


ES+Redis+MySQL，这个高可用架构设计太顶了！

点击关注
 顶级架构师
 2023-08-08 19:31
 浙江

推荐关注



Java后端栈

Java后端栈，专注分享Java技术，专研多线程、JVM、Spring Boot、Spring Cloud、Int...
5篇原创内容

公众号

顶级架构师后台回复
 1024
 有特别礼包

来源：同程艺龙技术中心

上一篇：支付系统就该这么设计（万能通用），稳的一批！

大家好，我是顶级架构师。

背景

会员系统是一种基础系统，跟公司所有业务线的下单主流程密切相关。如果会员系统出故障，会导致用户无法下单，影响范围是全公司所有业务线。所以，会员系统必须保证高性能、高可用，提供稳定、高效的基础服务。

随着同程和艺龙两家公司的合并，越来越多的系统需要打通同程 APP、艺龙 APP、同程微信小程序、艺龙微信小程序等多平台会员体系。

例如微信小程序的交叉营销，用户买了一张火车票，此时想给他发酒店红包，这就需要查询该用户的统一会员关系。

因为火车票用的是同程会员体系，酒店用的是艺龙会员体系，只有查到对应的艺龙会员卡号后，才能将红包挂载到该会员账号。

除了上述讲的交叉营销，还有许多场景需要查询统一会员关系，例如订单中心、会员等级、里程、红包、常旅、实名，以及各类营销活动等等。

所以，会员系统的请求量越来越大，并发量越来越高，今年清明小长假的秒并发 tps 甚至超过 2 万多。

在如此大流量的冲击下，会员系统是如何做到高性能和高可用的呢？这就是本文着重讲述的内容。

ES 高可用方案

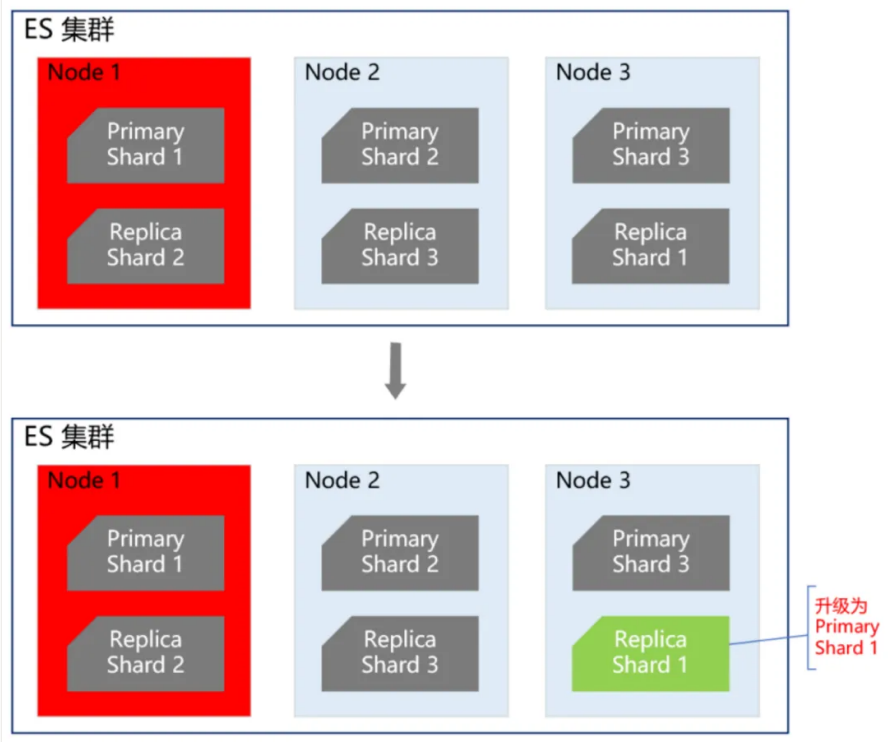
| ES 双中心主备集群架构

同程和艺龙两家公司融合后，全平台所有体系的会员总量是十多亿。在这么大的数据体量下，业务线的查询维度也比较复杂。

有的业务线基于手机号，有的基于微信 unionid，也有的基于艺龙卡号等查询会员信息。

这么大的数据量，又有这么多的查询维度，基于此，我们选择 ES 用来存储统一会员关系。ES 集群在整个会员系统架构中非常重要，那么如何保证 ES 的高可用呢？

首先我们知道，ES 集群本身就是保证高可用的，如下图所示：



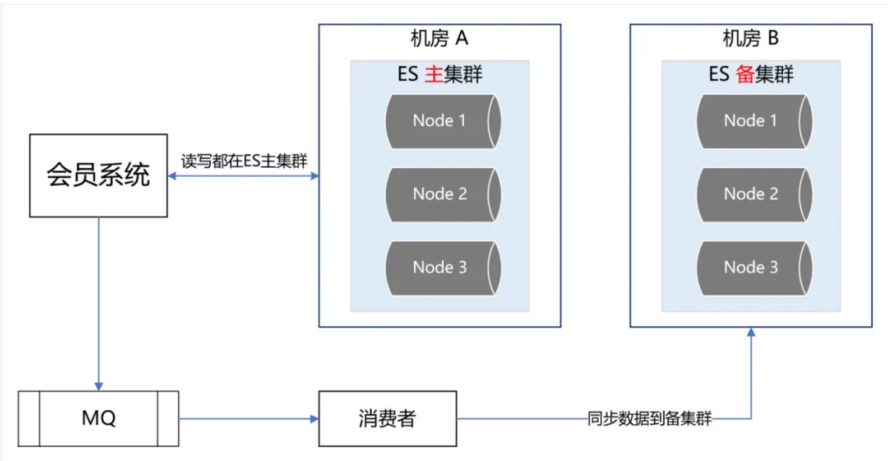
当 ES 集群有一个节点宕机了，会将其他节点对应的 Replica Shard 升级为 Primary Shard，继续提供服务。

但即使是这样，还远远不够。例如 ES 集群都部署在机房 A，现在机房 A 突然断电了，怎么办？

例如服务器硬件故障，ES 集群大部分机器宕机了，怎么办？或者突然有个非常热门的抢购秒杀活动，带来了一波非常大的流量，直接把 ES 集群打死了，怎么办？面对这些情况，让运维兄弟冲到机房去解决？

这个非常不现实，因为会员系统直接影响全公司所有业务线的下单主流程，故障恢复的时间必须非常短，如果需要运维兄弟人工介入，那这个时间就太长了，是绝对不能容忍的。

那 ES 的高可用如何做呢？我们的方案是 ES 双中心主备集群架构。



我们有两个机房，分别是机房 A 和机房 B。我们把 ES 主集群部署在机房 A，把 ES 备集群部署在机房 B。会员系统的读写都在 ES 主集群，通过 MQ 将数据同步到 ES 备集群。

此时，如果 ES 主集群崩了，通过统一配置，将会会员系统的读写切到机房 B 的 ES 备集群上，这样即使 ES 主集群挂了，也能在很短的时间内实现故障转移，确保会员系统的稳定运行。

最后，等 ES 主集群故障恢复后，打开开关，将故障期间的数据同步到 ES 主集群，等数据同步一致后，再将会会员系统的读写切到 ES 主集群。

| ES 流量隔离三集群架构

双中心 ES 主备集群做到这一步，感觉应该没啥大问题了，但去年的一次恐怖流量冲击让我们改变了想法。

那是一个节假日，某个业务上线了一个营销活动，在用户的一次请求中，循环 10 多次调用了会员系统，导致会员系统的 tps 暴涨，差点把 ES 集群打爆。

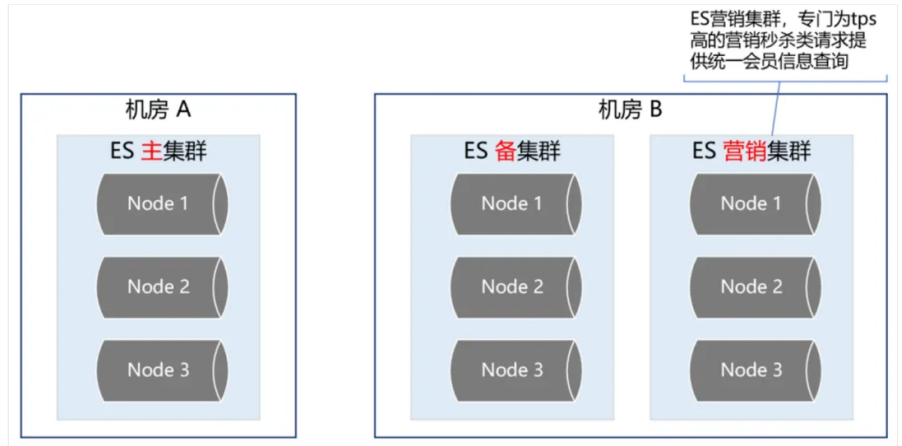
这件事让我们后怕不已，它让我们意识到，一定要对调用方进行优先级分类，实施更精细的隔离、熔断、降级、限流策略。

首先，我们梳理了所有调用方，分出两大类请求类型：

- 第一类是跟用户的下单主流程密切相关的请求，这类请求非常重要，应该高优先级保障。
- 第二类是营销活动相关的，这类请求有个特点，他们的请求量很大，tps 很高，但不影响下单主流程。

基于此，我们又构建了一个 ES 集群，专门用来应对高 tps 的营销秒杀类请求，这样就跟 ES 主集群隔离开来，不会因为某个营销活动的流量冲击而影响用户的下单主流程。

如下图所示：



| ES 集群深度优化提升

讲完了 ES 的双中心主备集群高可用架构，接下来我们深入讲解一下 ES 主集群的优化工作。

有一段时间，我们特别痛苦，就是每到饭点，ES 集群就开始报警，搞得每次吃饭都心慌慌的，生怕 ES 集群一个扛不住，就全公司炸锅了。

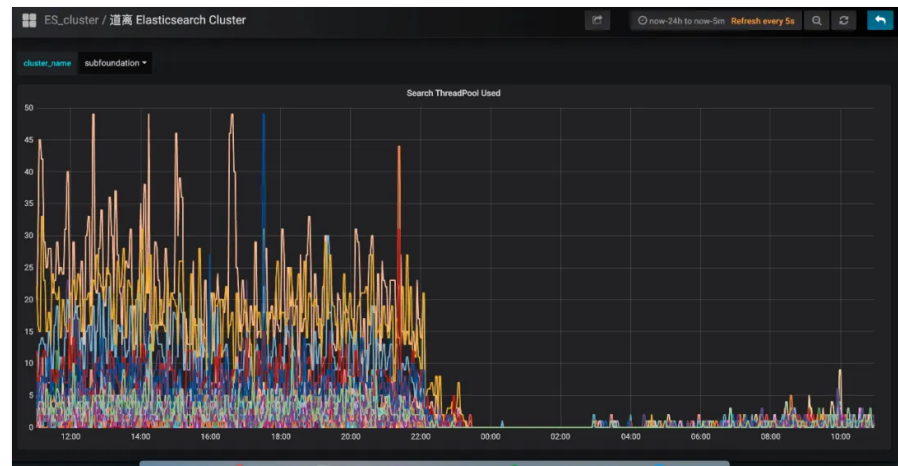
那为什么一到饭点就报警呢？因为流量比较大，导致 ES 线程数飙升，cpu 直往上窜，查询耗时增加，并传导给所有调用方，导致更大范围的延时。那么如何解决这个问题呢？

通过深入 ES 集群，我们发现了以下几个问题：

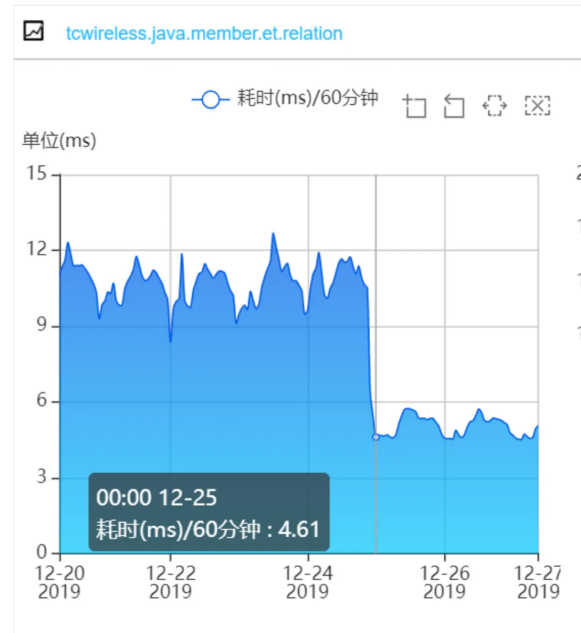
- **ES 负载不合理，热点问题严重。** ES 主集群一共有几十个节点，有的节点上部署的 shard 数偏多，有的节点部署的 shard 数很少，导致某些服务器的负载很高，每到流量高峰期，就经常预警。
- **ES 线程池的大小设置得太高，导致 cpu 飙升。** 我们知道，设置 ES 的 threadpool，一般将线程数设置为服务器的 cpu 核数，即使 ES 的查询压力很大，需要增加线程数，那最好也不要超过“cpu core * 3 / 2 + 1”。如果设置的线程数过多，会导致 cpu 在多个线程上下文之间频繁来回切换，浪费大量 cpu 资源。
- **shard 分配的内存太大，100g，导致查询变慢。** 我们知道，ES 的索引要合理分配 shard 数，要控制一个 shard 的内存大小在 50g 以内。如果一个 shard 分配的内存过大，会导致查询变慢，耗时增加，严重拖累性能。
- **string 类型的字段设置了双字段，既是 text，又是 keyword，导致存储容量增大了一倍。** 会员信息的查询不需要关联度打分，直接根据 keyword 查询就行，所以完全可以将 text 字段去掉，这样就能节省很大一部分存储空间，提升性能。
- **ES 查询，使用 filter，不使用 query。** 因为 query 会对搜索结果进行相关性算分，比较耗 cpu，而会员信息的查询是不需要算分的，这部分的性能损耗完全可以避免。
- **节约 ES 算力，** 将 ES 的搜索结果排序放在会员系统的 jvm 内存中进行。

- **增加 routing key。**我们知道，一次 ES 查询，会将请求分发给所有 shard，等所有shard返回结果后再聚合数据，最后将结果返回给调用方。如果我们事先已经知道数据分布在哪些 shard 上，那么就可以减少大量不必要的请求，提升查询性能。

经过以上优化，成果非常显著，ES 集群的 cpu 大幅下降，查询性能大幅提升。ES 集群的 cpu 使用率：



会员系统的接口耗时：



会员 Redis 缓存方案

一直以来，会员系统是不做缓存的，原因主要有两个：

- 第一个，前面讲的 ES 集群性能很好，秒并发 3 万多，99 线耗时 5 毫秒左右，已经足够应付各种棘手的场景。
- 第二个，有的业务对会员的绑定关系要求实时一致，而会员是一个发展了 10 多年的老系统，是一个由好多接口、好多系统组成的分布式系统。

所以，只要有一个接口没有考虑到位，没有及时去更新缓存，就会导致脏数据，进而引发一系列的问题。

例如：用户在 APP 上看不到微信订单、APP 和微信的会员等级、里程等没合并、微信和 APP 无法交叉营销等等。

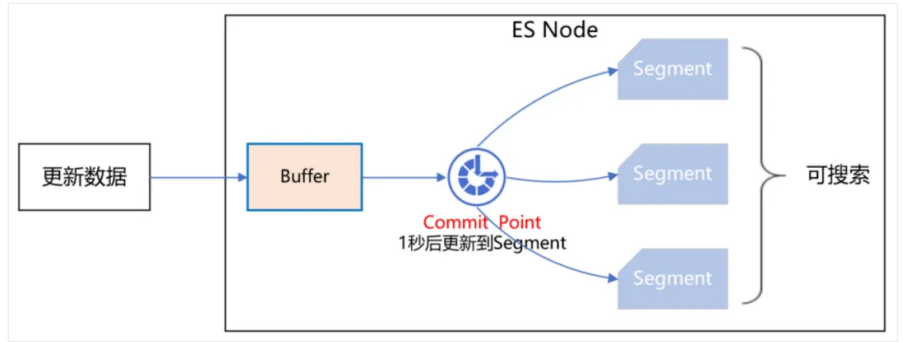
那后来为什么又要做缓存呢？是因为今年机票的盲盒活动，它带来的瞬时并发太高了。虽然会员系统安然无恙，但还是有点心有余悸，稳妥起见，最终还是决定实施缓存方案。

| ES 近一秒延时导致的 Redis 缓存数据不一致问题的解决方案

在做会员缓存方案的过程中，遇到一个 ES 引发的问题，该问题会导致缓存数据的不一致。

我们知道，ES 操作数据是近实时的，往 ES 新增一个 Document，此时立即去查，是查不到的，需要等待 1 秒后才能查询到。

如下图所示：

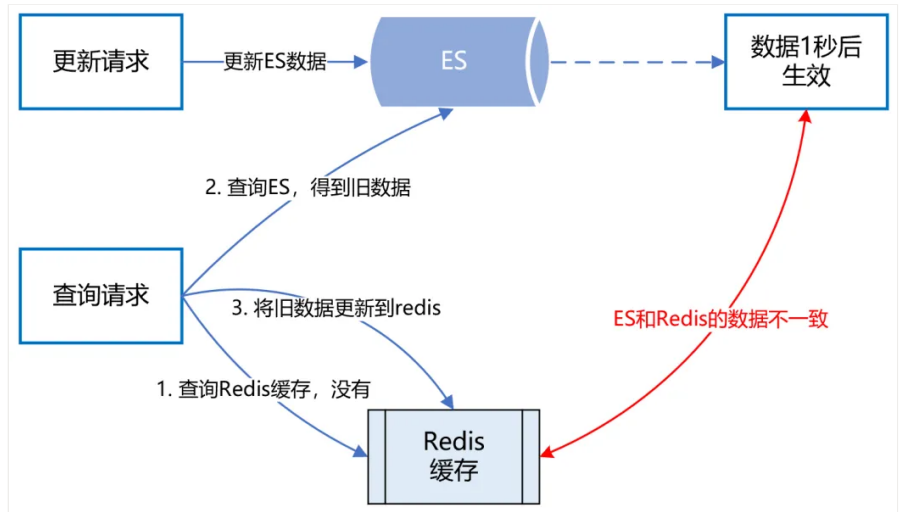


ES 的近实时机制为什么会导致 Redis 缓存数据不一致呢？具体来讲，假设一个用户注销了自己的 APP 账号，此时需要更新 ES，删除 APP 账号和微信账号的绑定关系。而 ES 的数据更新是近实时的，也就是说，1 秒后你才能查询到更新后的数据。

而就在这 1 秒内，有个请求来查询该用户的会员绑定关系，它先到 Redis 缓存中查，发现没有，然后到 ES 查，查到了，但查到的是更新前的旧数据。

最后，该请求把查询到的旧数据更新到 Redis 缓存并返回。就这样，1 秒后，ES 中该用户的会员数据更新了，但 Redis 缓存的数据还是旧数据，导致了 Redis 缓存跟 ES 的数据不一致。

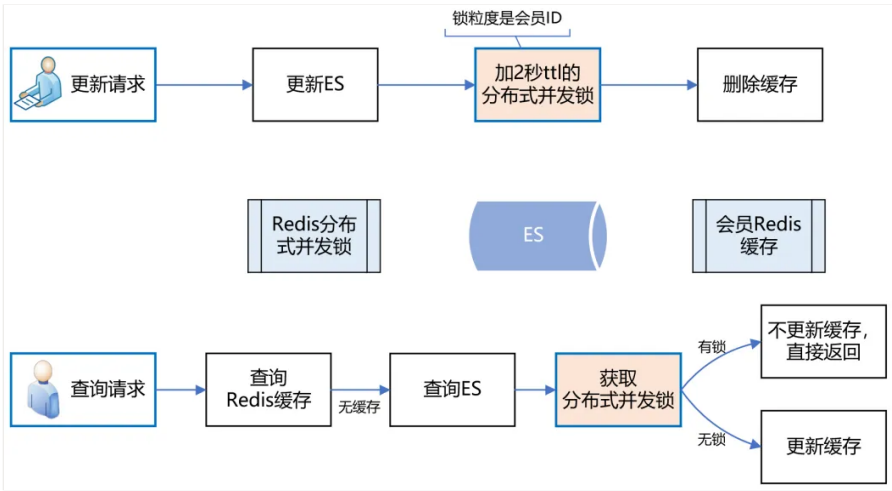
如下图所示：



面对该问题，如何解决呢？我们的思路是，在更新 ES 数据时，加一个 2 秒的 Redis 分布式并发锁，为了保证缓存数据的一致性，接着再删除 Redis 中该会员的缓存数据。

如果此时有请求来查询数据，先获取分布式锁，发现该会员 ID 已经上锁了，说明 ES 刚刚更新的数据尚未生效，那么此时查询完数据后就不更新 Redis 缓存了，直接返回，这样就避免了缓存数据的不一致问题。

如下图所示：



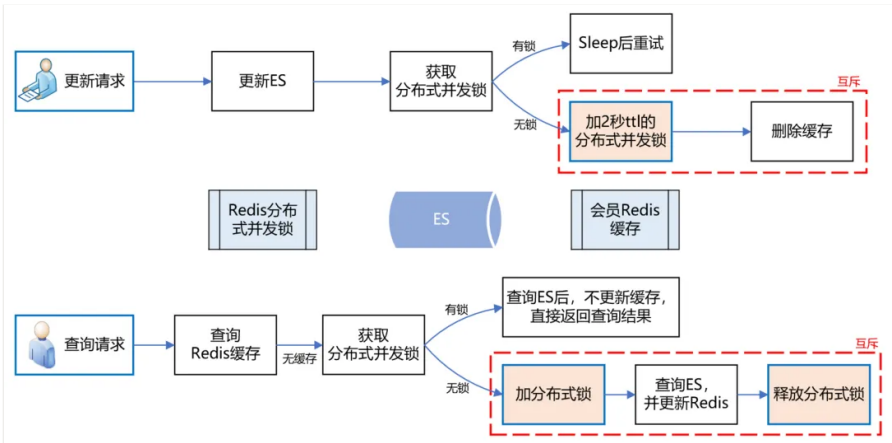
上述方案，乍一看似乎没什么问题了，但仔细分析，还是有可能导致缓存数据的不一致。

例如，在更新请求加分布式锁之前，恰好有一个查询请求获取分布式锁，而此时是没有锁的，所以它可以继续更新缓存。

但就在他更新缓存之前，线程 block 了，此时更新请求来了，加了分布式锁，并删除了缓存。当更新请求完成操作后，查询请求的线程活过来了，此时它再执行更新缓存，就把脏数据写到缓存中了。

发现没有？主要的问题症结就在于“删除缓存”和“更新缓存”发生了并发冲突，只要将它们互斥，就能解决问题。

如下图所示：

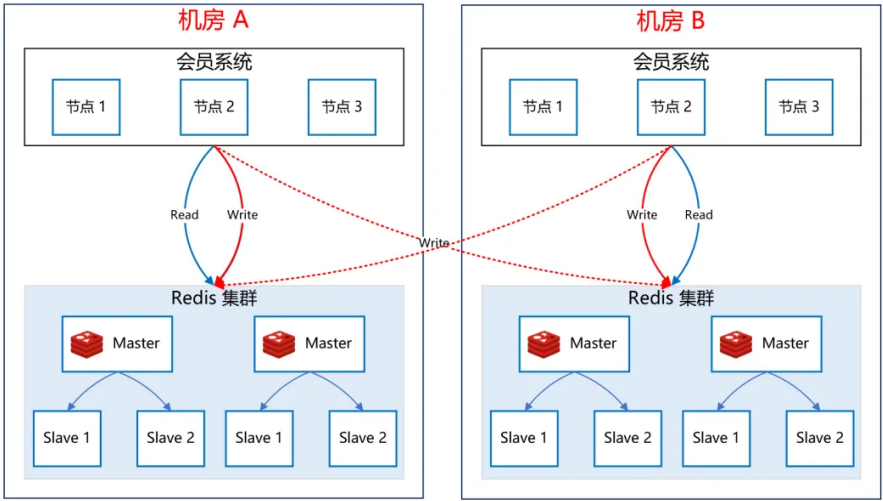


实施了缓存方案后，经统计，缓存命中率 90%+，极大缓解了 ES 的压力，会员系统整体性能得到了很大提升。

| Redis 双中心多集群架构

接下来，我们看一下如何保障 Redis 集群的高可用。

如下图所示：



关于 Redis 集群的高可用，我们采用了双中心多集群的模式。在机房 A 和机房 B 各部署一套 Redis 集群。

更新缓存数据时，双写，只有两个机房的 Redis 集群都写成功了，才返回成功。查询缓存数据时，机房内就近查询，降低延时。这样，即使机房 A 整体故障，机房 B 还能提供完整的会员服务。

高可用会员主库方案

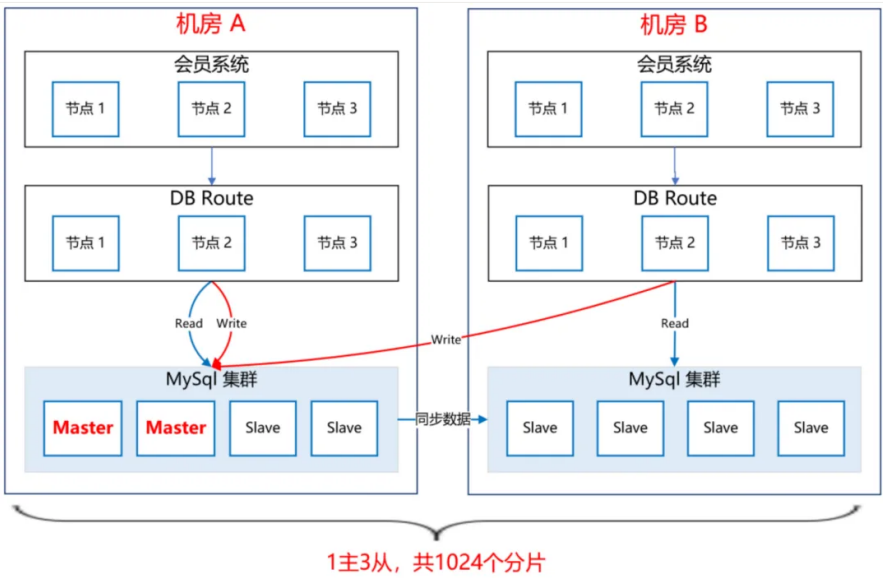
上述讲到，全平台会员的绑定关系数据存在 ES，而会员的注册明细数据存在关系型数据库。

最早，会员使用的数据库是 SqlServer，直到有一天，DBA 找到我们说，单台 SqlServer 数据库已经存储了十多亿的会员数据，服务器已达到物理极限，不能再扩展了。按照现在的增长趋势，过不了多久，整个 SqlServer 数据库就崩了。

你想想，那是一种什么样的灾难场景：会员数据库崩了，会员系统就崩了；会员系统崩了，全公司所有业务线就崩了。想想就不寒而栗，酸爽无比，为此我们立刻开启了迁移 DB 的工作。

| MySQL 双中心 Partition 集群方案

经过调研，我们选择了双中心分库分表的 MySQL 集群方案，如下图所示：



会员一共有十多亿的数据，我们把会员主库分了 1000 多个分片，平分到每个分片大概百万的量级，足够使用了。

MySQL 集群采用 1 主 3 从的架构，主库放在机房 A，从库放在机房 B，两个机房之间通过专线同步数据，延迟在 1 毫秒内。

会员系统通过 DBRoute 读写数据，写数据都路由到 master 节点所在的机房 A，读数据都路由到本地机房，就近访问，减少网络延迟。

这样，采用双中心的 MySQL 集群架构，极大提高了可用性，即使机房 A 整体都崩了，还可以将机房 B 的 Slave 升级为 Master，继续提供服务。

双中心 MySQL 集群搭建好后，我们进行了压测，测试下来，秒并发能达到 2 万多，平均耗时在 10 毫秒内，性能达标。

| 会员主库平滑迁移方案

接下来的工作，就是把会员系统的底层存储从 SqlServer 切到 MySQL 上，这是个风险极高的工作。

主要有以下几个难点：

- 会员系统是一刻都不能停机的，要在不停机的情况下完成 SqlServer 到 MySQL 的切换，就像是在给高速行驶的汽车换轮子。
- 会员系统是由很多个系统和接口组成的，毕竟发展了 10 多年，由于历史原因，遗留了大量老接口，逻辑错综复杂。这么多系统，必须一个不落的全部梳理清楚，DAL 层代码必须重写，而且不能出任何问题，否则将是灾难性的。
- 数据的迁移要做到无缝迁移，不仅是存量 10 多亿数据的迁移，实时产生的数据也要无缝同步到 MySQL。另外，除了要保障数据同步的实时性，还要保证数据的正确性，以及 SqlServer 和 MySQL 数据的一致性。

基于以上痛点，我们设计了“全量同步、增量同步、实时流量灰度切换”的技术方案。

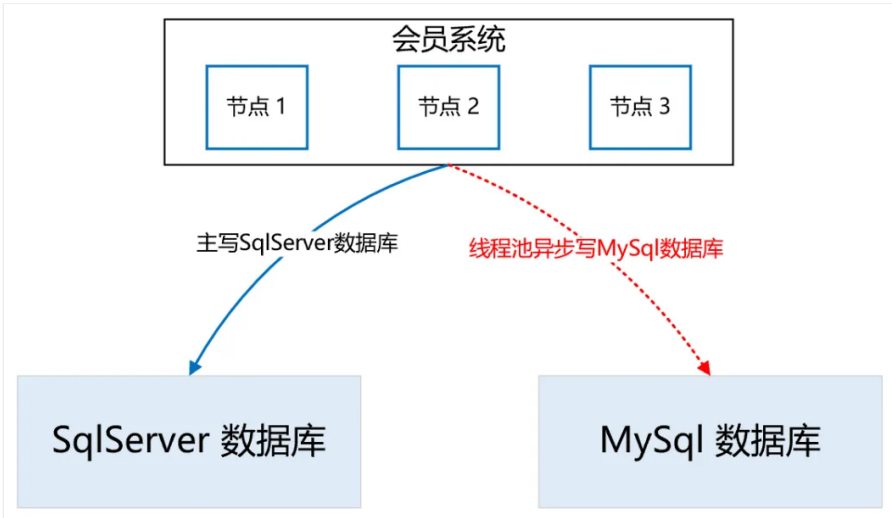
首先，为了保证数据的无缝切换，采用实时双写的方案。因为业务逻辑的复杂，以及 SqlServer 和 MySQL 的技术差异性，在双写 MySQL 的过程中，不一定会写成功，而一旦写失败，就会导致 SqlServer 和 MySQL 的数据不一致，这是绝不允许的。

所以，我们采取的策略是，在试运行期间，主写 SqlServer，然后通过线程池异步写 MySQL，如果写失败了，重试三次，如果依然失败，则记日志，然后人工排查原因，解决后，继续双写，直到运行一段时间，没有双写失败的情况。

通过上述策略，可以确保在绝大部分情况下，双写操作的正确性和稳定性，即使在试运行期间出现了 SqlServer 和 MySQL 的数据不一致的情况，也可以基于 SqlServer 再次全量构建出 MySQL 的数据。

因为我们在设计双写策略时，会确保 SqlServer 一定能写成功，也就是说，SqlServer 中的数据是 全量最完整、最正确的。

如下图所示：

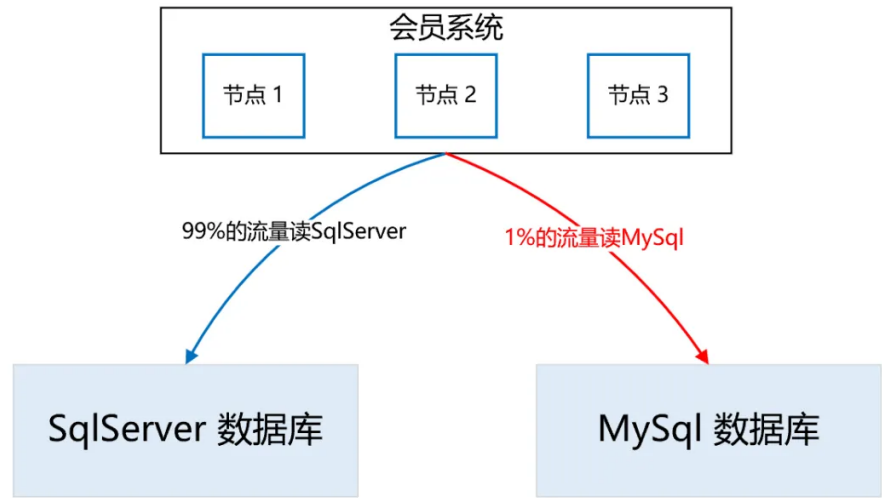


讲完了双写，接下来我们看一下“读数据”如何灰度。整体思路是，通过 A/B 平台逐步灰度流量，刚开始 100% 的流量读取 SqlServer 数据库，然后逐步切流量读取 MySQL 数据库，先 1%，如果没有问题，再逐步放流量，最终 100% 的流量都走 MySQL 数据库。

在逐步灰度流量的过程中，需要有验证机制，只有验证没问题了，才能进一步放大流量。

那么这个验证机制如何实施呢？方案是，在一次查询请求里，通过异步线程，比较 SqlServer 和 MySQL 的查询结果是否一致，如果不一致，记日志，再人工检查不一致的原因，直到彻底解决不一致的问题后，再逐步灰度流量。

如下图所示：



所以，整体的实施流程如下：



首先，在一个夜黑风高的深夜，流量最小的时候，完成 SqlServer 到 MySQL 数据库的全量数据同步。

接着，开启双写，此时，如果有用户注册，就会实时双写到两个数据库。那么，在全量同步和实时双写开启之间，两个数据库还相差这段时间的数据，所以需要再次增量同步，把数据补充完整，以防数据的不一致。

剩下的时间，就是各种日志监控，看双写是否有问题，看数据比对是否一致等等。

这段时间是耗时最长的，也是最容易发生问题的，如果有的问题比较严重，导致数据不一致了，就需要从头再来，再次基于 SqlServer 全量构建 MySQL 数据库，然后重新灰度流量。

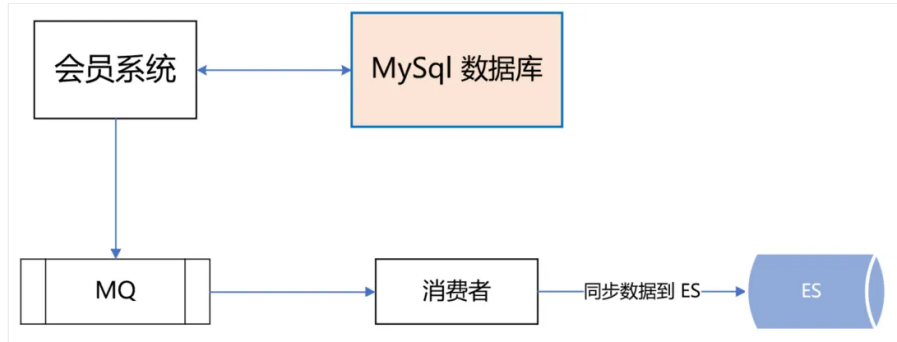
直到最后，100% 的流量全部灰度到 MySQL，此时就大功告成了，下线灰度逻辑，所有读写都切到 MySQL 集群。

| MySQL 和 ES 主备集群方案

做到这一步，感觉会员主库应该没问题了，可 dal 组件的一次严重故障改变了我们的想法。

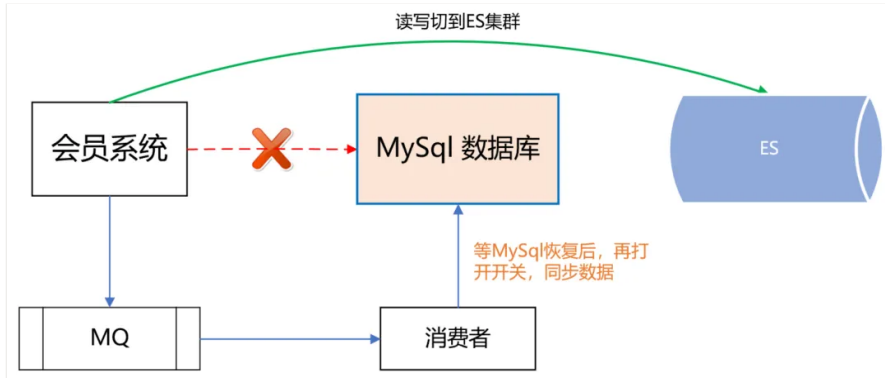
那次故障很恐怖，公司很多应用连接不上数据库了，创单量直线往下掉，这让我们意识到，即使数据库是好的，但 dal 组件异常，依然能让会员系统挂掉。

所以，我们再次异构了会员主库的数据源，双写数据到 ES，如下所示：



如果 dal 组件故障或 MySQL 数据库挂了，可以把读写切到 ES，等 MySQL 恢复了，再把数据同步到 MySQL，最后把读写再切回到 MySQL 数据库。

如下图所示：



异常会员关系治理

会员系统不仅仅要保证系统的稳定和高可用，数据的精准和正确也同样重要。

举个例子，一个分布式并发故障，导致一名用户的 APP 账户绑定了别人的微信小程序账户，这将会带来非常恶劣的影响。

首先，一旦这两个账号绑定了，那么这两个用户下的酒店、机票、火车票订单是互相可以看到的。

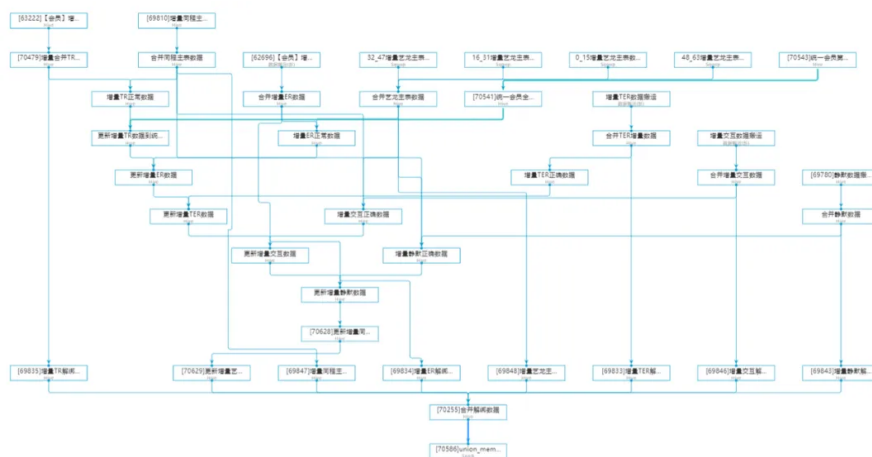
你想想，别人能看到你订的酒店订单，你火不火，会不会投诉？除了能看到别人的订单，你还能操作订单。

例如，一个用户在 APP 的订单中心，看到了别人订的机票订单，他觉得不是自己的订单，就把订单取消了。

这将会带来非常严重的客诉，大家知道，机票退订费用是挺高的，这不仅影响了该用户的正常出行，还导致了比较大的经济损失，非常糟糕。

针对这些异常会员账号，我们进行了详细的梳理，通过非常复杂烧脑的逻辑识别出这些账号，并对会员接口进行了深度优化治理，在代码逻辑层堵住了相关漏洞，完成了异常会员的治理工作。

如下图所示：



展望：更精细化的流控和降级策略

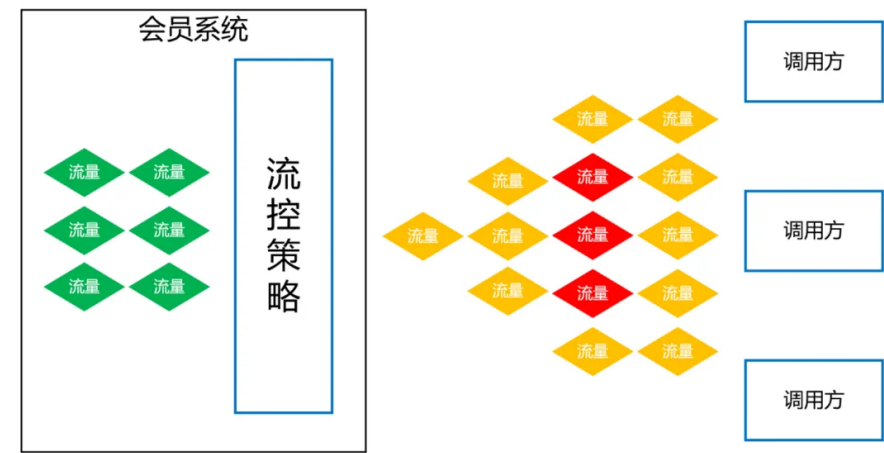
任何一个系统，都不能保证百分之百不出问题，所以我们要面向失败的设计，那就是更精细化的流控和降级策略。

| 更精细化的流控策略

热点控制。针对黑产刷单的场景，同一个会员 id 会有大量重复的请求，形成热点账号，当这些账号的访问超过设定阈值时，实施限流策略。

基于调用账号的流控规则。这个策略主要是防止调用方的代码 bug 导致的大流量。例如，调用方在一次用户请求中，循环很多次来调用会员接口，导致会员系统流量暴增很多倍。所以，要针对每个调用账号设置流控规则，当超过阈值时，实施限流策略。

全局流控规则。我们会员系统能抗下 tps 3 万多的秒并发请求量，如果此时，有个很恐怖的流量打过来，tps 高达 10 万，与其让这波流量把会员数据库、ES 全部打死，还不如把超过会员系统承受范围之外的流量快速失败，至少 tps 3 万内的会员请求能正常响应，不会让整个会员系统全部崩溃。



| 更精细化的降级策略

基于平均响应时间的降级。会员接口也有依赖其他接口，当调用其他接口的平均响应时间超过阈值，进入准降级状态。

如果接下来 1s 内进入的请求，它们的平均响应时间都持续超过阈值，那么在接下的时间窗口内，自动地熔断。

基于异常数和异常比例的降级。当会员接口依赖的其他接口发生异常，如果 1 分钟内的异常数超过阈值，或者每秒异常总数占通过量的比值超过阈值，进入降级状态，在接下的时间窗口之内，自动熔断。

目前，我们最大的痛点是会员调用账号的治理。公司内，想要调用会员接口，必须申请一个调用账号，我们会记录该账号的使用场景，并设置流控、降级策略的规则。

但在实际使用的过程中，申请了该账号的同事，可能异动到其他部门了，此时他可能也会调用会员系统，为了省事，他不会再次申请会员账号，而是直接沿用以前的账号过来调用，这导致我们无法判断一个会员账号的具体使用场景是什么，也就无法实施更精细的流控和降级策略。

所以，接下来，我们将会对所有调用账号进行一个个的梳理，这是个非常庞大且繁琐的工作，但无论如何，硬着头皮也要做好。

欢迎大家进行观点的探讨和碰撞，各抒己见。如果你有疑问，也可以找我沟通和交流。扩展：接私活儿

最后给读者整理了一份BAT大厂面试真题，需要的可扫码回复“面试题”即可获取。



— END —

公众号后台回复 架构 或者 架构整洁 有惊喜礼包！

顶级架构师交流群