也从方案上避免了曾经在全量同步方案遇到的 hash 值重复的异常情况。

并且在节点结构体里带上了 seq 。节点上的 seq 来表示该节点的版本，每次节点的具体信息有更新，服务端会提高节点的 seq，客户端发现服务端下发的节点 seq 比客户端本地的 seq 大，则需要去请求节点的具体信息，避免无效的节点信息请求。

5.6 判断完整架构同步完成

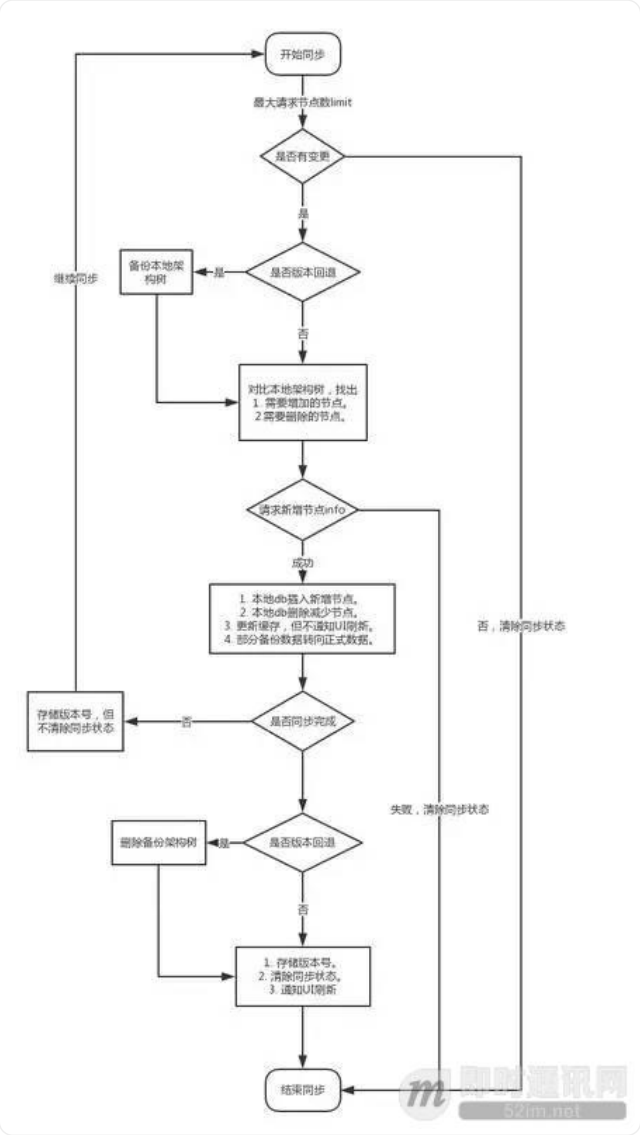
因为 svr 接口支持传阈值批量拉取变更节点，一次网络操作并不意味着架构同步已经完成。那么怎么判断架构同步完成了呢？

这里客户端和服务端约定的方案是：
若服务端下发的（新增节点 + 删除节点）小于客户端传的阈值，则认为架构同步结束。

当完整架构同步完成后，客户端需要清除掉缓存，并进行一些额外的业务工作，譬如计算部门人数，计算成员搜索热度等。

5.7 增量同步方案 - 完整流程图

考虑到各种边界条件和异常情况，增量同步方案的完整流程图为：



6、增量同步方案难点

6.1 难点

在加入增量和分片特性后，针对几十万人的超大企业，在版本号回退的场景，怎样保证架构同步的完整性和方案选择成为了难点。

前文提到，隐藏规则变更以及后台物理删除无效节点后，客户端若用很旧的版本同步，服务端算不出增量节点，此时服务端会下发全量节点，客户端需要本地对比所有数据找出变更节点，该场景可以理解为版本号回退。在这种场景下，对于几十万节点的超大型企业，若服务端下发的增量节点过多，客户端请求的时间会很长，成功率会很低，因此需要分片拉取增量节点。而且拉取下来的全量节点，客户端处理不能请求全量节点的具体信息覆盖旧数据，这样的话每次版本号回退的场景流量消耗过大。

因此，针对几十万节点的超大型企业的增量同步，客户端难点在于：

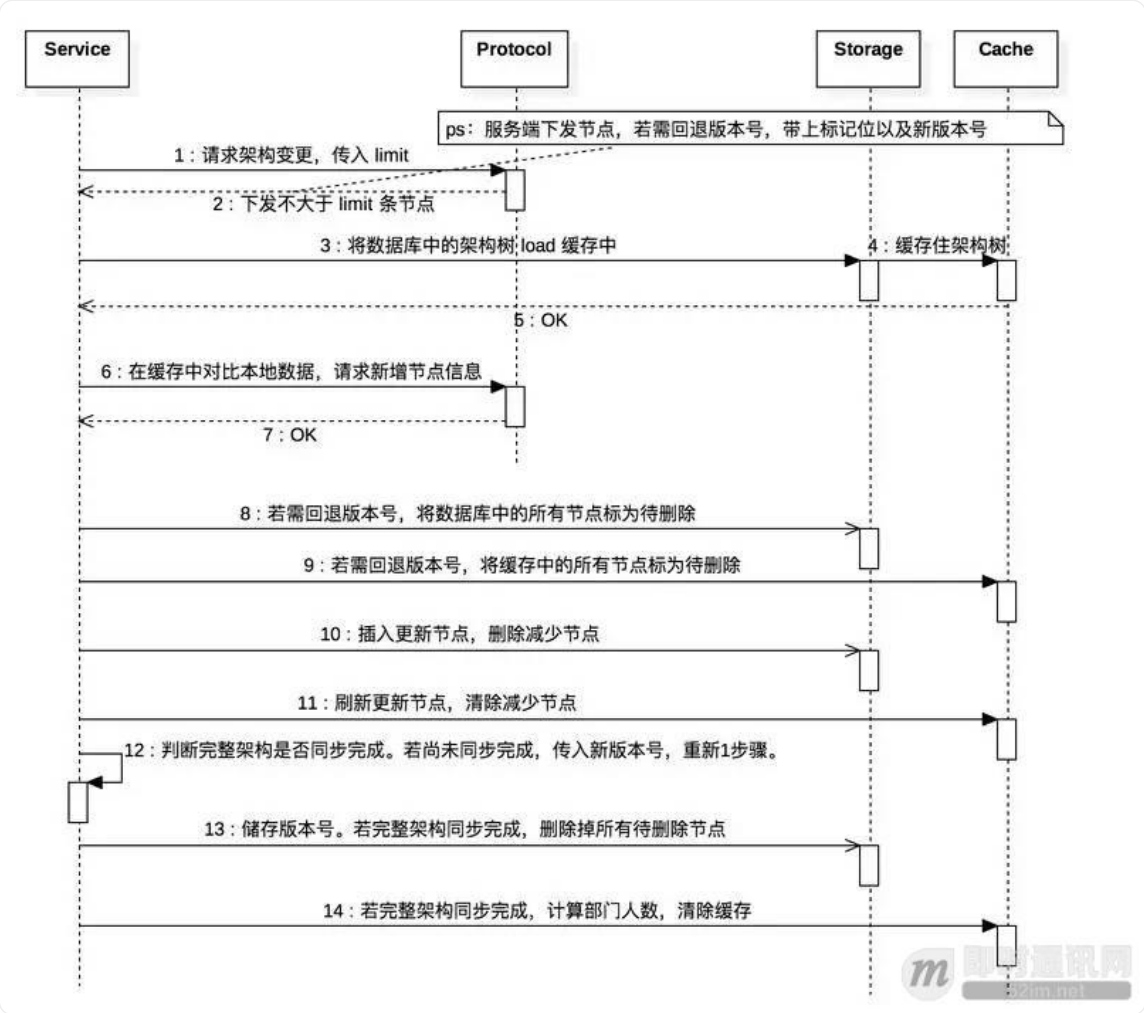
- **断点续传：** 增量同步过程中，若客户端遇到网络问题或应用中止了，在下次网络或应用恢复时，能够接着上次同步的进度继续同步；
- **同步过程中不影响正常展示：** 超大型企业同步的耗时可能较长，同步的时候不应影响正常的组织架构展示；
- **控制同步耗时：** 超大型企业版本号回退的场景同步非常耗时，但是我们需要想办法加快处理速度，减少同步的消耗时间。

6.2 思路

基本思路如下：

- 架构同步开始，将架构树缓存在内存中，加快处理速度；
- 若服务端端下发了需要版本号回退的 flag，本地将 db 中的节点信息做一次备份操作；
- 将服务端端下发的所有 update 节点，在架构树中查询，若找到了，则将备份数据转为正式数据。若找不到，则为新增节点，需要拉取具体信息并保存在架构树中；
- 当完整架构同步结束后，在 db 中找到并删除掉所有备份节点，清除掉缓存和同步状态。

若服务端下发了全量节点，客户端的处理时序图为：



6.2 服务端下发版本号回退标记

从时序图中可以看出，服务端下发的版本号回退标记是很重要的信号。

而版本号回退这个标记，仅仅在同步的首次会随着新的版本号而下发。在完整架构同步期间，客户端需要将该标记缓存，并且跟着版本号一起存在数据库中。在完整架构同步结束后，需要根据是否版本号回退来决定删除掉数据库中的待删除节点。

6.3 备份架构树方案

架构树备份最直接的方案是将 db 中数据 copy 一份，并存在新表里。如果在数据量很小的情况下，这样做是完全没有问题的，但是架构树的节点往往很多，采取这样简单粗暴的方案在移动端是完全不可取的，在几十万人的企业里，这样做会造成极大的性能问题。

经过考虑后，企业微信采取的方案是：

- 若同步架构时，后台下发了需要版本号回退的 flag，客户端将缓存和 db 中的所有节点标为待删除（时序图中 8，9 步）；
- 针对服务端下发的更新节点，在架构树中清除掉节点的待删除标记（时序图中 10,11 步）；
- 在完整架构同步结束后，在 db 中找到并删除掉所有标为待删除的节点（时序图中 13 步），并且清除掉所有缓存数据。

而且，在增量同步过程中，不应该影响正常的架构树展示。所以在架构同步过程中，若有上层来请求 db 中的数据，则需要过滤掉有待删除标记的节点。

6.4 缓存架构树

方案决定客户端避免不了全量节点对比，将重要的信息缓存到内存中会大大加快处理速度。

内存中的架构树节点体定义为：

```
namespace arch {
    typedef uint64_t vid_type;
    typedef uint64_t partyid_type;
    typedef uint64_t seq_type;
    typedef std::pair<vid_type, partyid_type> arch_node_type;

    typedef uint64_t parent_id_type;
};
```

```
struct ArchTreeNode {
    arch::seq_type seq;
    arch::parent_id_type parent_id; // 部门父节点，业务使用
    bool to_be_del; // 待删除标记
};
```

```
typedef std::map<arch::arch_node_type, ArchTreeNode> ArchNodeTreeMap;
```


此处我们用 std::map 来缓存架构树，用 std::pair 作为 key。我们在比较节点的时候，会涉及到很多查询操作，使用 map 查询的时间复杂度仅为 O(logn)。

7、增量同步方案关键点

本节单独将优化同步方案中关键点拿出来写，这些关键点不仅仅适用于本文架构同步，也适用于大多数同步逻辑。

7.1 保证数据处理完成后，再储存版本号

在几乎所有的同步中，版本号都是重中之重，一旦版本号乱掉，后果非常严重。

在架构同步中，最最重要的一点是：
保证数据处理完成后，再储存版本号。

在组织架构同步的场景下，为什么不能先存版本号，再存数据呢？
这涉及到组织架构同步数据的一个重要特征：架构节点数据是可重复拉取并覆盖的。

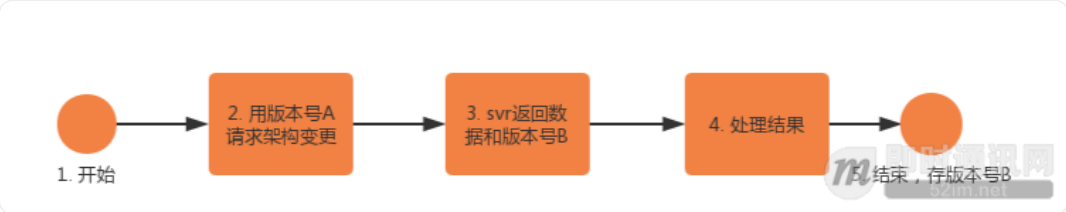
考虑下实际操作中遇到的真实场景：

- 若客户端已经向服务端请求了新增节点信息，客户端此时刚刚插入了新增节点，还未储存版本号，客户端应用中止了；
- 此时客户端重新启动，又会用相同版本号拉下刚刚已经处理过的节点，而这些节点跟本地数据对比后，会发现节点的 seq 并未更新而不会再去拉节点信息，也不会造成节点重复。

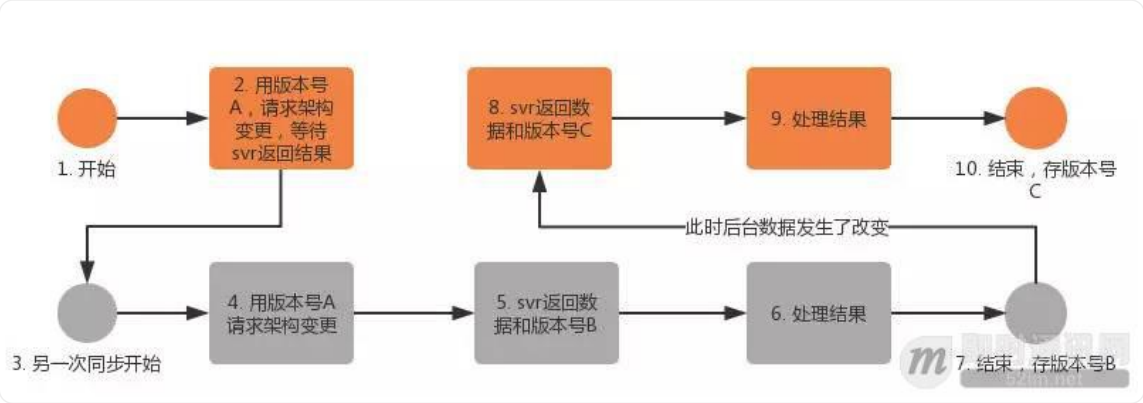
若一旦先存版本号再存具体数据，一定会有概率丢失架构更新数据。

7.2 同步的原子性

正常情况下，一次同步的逻辑可以简化为：



在企业微信的组织架构同步中存在异步操作，若进行同步的过程不保证原子性，极大可能出现下图所示的情况：



该图中，同步的途中插入了另外一次同步，很容易造成问题：

- **输出结果不稳定：** 若两次同步几乎同时开始，但因为存在网络波动等情况，返回结果可能不同，给调试造成极大的困扰；
- **中间状态错乱：** 若同步中处理服务端返回的结果会依赖于请求同步时的某个中间状态，而新的同步发起时又会重置这个状态，很可能会引起匪夷所思的异常；
- **时序错乱：** 整个同步流程应该是原子的，若中间插入了其他同步的流程会造成整个同步流程时序混乱，引发异常。

怎样保证同步的原子性呢？

我们可以在开始同步的时候记一个 flag 表示正在同步，在结束同步时，清除掉该 flag。若另外一次同步到来时，发现正在同步，则可以直接舍弃掉本次同步，或者等本次同步成功后再进行一次同步。

此外也可将同步串行化，保证同步的时序，多次同步的时序应该是 FIFO 的。

7.3 缓存数据一致性

移动端同步过程中的缓存多分为两种：

- **内存缓存：** 加入内存缓存的目的是减少文件 IO 操作，加快程序处理速度；
- **磁盘缓存：** 加入磁盘缓存是为了防止程序中止时丢失掉同步状态。

内存缓存多缓存同步时的数据以及同步的中间状态，磁盘缓存用于缓存同步的中间状态防止缓存状态丢失。

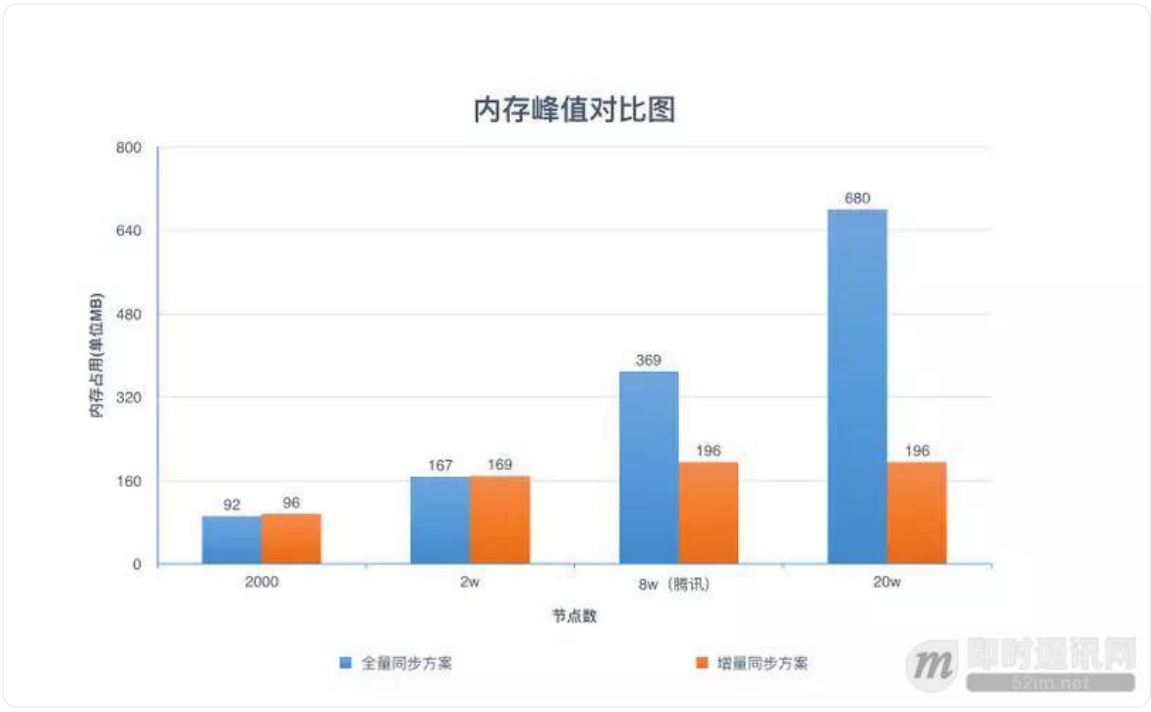
在整个同步过程中，我们都必须保证缓存中的数据和数据库的数据的更改需要一一对应。在增量同步的情况中，我们每次需要更新 / 删除数据库中的节点，都需要更新相应的缓存信息，来保证数据的一致性。

8、优化数据对比

8.1 内存使用

测试方法： 使用工具 Instrument，用同一账号监控全量同步和增量同步分别在首次加载架构时的 App 内存峰值。

内存峰值测试结果：



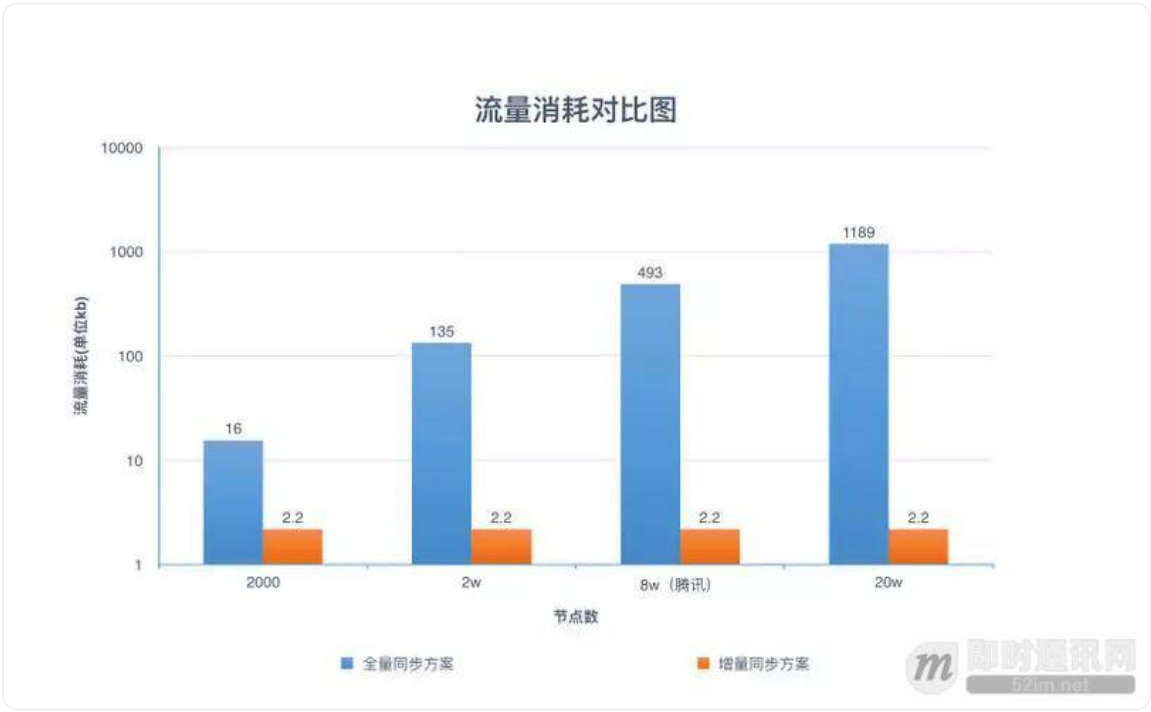
分析：
随着架构的节点增多，全量同步方案的内存峰值会一直攀升，在极限情况下，会出现内存不足应用程序 crash 的情况（实际测试中，30w 节点下，iPhone 6 全量同步方案必 crash）。而增量同步方案中，总节点的多少并不会影响内存峰值，仅仅会增加同步分片的次数。

优化后，在腾讯域下，增量同步方案的 App 总内存使用仅为全量同步方案的 53.1%，且企业越大优化效果越明显。并且不论架构的总节点数有多少，增量同步方案都能将完整架构同步下来，达到了预期的效果。

8.2 流量使用

测试方法： 在管理端对成员做增加操作五次，通过日志分析客户端消耗流量，取其平均值。日志会打印出请求的 header 和 body 大小并估算出流量使用值。

测试结果：



分析：
增加成员操作，针对增量同步方案仅仅会新拉单个成员的信息，所以无论架构里有多少人，流量消耗都是相近的。同样的操作针对全量同步方案，每次请求变更，服务端都会下发全量 hash 列表，企业越大消耗的流量越多。可以看到，当企业的节点数达到 20w 级别时，全量同步方案的流量消耗是增量同步方案的近 500 倍。

优化后，在腾讯域下，每次增量同步流量消耗仅为全量同步方案的 0.4%，且企业越大优化效果越明显。

9、写在最后

增量同步方案从方案上避免了架构同步不及时以及流量消耗过大的问题。通过用户反馈和数据分析，增量架构同步上线后运行稳定，达到了理想的优化效果。

（原文链接：点此进入）

附录：更多微信、QQ技术文章

[1] 有关QQ、微信的技术文章：

《企业微信客户端中组织架构数据的同步更新方案优化实战》
《微信团队披露：微信界面卡死超级bug“15。。。。”的来龙去脉》
《QQ 18年：解密8亿月活的QQ后台服务接口隔离技术》
《月活8.89亿的超级IM微信是如何进行Android端兼容测试的》
《以手机QQ为例探讨移动端IM中的“轻应用”》
《一篇文章get微信开源移动端数据库组件WCDB的一切！》
《微信客户端团队负责人技术访谈：如何着手客户端性能监控和优化》
《微信后台基于时间序的海量数据冷热分级架构设计实践》
《微信团队原创分享：Android版微信的臃肿之困与模块化实践之路》
《微信后台团队：微信后台异步消息队列的优化升级实践分享》
《微信团队原创分享：微信客户端SQLite数据库损坏修复实践》
《腾讯原创分享(一)：如何大幅提升移动网络下手机QQ的图片传输速度和成功率》
《腾讯原创分享(二)：如何大幅压缩移动网络下APP的流量消耗（下篇）》
《腾讯原创分享(二)：如何大幅压缩移动网络下APP的流量消耗（上篇）》
《微信Mars：微信内部正在使用的网络层封装库，即将开源》
《如约而至：微信自用的移动端IM网络层跨平台组件库Mars已正式开源》
《开源libco库：单机千万连接、支撑微信8亿用户的后台框架基石 [源码下载]》
《微信新一代通信安全解决方案：基于TLS1.3的MMTLS详解》
《微信团队原创分享：Android版微信后台保活实战分享(进程保活篇)》
《微信团队原创分享：Android版微信后台保活实战分享(网络保活篇)》
《Android版微信从300KB到30MB的技术演进(PPT讲稿) [附件下载]》
《微信团队原创分享：Android版微信从300KB到30MB的技术演进》
《微信技术总监谈架构：微信之道——大道至简(演讲全文)》
《微信技术总监谈架构：微信之道——大道至简(PPT讲稿) [附件下载]》
《如何解读《微信技术总监谈架构：微信之道——大道至简》》
《微信海量用户背后的后台系统存储架构(视频+PPT) [附件下载]》
《微信异步化改造实践：8亿月活、单机千万连接背后的后台解决方案》
《微信朋友圈海量技术之道PPT [附件下载]》
《微信对网络影响的技术试验及分析（论文全文）》
《一份微信后台技术架构的总结性笔记》
《架构之道：3个程序员成就微信朋友圈日均10亿发布量[有视频]》
《快速裂变：见证微信强大后台架构从0到1的演进历程（一）》
《快速裂变：见证微信强大后台架构从0到1的演进历程（二）》
《微信团队原创分享：Android内存泄漏监控和优化技巧总结》
《全面总结iOS版微信升级iOS9遇到的各种“坑”》
《微信团队原创资源混淆工具：让你的APK立减1M》
《微信团队原创Android资源混淆工具：AndResGuard [有源码]》
《Android版微信安装包“减肥”实战记录》
《iOS版微信安装包“减肥”实战记录》
《移动端IM实践：iOS版微信界面卡顿监测方案》
《微信“红包照片”背后的技术难题》
《移动端IM实践：iOS版微信小视频功能技术方案实录》
《移动端IM实践：Android版微信如何大幅提升交互性能（一）》
《移动端IM实践：Android版微信如何大幅提升交互性能（二）》
《移动端IM实践：实现Android版微信的智能心跳机制》
《移动端IM实践：WhatsApp、Line、微信的心跳策略分析》
《移动端IM实践：谷歌消息推送服务(GCM)研究（来自微信）》
《移动端IM实践：iOS版微信的多设备字体适配方案探讨》
《信鸽团队原创：一起走过 iOS10 上消息推送(APNS)的坑》
《腾讯信鸽技术分享：百亿级实时消息推送的实战经验》
>> 更多同类文章 ……

[2] 有关QQ、微信的技术故事：

《腾讯开发微信花了多少钱？技术难度真这么大？难在哪？》
《技术往事：创业初期的腾讯——16年前的冬天，谁动了马化腾的代码》
《技术往事：史上最全QQ图标变迁过程，追寻IM巨人的演进历史》
《技术往事："QQ群"和"微信红包"是怎么来的？》
《开发往事：深度讲述2010到2015，微信一路风雨的背后》
《开发往事：微信千年不变的那张闪屏图片的由来》
《开发往事：记录微信3.0版背后的故事（距微信1.0发布9个月时）》
《一个微信实习生自述：我眼中的微信开发团队》
《首次揭秘：QQ实时视频聊天背后的神秘组织》
>> 更多同类文章 ……

来源：即时通讯网 - 即时通讯开发者社区！

标签： 微信 移动端IM开发

本主题由 JackJiang 于 9 个月前 加入精华

上一篇： 请教如何学习IM的开发？ · 下一篇：微信手机端的本地数据全文检索优化之路

本帖已收录至以下技术专辑

QQ、微信技术分享 | 主题 72 · 关注 6 **有关微信、QQ的故事** | 主题 26 · 关注 5

相关文章

- 请教大佬，关于IM方案选型和技术调研的问题请教
- 即时通讯创业必读：解密微信的产品定位、创新思维、设计法则等
- 腾讯技术分享：微信小程序音视频技术背后的故事
- 微信多媒体团队梁俊斌访谈：聊一聊我所了解的音视频技术
- 最火移动端跨平台方案盘点：React Native、weex、Flutter
- 小白必读：闲话HTTP短连接中的Session和Token
- 微信多媒体团队访谈：音视频开发的学习、微信的音视频技术和挑战等
- 全面掌握移动端主流图片格式的特点、性能、调优等
- 从零开始搭建瓜子二手车IM系统(PPT) [附件下载]

推荐方案

MobileIMSDK (v3.3精编版)
轻量级开源移动端即时通讯框架。
快速入门 / 性能 / 指南 / 提问

MobileIMSDK-Web (有偿开源)
轻量级Web端即时通讯框架。
详细介绍 / 精编源码 / 手册教程

RainbowAV new (有偿开源)
移动端实时音视频框架。
详细介绍 / 性能测试 / 安装体验

RainbowChat (技术转让)
基于MobileIMSDK的移动IM系统。
详细介绍 / 产品截图 / 安装体验

评论 2

2 楼: 山鳅 *Lv.1* 8 个月前 引用此评论
厉害，总结的很到位

签名: 非常不错，技术又长进了

3 楼: x931609201 *Lv.2* 7 个月前 引用此评论
有一点很不理解，
对于5.4阈值分片拉取：
因为TCP拉取是可靠的，拉取大量的数据也是可靠的，唯一需要注意的一点就是，tcp 的网络层需要分包，只要调用send时候分包就好了，为什么还需要在业务层分包，这不是增加复杂度了吗？

+ 发新帖 发表评论 返回列表