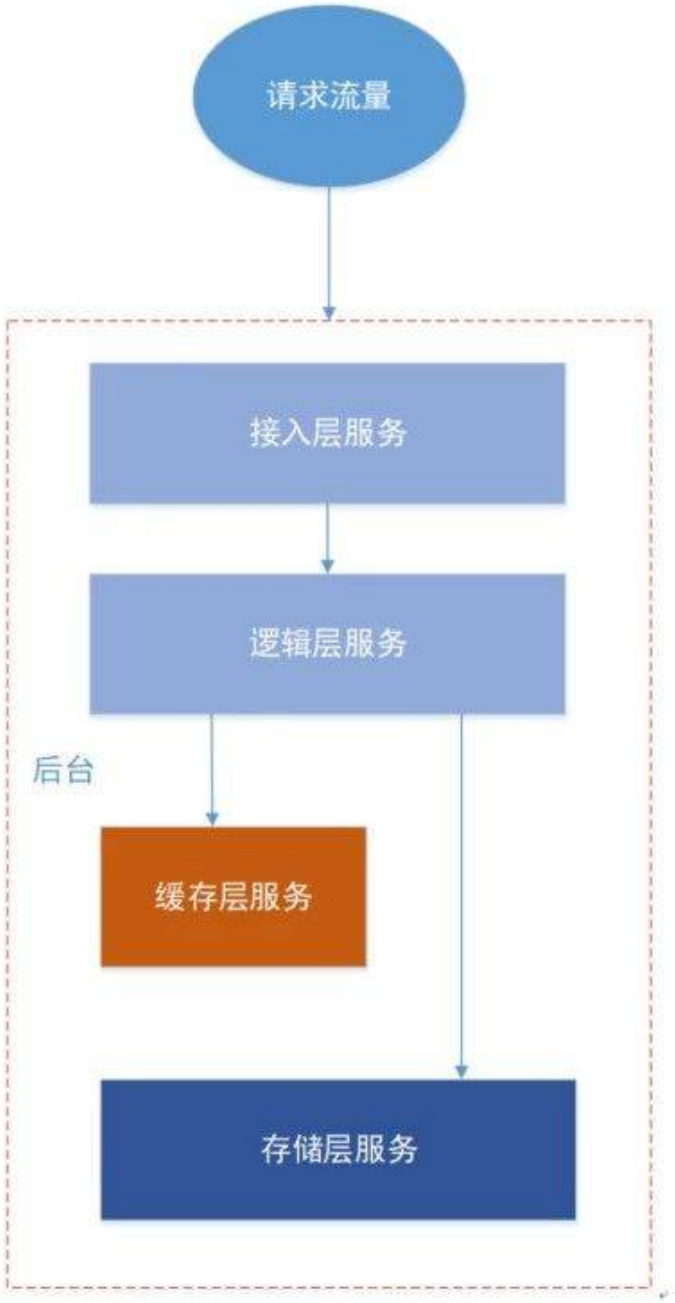
OPPO缓存层6次版本迭代的异地多活实践

前言

互联网后台服务的常用经典架构中，整个后台服务框架一般可以分四层：

- 接入层，一般用于请求的安全校验、频率和流量控制
- 逻辑层，用于用户数据的请求和返回逻辑控制
- 缓存层，为了加快访问性能，cache用户数据，常见的redis、memcache等
- 存储层，落地了用户各种数据，比如mysql、leveldb等



关于异地多活考虑这样一个常见问题，假设某当地机房的服务全部宕机（机房掉电或光纤挖断）了，这个时候怎么挽救？比如上海登陆服务宕机了，总不能让上海的用户无法登陆。很显然我们要把后台的请求流量切换到别的机房，当然后台的4层架构每一层的切换场景都不太一样。

缓存层的作用是为了加速访问性能，减轻对存储的访问压力。如果直接把流量切换到别的机房可能会由于缓存命中率太低，把处于底层的存储层打爆，严重影响机房的服务。如何解决这些问题，这个就是本篇文章要讨论的话题。

本篇主要围绕数据的一致性和数据的安全性等2个主要挑战，通过版本迭代的方式来阐述缓存层的多机房异地多活会遇到的一些问题。

1）一致性

在多机房中，机房之间的网络延时不确定性大；各个机房物理条件可能也不一样；多机房服务运行的状态也会不一样。如何保障在各种异常和不确定性的情况下还能保障数据的一致性是本篇重点要讨论的话题。

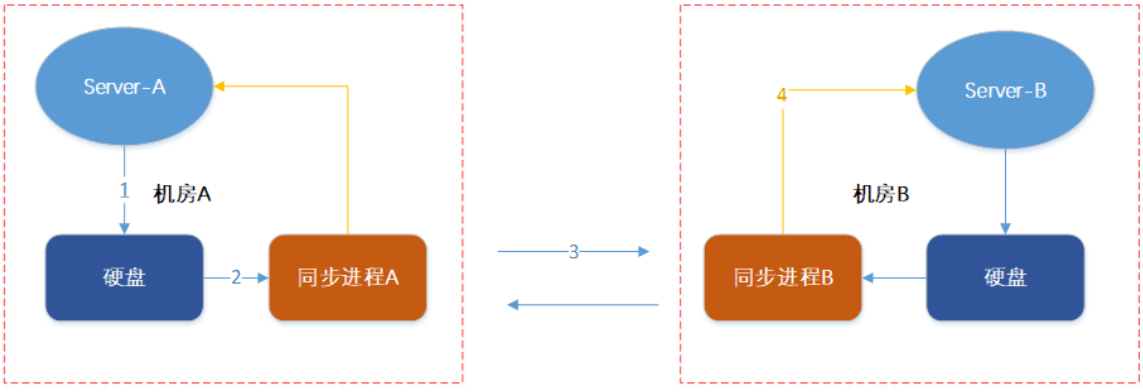
2）安全性

Google在GFS的论文开篇也说过类似的前提：“服务异常或者硬件故障有可能是常态”， 如何保障已经写入的数据在异常的常态下还能恢复也是本篇的另外一个重点。

起源版本

基于本篇要讨论的是缓存层异地多活问题，而redis目前比较常用，所以我们这里以redis作为缓存层的例子来说明问题。

内存数据是个断电易失的，我们很容易想到的第一个版本：**我们先把写入数据（修改和删除统称为写入）写到磁盘，然后用另外一个进程读取磁盘数据发送到其它机房**，如下图：



如此，我们的第一个起源版本就做好了， 要在生产环境运行还需要解决几个问题。

- 第一个问题 “同步进程”有可能被重启(代码bug或者其它多方面的原因)，重启后 “同步进程”进程从哪里开始重新同步呢？
- 第二个问题是数据可能不完整，比如像redis会有aof rewrite，删除原来的aof生成新的aof文件，老的aof文件会被删除而只保留新的aof，这样老aof的offset在新aof文件就无效了。

版本一

针对上面起源版本问题我们做一些解决办法。

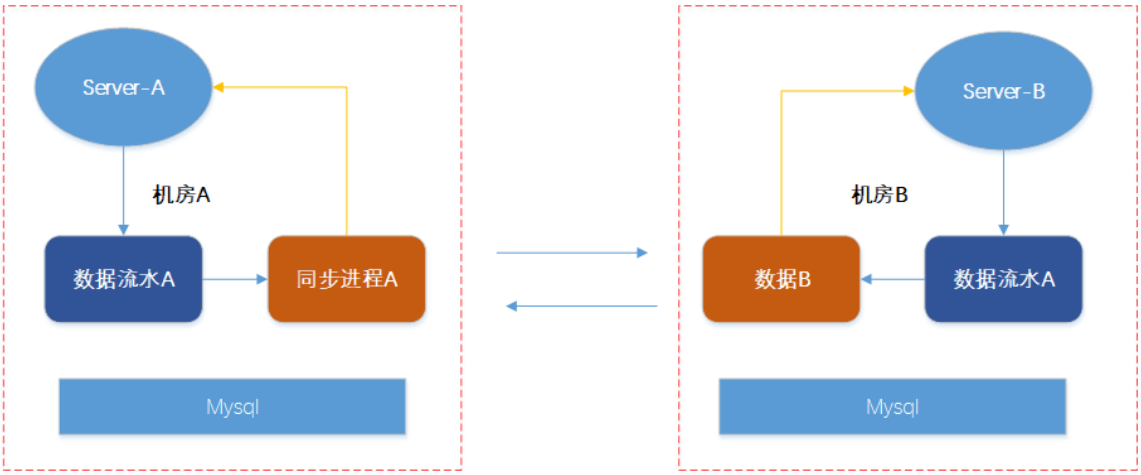
可靠的DB

我们引入一个DB，把同步进程每次同步的位置写入到DB里面去，同步进程重启的时候首先从DB找到自己的offset，然后从offset后面开始数据同步就可以了。

流水记录

参考mysql的方式，把所有的写入数据按照时间的先后用流水日志的方式记录下来不能被修改，在没有同步到别的机房前也不能被删除。

经过上面的修改后我们得到修改版本一。



把修改版本一放到生产环境跑了几天后，又会发现一些问题：

性能低下

同步进程每次同步一条数据都要修改Mysql同步offset，导致同步进程性能低下

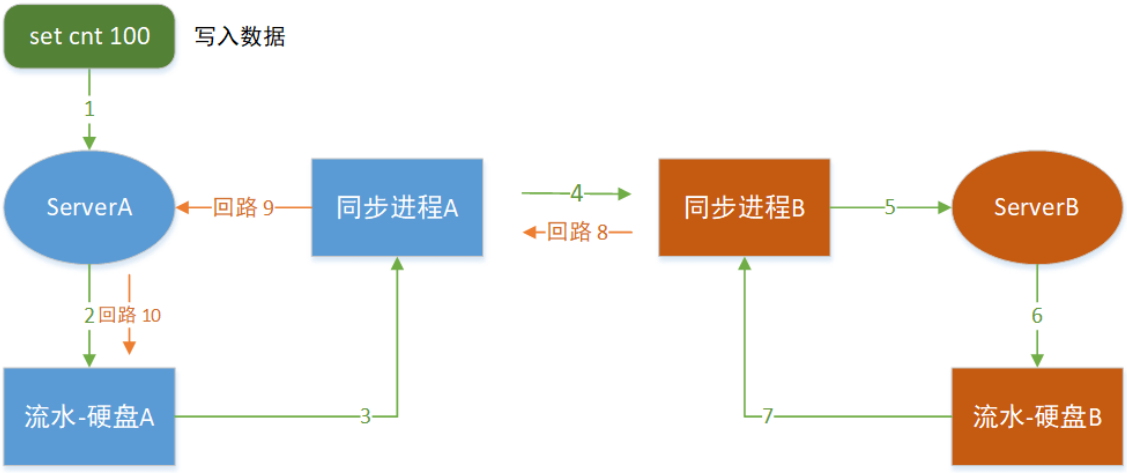
单机磁盘空间不够

- 因为物理机器上会混部多个服务，每个服务都在往磁盘写入数据
- 跨机房同步本身又比较慢

可能导致单机磁盘空间被打爆，磁盘被打爆后数据再也无法写入了。

数据回环

Server A的数据被同步进程A同步到B机房以后， 被写入到数据流水B，然后同步进程B又把这个流水同步给了A机房，就这样形成了一个循环回路浪费了网络带宽和资源。



如上图，A机房写入的数据同步到B机房后又回流到了A机房。

版本二

1、解决流水问题

针对版本一的2个问题，我们做一些方法来解决。

同步offset定时写入DB

同步进程每次同步了一跳数据后，不会立马把offset写入到mysql，而是一个时间段写一次。很显然性能问题解决了，但是同样带来了问题，如果同步进程重启了之后较大概率拿到的不是一个实时的offset。这样就会重复执行一部分流水日志。

幂等流水

为了达到重放了一部分日志也不影响数据的准确性。我们把所有的写入操作转换成内存镜像操作，比如string数据结构内存中cnt=100，那么执行inc cnt的可以转成set inc 101 这种操作。

inc cnt	set inc 101

其它的数据结构如果hash、set、zset都可以转换成这种操作，但是对于list队列这种数据就没办法支持类似的幂等操作了，这时候可以先不用管。

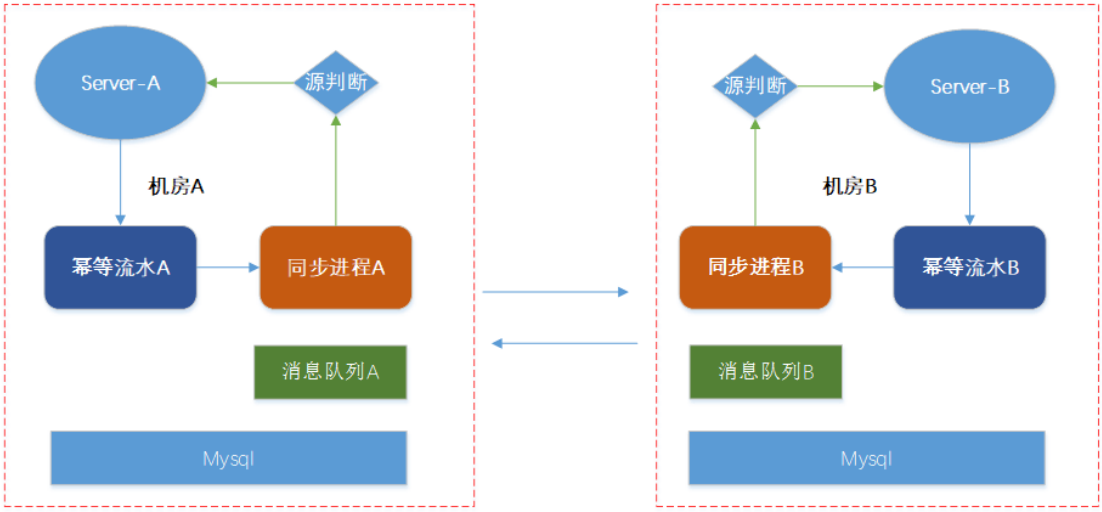
加入消息队列

因为单机磁盘毕竟有限很容写满，而且如出现磁盘损坏了数据还不能恢复。使用如kafka或者RocketMQ用多台机器做集群做数据的冗余保证了容量空间和数据安全。

流水格式加入idcid

如果同步进程可以识别同步过来的数据源是来自哪里，那么就可以解决数据回环问题了，比如同步进程A发现从B同步的数据源是自己，那么A就可以过滤掉这条数据。

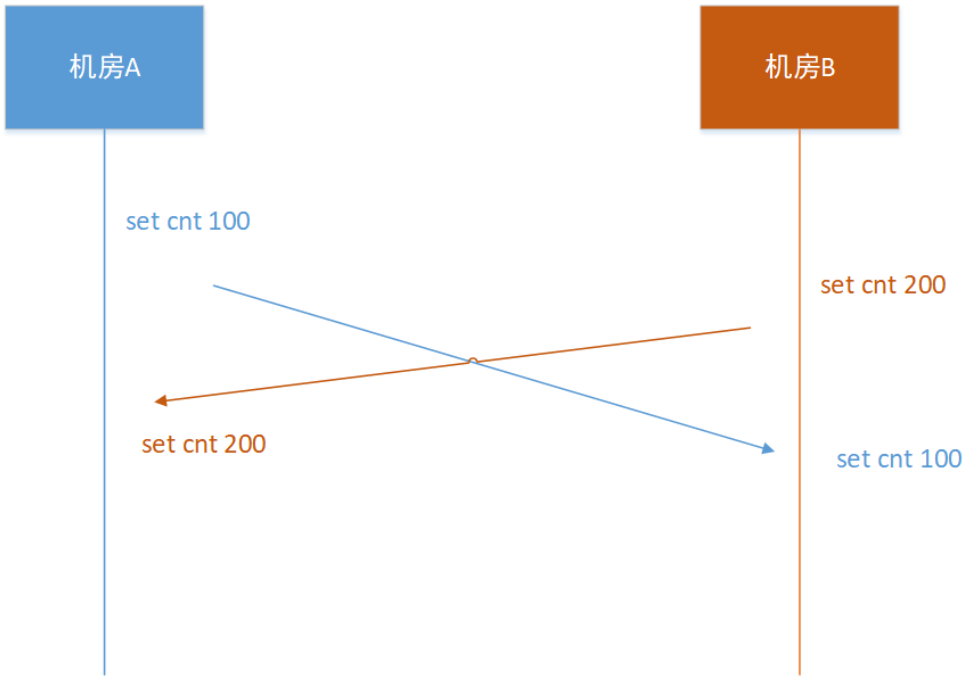
2、版本二的架构



经过了上面的几个版本迭代后，整个系统在生产环境跑起来了，但是经常一段时间后，很快你就会发现又出了很多新的问题：

机房A和机房B的数据出现了大量的不一致

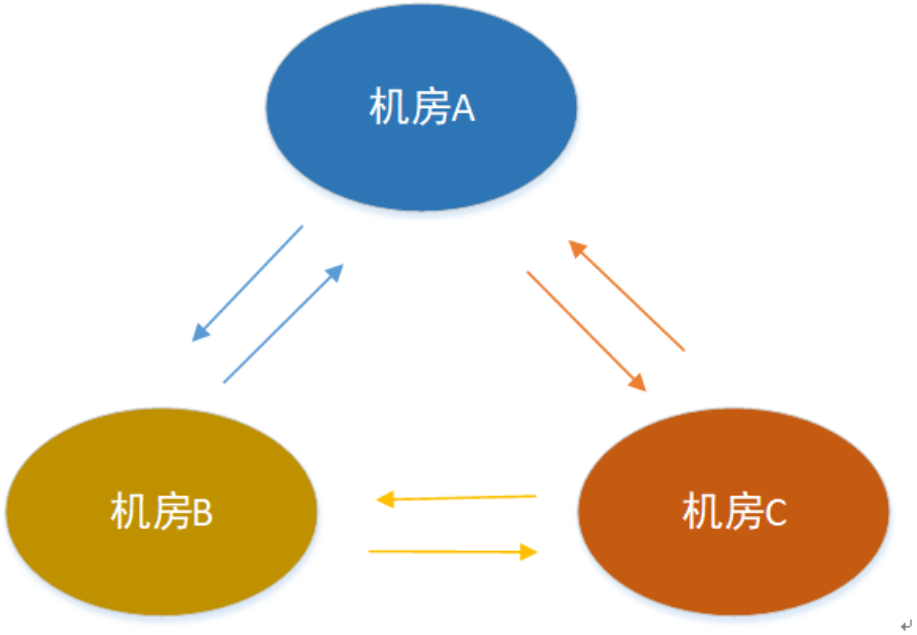
如果多机房写入一个同一个key时，就很容易出现数据不一致。



如上图，机房A和机房B同时设置了 cnt，因为机房之间同步的时间不确定导致了机房A和机房B的数据完全不一样，这个问题如果没有后续的写入，基本不可修复。

更多机房数

不是所有的业务仅仅是2个机房相互同步，可能需要三个机房或者更多机房数，单纯的相互收发数据已经不能用了，需要改动整体架构。

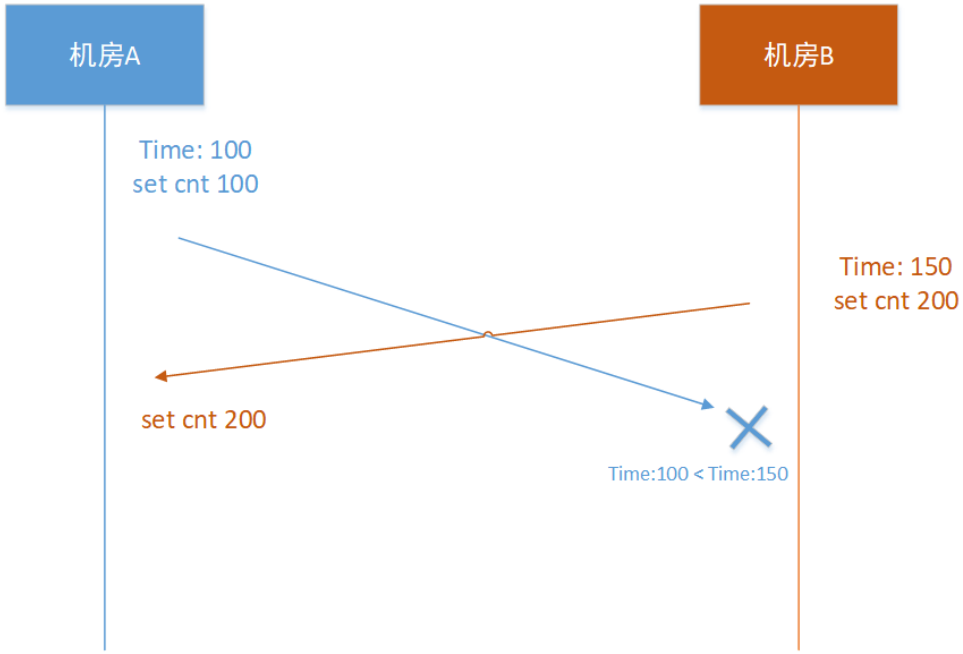


版本三

继续针对版本二的2个问题我们逐一讨论解决。

1、解决数据不一致

上面的第一个问题出现数据不一致的本质原因，在于在写入数据到cache时没有做统一的版本控制，如果我们能对数据做版本控制，不能随意写入，这样就可以达到数据的最终一致性了。作为多机房的数据版本控制用时间戳是我们很快的想到一个版本方法，如果按照写入绝对时间顺序为准似乎可以解决，如下图：



- 我们以时间戳为版本，时间戳越大版本越大，优先级越高，优先级高的能覆盖小优先级别的版本数据。
- 机房A的写入时间戳是100，而机房B的时间戳为150，从机房A同步到机房B的数据是没法执行成功的，因为机房B拥有更高优先级别的数据；而机房B同步到机房A的数据是可以同步成功的，从而达到了数据一致。
- 这里有个问题如果时间戳一致，怎么解决呢？优先级都是一样的，2个人各执己见似乎又会不一致，这里可以人工设置一个优先级放到配置文件里面。比如机房A高于机房B的优先级，如果出现版本优先级相同的情况下，以机房A为准。

这样，通过上面的简单的版本我们暂认为解决了数据不一致的问题。

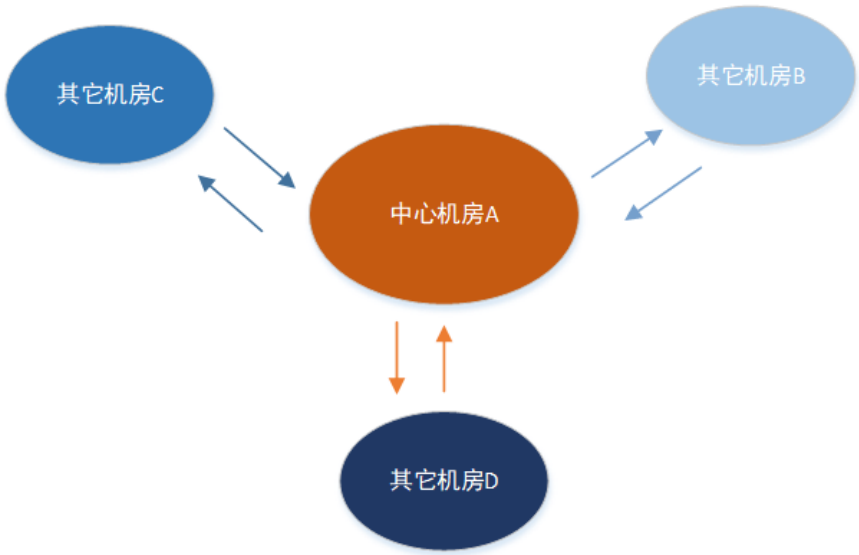
2、多机房支持

上面讨论的都是在2个机房相互同步的情况下，还比较简单，但如果是三个机房或者更多机房的情况下，就已经不能用了。面对多机房的数据要做同步这个时候可以有2个选择。

- 中心型：数据先同步到一个中心机房，然后由中心机房同步到其它的机房
- 两两相互型：机房之间数据相互同步，任意2个机房建立数据通道

针对上面的2个方式，我们各自讨论优缺点

中心型



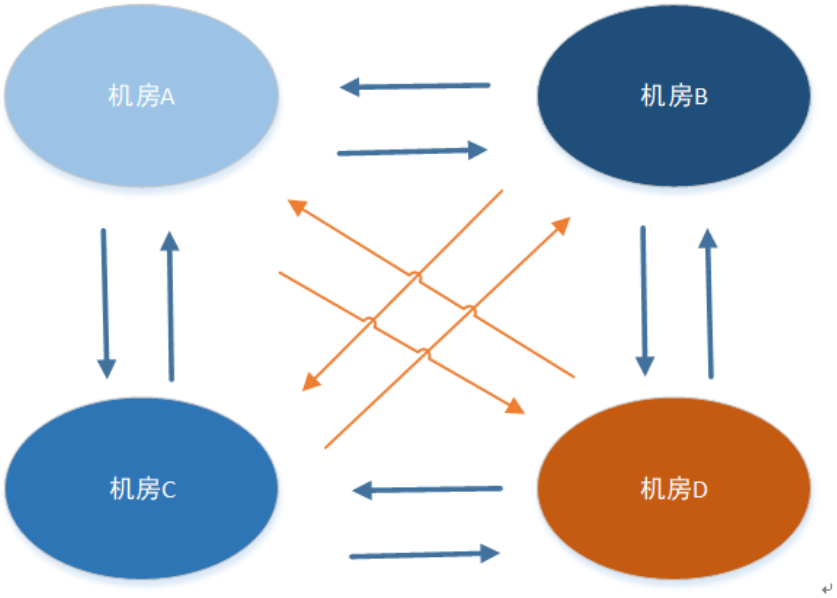
优点

- 架构简单：所有的机房数据只需要同步到中心机房即可
- 解决数据一致性：上面有说到，需要设置一个优先级的机房到配置文件，如果有中心机房这样一个角色存在，那么都以中心机房为准，就可以顺带解决一致性问题里面的如果版本号一致以谁为准的问题

缺点

- 中心机房的流量大：因为所有的流量都经过中心机房中转，所以流量会比较大
- Failover：当中心机房出现故障后，整体的同步通道都将被关闭

两两相互型



优点

- 去中心化：任何一个机房都是对等的，任何一个机房去掉，都不影响其它机房的数据同步，其它机房都可以照常同步，流量上也是比较均等

缺点

- 过于复杂：从上面的4个机房相互同步可以看到，建立的通道数据太多，架构层面也要同时考虑与每一个机房的异常情况，会导致整体会非常复杂
- 实际当中如果同步的机房个数不会超过3个，上面的2个类型都可以用，需要解决每个类型面临的问题就可以。

这里我们选择中心型来做多机房的数据同步，接下来要解决的就是中心型问题。

中心型的需要解决的几个问题

1）流量大

因为我们的流量是写入流量，也就是说读取的数据是没有流水的，对于cache来说一般写入的量不会很大，写入流量比较大的场景可能就是在导入数据的时候那个短暂的瞬间。

2）Failover

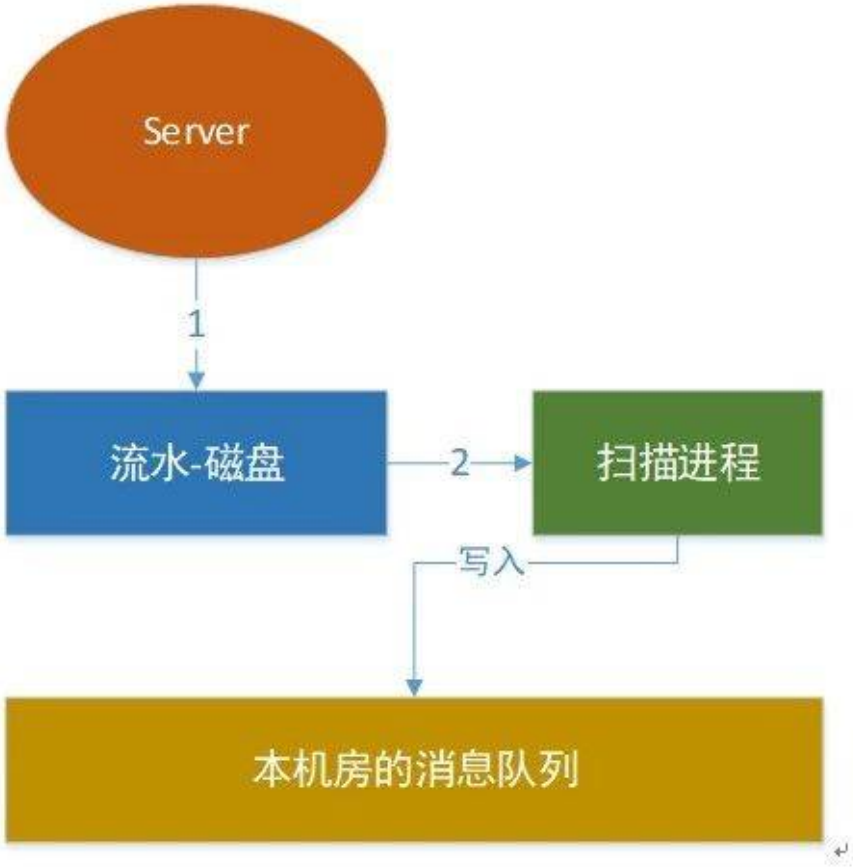
中心机房如果出现故障，或者中心机房网络成为孤岛，要做到：

① 不影响其他redis机房的正常写入和读取。

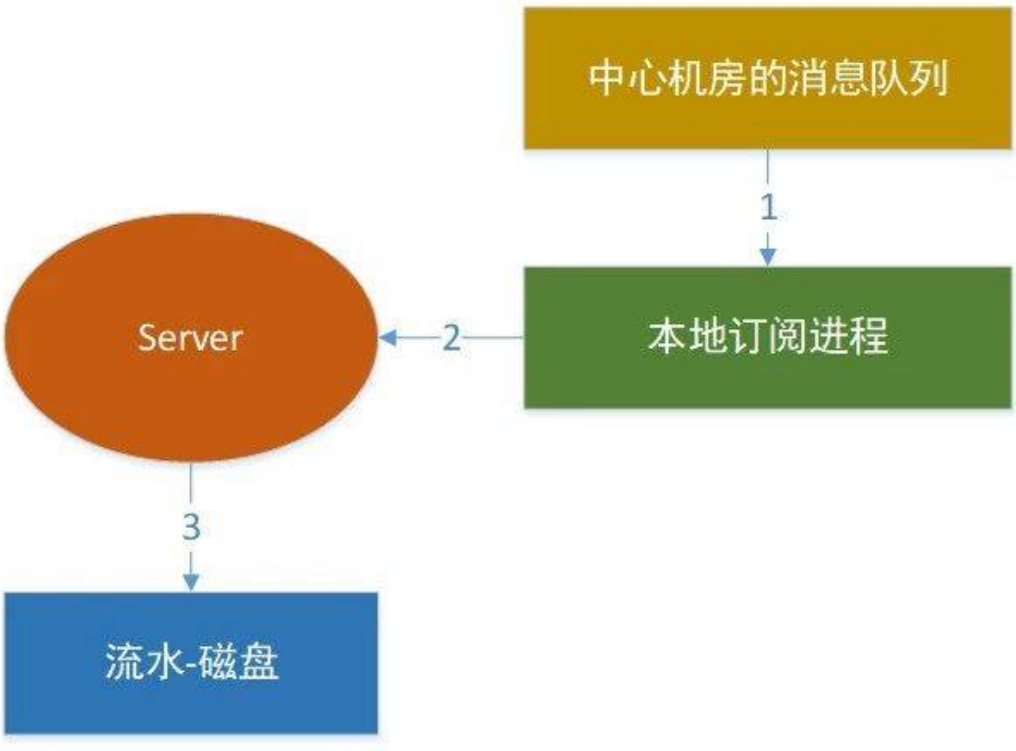
根据消息队列的订阅和写入，我们把同步进程拆分开2个不同的单独的第三方进程，“订阅进程”和“同步扫描进程”。

redis本身还是不参与同步，由这2个第三方进程来做同步。

- 同步扫描进程：只负责扫描本地的server binlog，然后写入到本地消息队列，结束返回。



- 订阅进程：中心机房要订阅所有其它非中心机房的消息队列，汇总数据；非中心机房只要订阅中心机房的消息队列就可以拿到全量数据了。

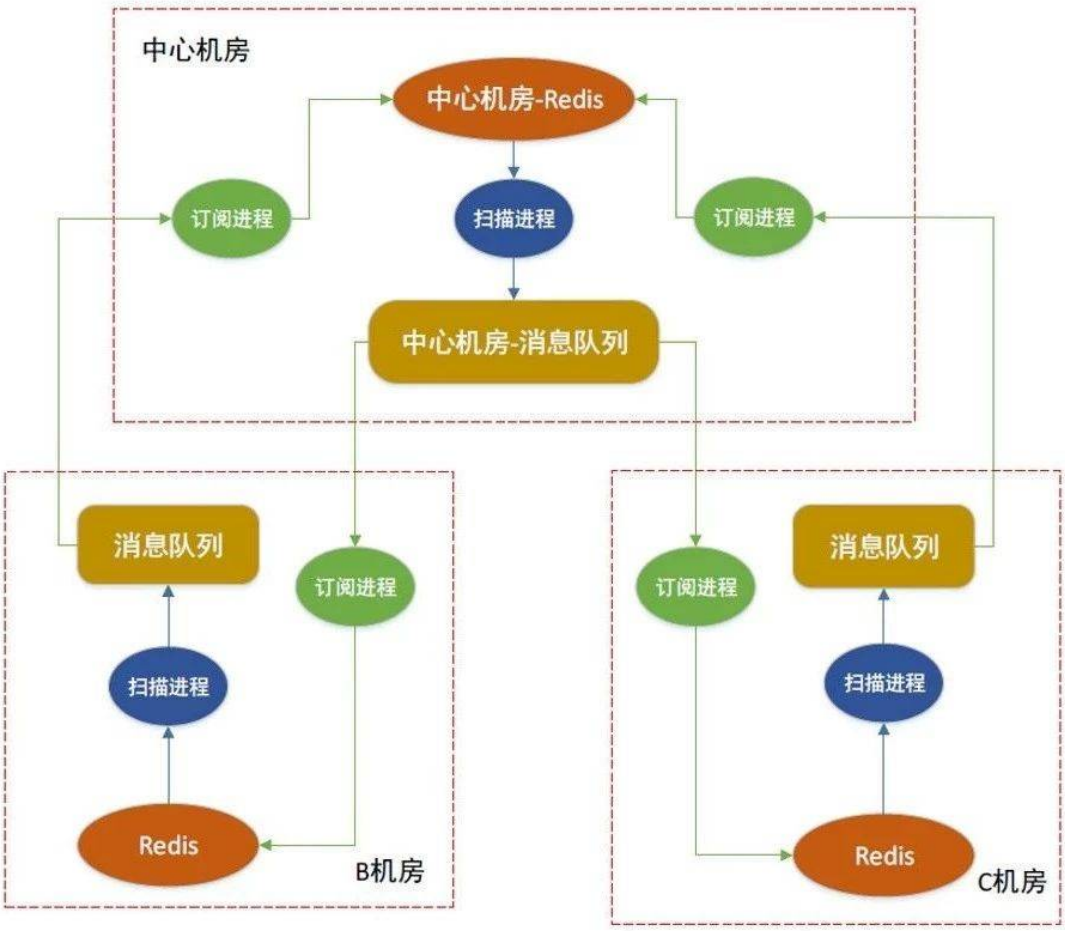


② 快速设置新的中心机房

当中心机房可能整个成为孤岛无法和别的机房通信后，运维可以在DB设置一个新的中心机房idcid。订阅进程和扫描进程如果发现DB发生了变动会立马同步新的中心机房的idcid到所有的同步组件。

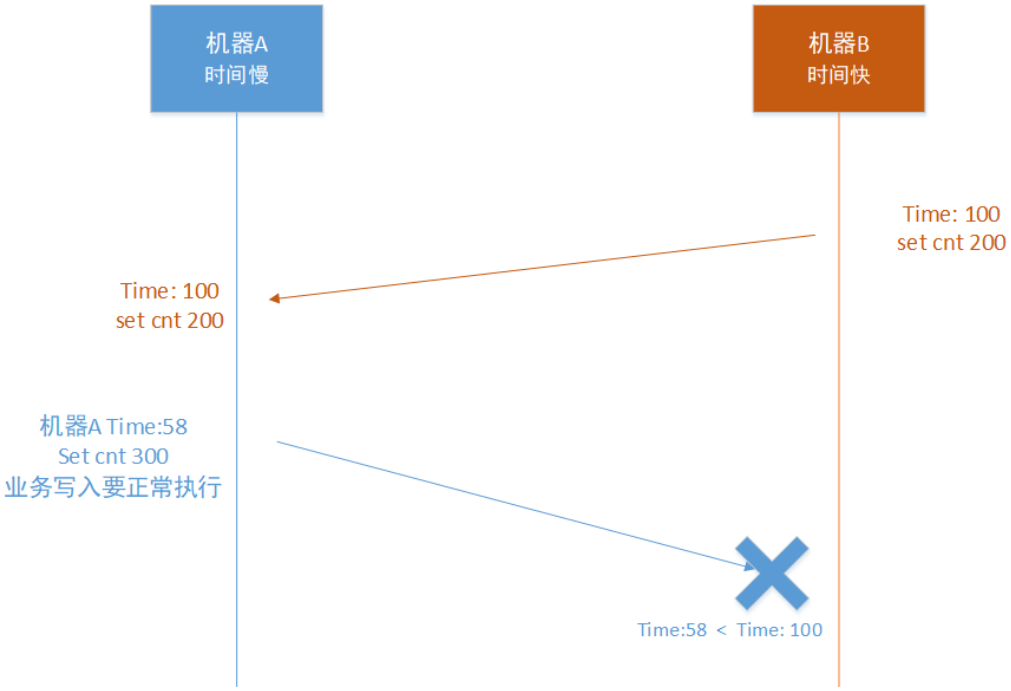
3、版本三的架构

解决了上面的2个问题后，我们的版本架构以三个机房为例子：



版本四

版本三看似已经解决了一些问题，可在生成环境跑了一段时间后，你就发现还是会出现了越来越多的数据不一致key，甚至差异较大。看下面的图：



机房B，执行 “set cnt 200”， 时间戳是100，然后同步到机房A。

机房A，执行 “set cnt 300”，机房A的时间戳可能异常是58，因为是业务的正常写入，假设先不能拒绝，那么在此时机房A的内存状态是 cnt = 300，时间是58，当把这个状态同步到机房B的时候就会失败，因为A机房的时间58小于B机房的100。这个时候在A、B机房就存在相同key但是不同value的情况。

1、继续解决数据不一致

如上图所示，机器A的时间戳因为比较小，导致机器A时间后写入的数据无法同步到机器B， 导致机房A的数据和机房B的无法达成一致。这个时候有2个选择：

1） 一定条件下拒绝业务的正常写入

内存中数据有一个版本号，如果业务的正常写入此时获取的版本号比内存中的版本号还要小，那么就拒绝业务的正常写入。

比如上图中的场景：A机房的时间戳比较小，小于此时cnt的版本号就拒绝写入。

这个办法是一个很简单的解决思路，但是如果**某个机器的时间比较大，会导致其他机器就无法写入数据，这显然是不可取的。**

2）把这个艰巨的任务交给运维或者运营

时刻监控每一个机器的时间戳，如果出现了不一致就告警和自动修复，这个时候你的手机可能无时无刻的都在告警。

还会有另外一个问题？ 因为多机房之间网络通信本身就需要较长时间，等服务去取到机器的时间戳本身就耗费较长时间，多个机房的时间戳也要保住一致也是不太现实的。

经过无数次的折腾，最后你只能面对现实：**要保证所有机房的时间戳一致是不可能的。**

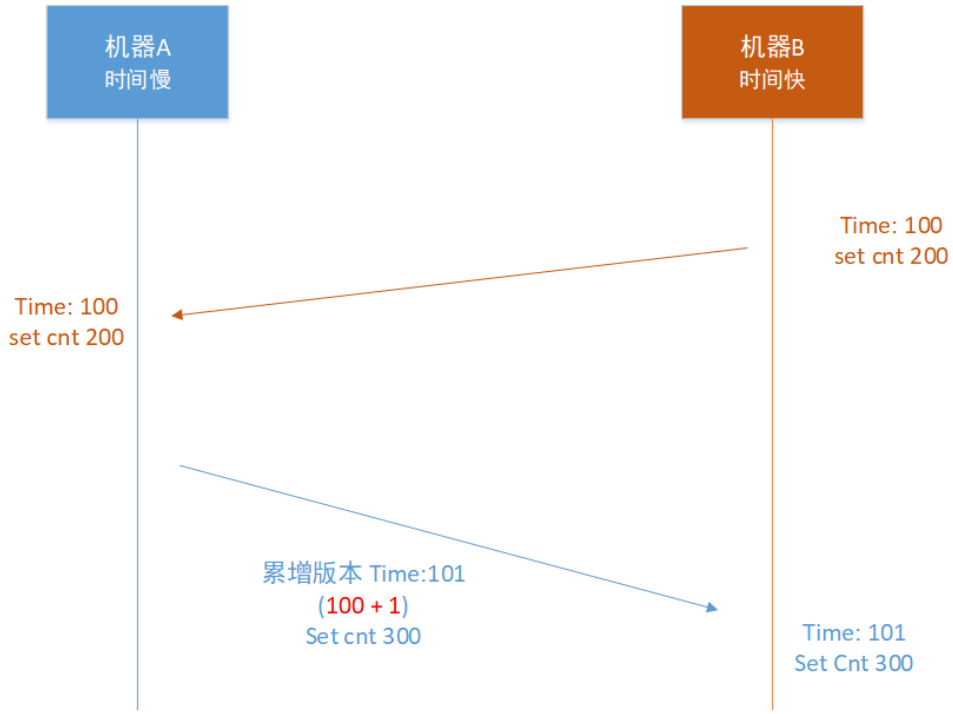
2、分布式的逻辑时钟

回到问题本身，我们用时间戳是用来作为key版本号的作用保证数据的一致性，假设就算时间戳不一致也能达到数据的的最终一致性，那就不需要纠结“保证所有机器的时间戳一致”这个问题了！我们解决思路简单来说是：

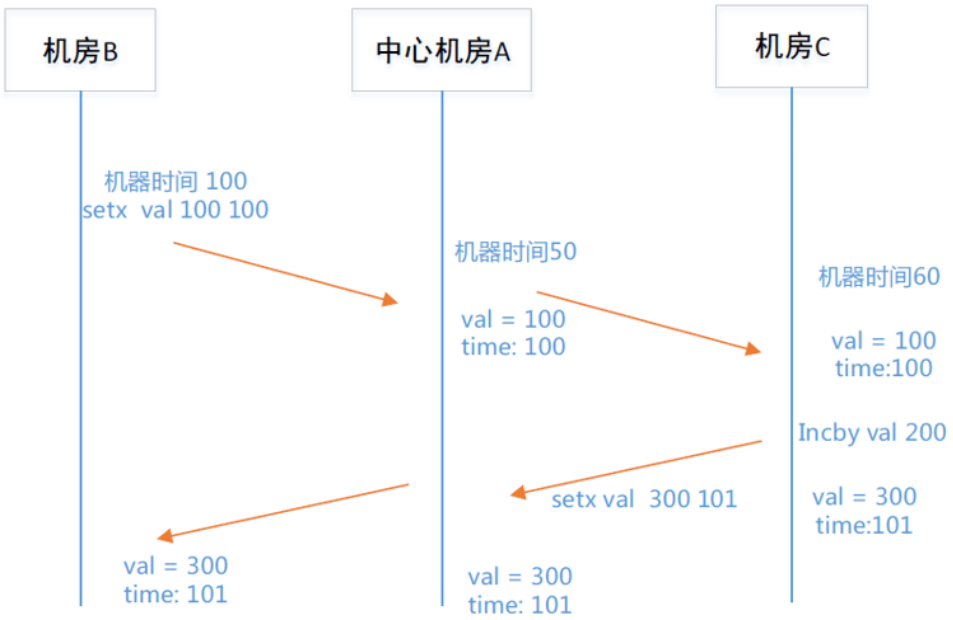
时钟不可逆, 时钟版本号只能递增！

每一个key在写入时的时间戳版本都不能变小，只能变大。我们key的版本号不在是绝对的物理机器时间戳，而是一个逻辑时间钟，这个时间钟不能变小。

看看上面的问题在机器A设置数据“set cnt 300”时，因为本机A的机器时间比较慢获取到的时间戳是58，但是cnt本身的时间戳是100，这样的话在机房A的写入操作版本号就变小了，肯定无法同步到机房B的，如果这个时候代码发现cnt 的版本号大于机器的时间戳，就把版本号进行累增到101，这个时候就可以同步到机房B了。



分布式的逻辑时钟解决多机房一致性。

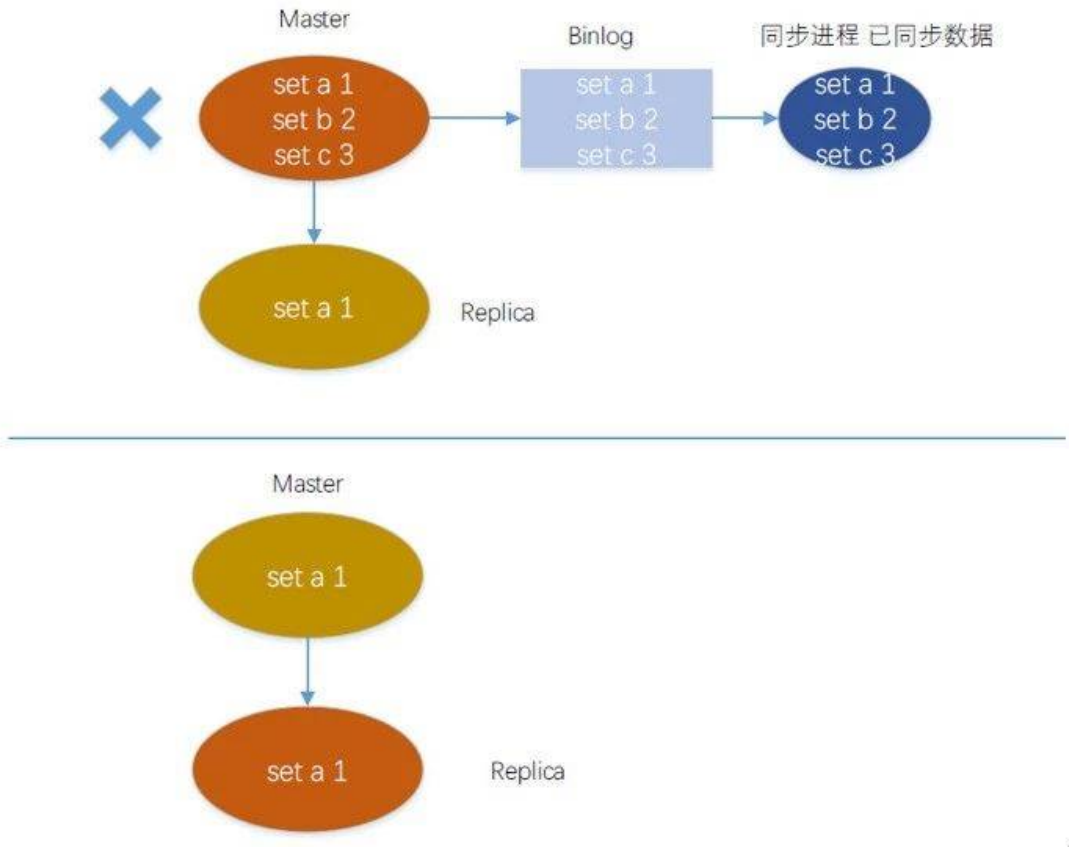


3、版本四最终形态

版本四和版本三的架构还是不变的，变动的是在于通过分布式逻辑时钟增这个方法，来做到不管多机房的机器时间戳如何不一致都可以做到最终一致性。

版本五

经过了前面多版本的迭代了，这个时候应该是没有什么问题了，在生产环境跑了好久也没出现啥问题。但是随着接入的业务越来越多，会发现又会偶尔出现数据不一致的情况，你不得已登陆到机器看到发现server发生主备切换了，对于数据的一致性你做的还不够。



Master写入了a、b、c三个数据，同步进程也已经同步到了别的机房，如这个时候master宕机了，但是b、c还没有同步到Replication备机，就发生了主备切换，Replication成为新的master，但是没有b、c这2个数据，而别的机房还存在这2个数据。

1、解决主备切换数据不一致

这个问题的根本原因在于主机和备机的数据不一致，但是同步进程却把不一致的数据同步到了别的机房，如果每一个数据都在replica都存在那同步到别的机房就不会有问题了。这个时候可以有2个解决思路：

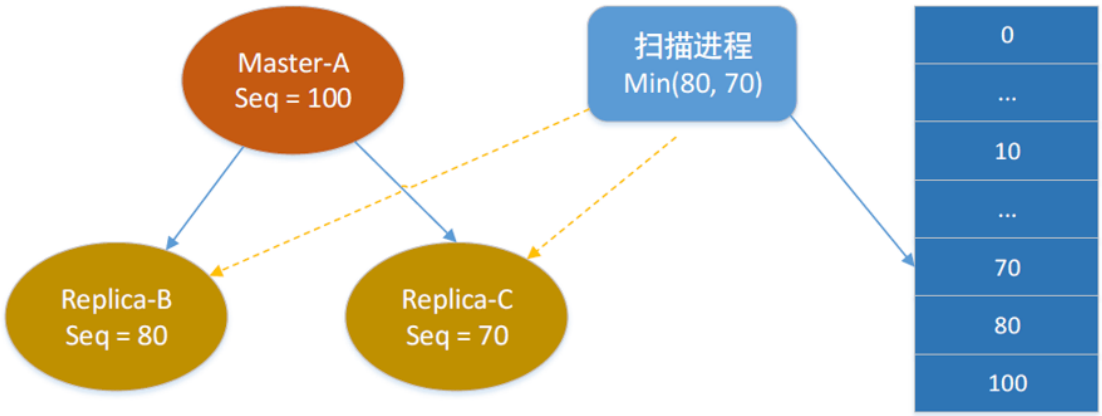
查询备机

扫描进程每次同步数据都去查询一下备机，如果数据一致就同步别的机房。显然这样做的成本太高，会导致同步非常慢，如果查询出现延迟，扫描同步进程很容易卡顿，导致吞吐量大大下降。

写入seq

用一个整型值来累计写入操作，每次写入操作加1，主备同时累增，同时binlog流水里面也加上这个seq，每一个写入流水对应一个seq。

比如master的seq是100，但是replica的seq是80，这个时候我们的扫描同步进程只要同步80之前的binlog流水同步到别的机房就没有问题，而80~100的binlog流水则先不同步。



2、修改版本五最终形态

版本五和版本四的架构还是不变的，变动的是在于通过加入seq这个机制，来保证server在做主备切换的时候也可以保证数据的一致性。

版本六

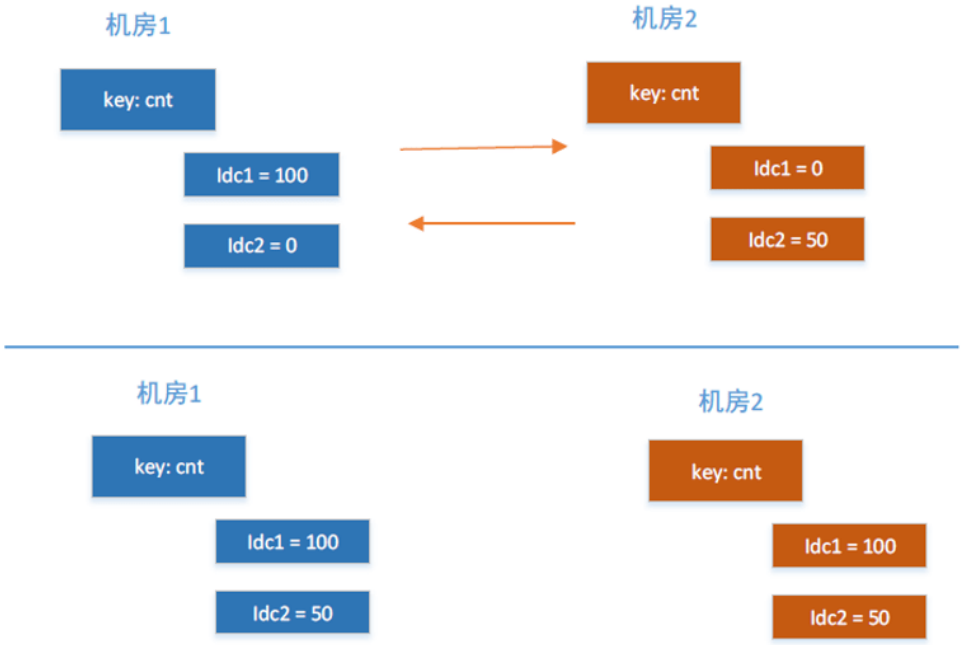
既然多个机房数据同步了，如果多个机房同时做累加操作的话，比如inc cnt这种操作，因为数据同步都是内存镜像操作。

比如cnt初始值为0， A机房inc cnt，B机房也inc cnt，通过上面的同步机制最终得出的值可能是1也可能是2，而用户需要的是2这个值。

1、新增多机房的数值统计功能

上面统计出现问题的原因在于多个机房对同一个key进行累增或者累减，然后同步的又是内存镜像就会导致数据相互覆盖。

所以我们的解决办法是把每个机房的数值进行单独统计，比如用一个hash结构，cnt为key名作为主key，每个机房的名字id做为子key。每个机房都单独累增或者累减相互不影响，这样读取的时候就可以得到一个正确的多方值了。



2、版本的形态

这个版本我们加入的全球多机房的数值统计功能，整体架构不变。

RedisPlus

上面的所有功能，我们的中间件RedisPlus都已经实现好了，即时可用。

参考文献：

https://www.sohu.com/a/403840395_411876

https://zhuanlan.zhihu.com/p/96917394