09 | 如何打造无状态的存储实现随时切库的写入服务?

在上一讲里，我介绍了如何实现分库分表的架构方案，以及如何解决业务不断发展所产生的数据容量的问题。

但是分库分表只解决了容量的问题，并没有解决写服务的高可用问题，或者说分库分表在一定程度上增加了系统故障的概率。从概率上看，原有的单库架构有 50% 的可能性会发生数据库故障。但如果是 5 个分库，则会有 96%（具体计算方法见下方注释）的可能性出现故障。因此，采用分库分表的架构之后，系统的稳定性变得更低了。

> 注：共计有 5 个分库，每一个分库不故障的概率都是 50%。如果整个集群不发生故障，就需要每一个分片都不故障，那么整个集群不发生故障的概率为 $1\text{-}50\%^5=96\%$。

在读服务里，可以采用数据冗余来保障架构的高可用，但在写服务里则无法使用此方案，因为写入服务的数据是用户提交产生的，无法在写入时使用冗余来提高高可用性。写冗余需要有满足 CAP 原则的存储支持，而我们知道，CAP 原则最多只能同时满足两个特性，要么 CP 要么 AP，因此写冗余无法直接满足。本讲将介绍一种能够实现随时切库的高可用写服务方案，不管是单库还是分库分表的原有架构，它都可以原生支持。

写入业务的目标是成功写入

在本专栏的第一讲里，我曾介绍过写服务的特点，此处再做一个简单的回顾。写业务是指需要将用户传入的数据进行全部存储的一种场景，常见的案例有：

- 在各大网站提交的申请表单，比如落户申请、身份证办理申请、护照办理申请等；
- 在电商、外派平台里的购物订单，其中会包含商品、价格、收货人等信息；
- 在重要期刊和一些论坛里，提交的论文、博客等。

假设明天就是论文提交截止日，你一定希望论文提交系统不要出现问题。即使系统出现故障，在论文提交后晚几分钟才能查看内容，但只要你提交成功了，这类故障并不会对你产生太大的影响。

其他场景也是类似。对于写入业务，当出现各种故障时，最重要的是保证系统可写入。

只要有可用存储即可写入

那么什么是系统可随时写入呢？就是当出现任何故障，如网络中断、CPU 飙升、磁盘满等问题时，你的系统依然可以随时写入数据。

如何保证随时可写入?

在上一讲提到的分库分表架构里，假设当前只有两个分库，并且这两个分库分别部署在不同机房里。架构如下图 1 所示：

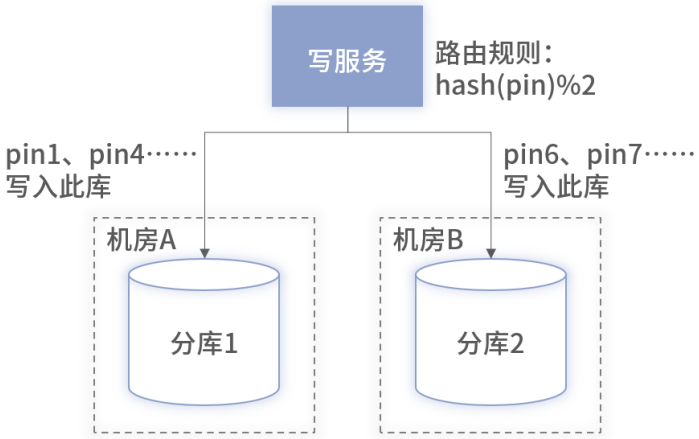

图 1：分库分表且分机房架构

当其中一个分库所处的机房出现网络故障，导致该分库不可达时，理论上系统就出现故障了。在上一讲里我们提到过，分库分表后，数据在写入时是按固定规则（比如用户账号）路由到具体分库的，当某个分库不可达时，对应规则的数据就无法写入了。

但是写服务最重要的是保障数据写入，为了保障可写入，能不能在某一个分库故障（如网络不可达）后，将原有的数据全部写入当前可用的数据库呢？从保障数据可随时写入的角度看，此方式是可行的。升级后的架构如下图 2 所示：

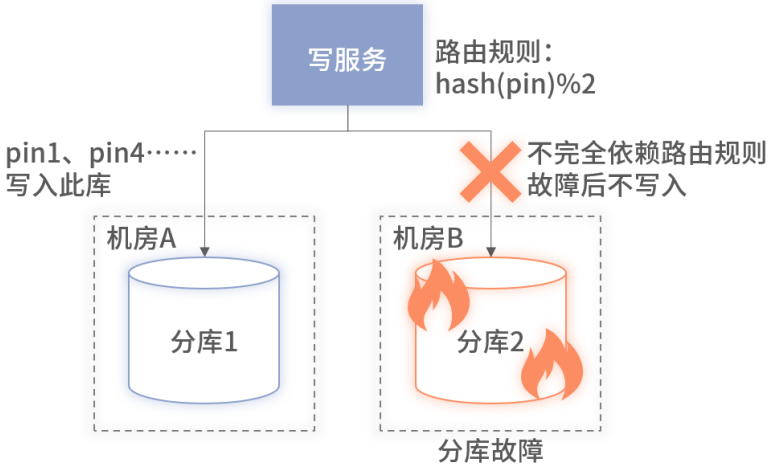

图 2：可随时写入的架构方案

上述这种当分库分表里一个分库出现故障后，就随机寻找一个可用的数据库进行写入的方式即是一种保障系统高可用的架构方案。此方案可以将图 2 和“**08 讲**”提到的按固定规则路由的分库分表方案进行结合，方案如下图 3 所示：

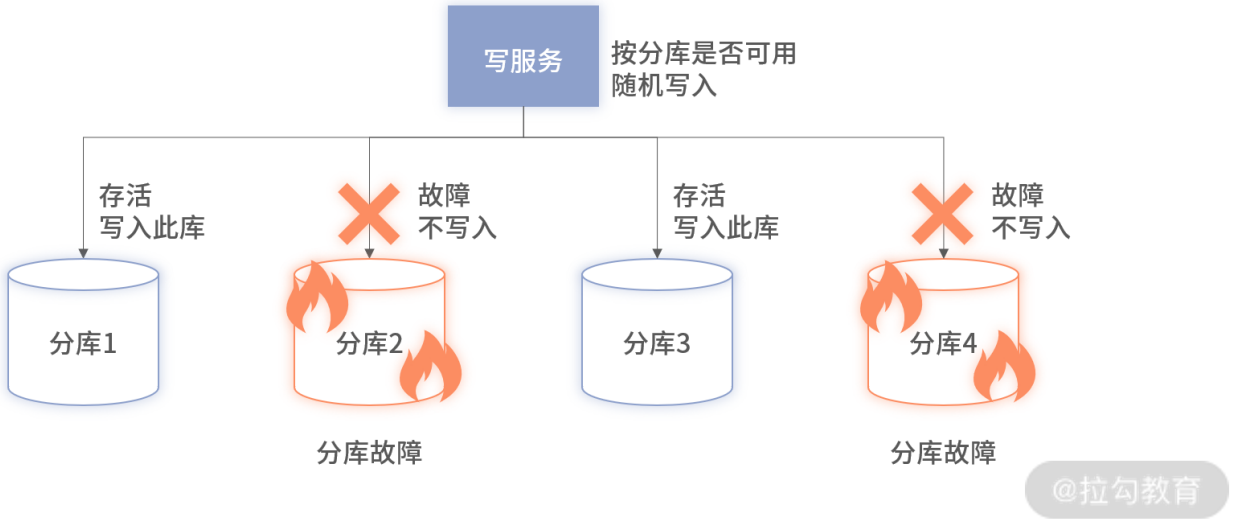

图 3：结合后的随机写入架构

结合后的架构里，存储依然使用分库分表，但写入规则则发生了一些变化。它不再按固定路由进行写入，而是根据当前实时可用的数据库列表进行随机（如顺序轮流）写入。如果某一台数据库出现故障不可用后，则把它从当前可用数据库列表移除。如果数据库大面积不可用，可用列表中的数据库变少时，你可以适当地扩容一些数据库资源，并将它添加至当前可用的数据列中。因为此架构可以实现随时切换问题数据库、随时低成本扩容数据库，故又称它为**无状态存储架构设计**。

如何维护可用列表?

在写服务运行过程中，可以通过自动感知或人工确认的方式维护可用的数据库列表。在写服务调用数据库写入时，可以设置一个阈值。如果写入某一台数据库，在连续几分钟内，失败多少次，则可以判定此数据库故障，并将此判定进行上报。

当整个写服务集群里，超过半数都认为此数据库故障了，则可以将此数据库从可用列表中剔除。你可能听出来了，此判定方法类似于 Paxos 算法，它在分布式协调和故障迁移中十分流行，此处也适当进行了一些借鉴。

判定某一台数据库故障并将其下线是一个挺耗费成本的事情，为了防止误剔除某一台只是发生网络抖动的数据库，可以在真正下线某一个机器前，增加一个报警，给人工确认一个机会。可以设置当多少时间内，人工未响应，即可自动下线。

上述是将可用列表机器进行下线的方案。对于新扩容的数据库资源，通过系统功能自动加入即可。虽然，本讲介绍的方案是可以按顺序进行随机写入的，但还是**建议在实现时将顺序随机写入升级为按权重写入**，比如对新加入的机器设置更高的写入权重。因为新扩容的机器容量为空，**更高的写入权重，可以让数据更快地在全部数据库里变得均衡**。增加权重的架构如下图 4 所示：

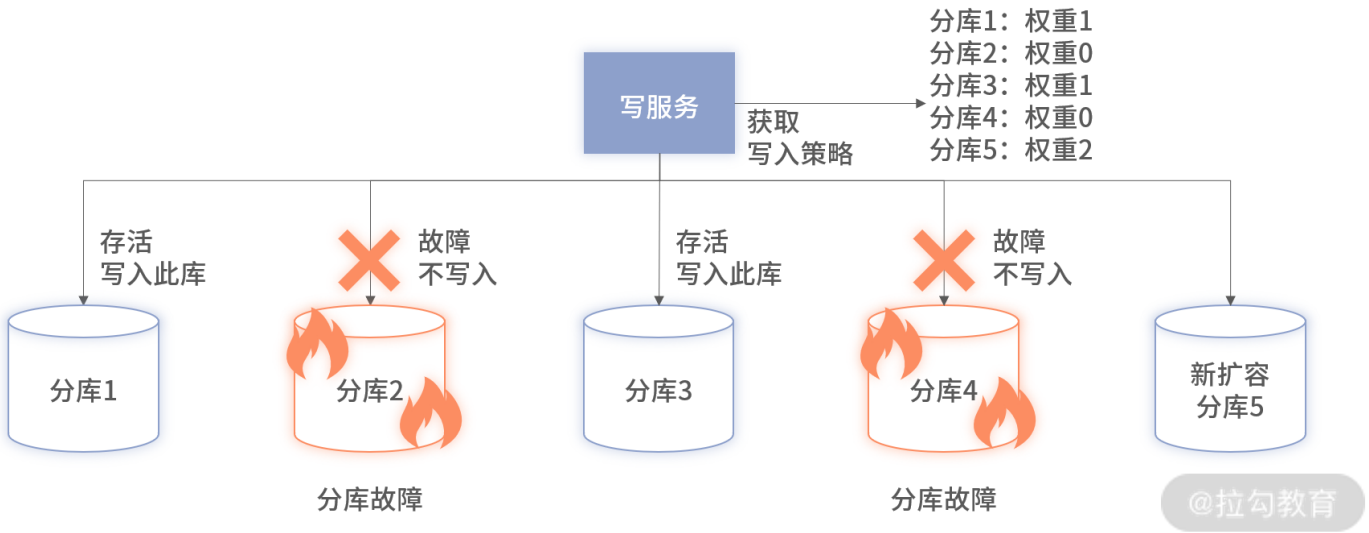

图 4：按权重的数据写入架构

写入后如何处理

通过数据库写入的随机化，实现了写服务的高可用方案。但不得不说，虽然解决了写入的高可用，但想要达成一个完整的架构方案，此设计还有几个重要的技术点需要解决。

- 如果某一个分库故障后便将其从可用列表中移除，应该如何处理其中已写入的数据呢？
- 因为数据是随机写入，应该如何查询写入的数据呢？

对于上述的问题，我先向你介绍一个整体的架构解决方案。如下图 5 所示：

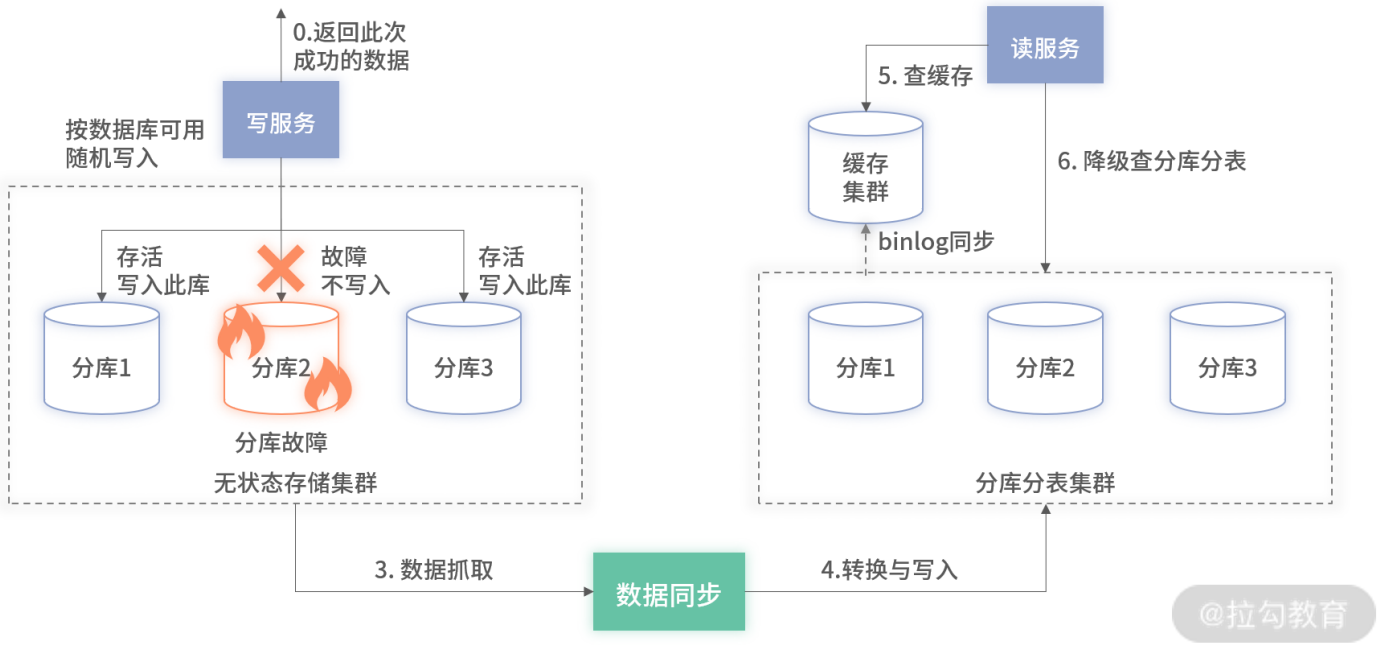

图 5：采用随机写入后的整体架构方案

简单来说，整体的架构方案就是在上一讲分库分表的方案基础上，做了进一步升级。架构图的右边部分和分库分表几乎一模一样，左边部分则是本讲的架构方案带来的新增内容，它主要包含以下两个部分的内容。

1. 第一部分是前面讲述的数据随机写入模块，它保证了在故障时数据依然可以写入。
2. 第二部分则是数据同步模块，它将数据从随机写入的数据库集群实时地同步至分库分表集群里。后续的所有流程，都和原有的保持一致了。

通过写入模块和同步模块的配合，即实现了基于无状态存储的高可用架构的整套方案。

如何做数据同步？

在采用同步模块后，从逻辑上是可以实现写入后数据可查询的。但这只是逻辑上的，因为增加了同步模块后数据延迟是不可避免。更有甚者，可能因数据同步存在 Bug 导致数据一直未同步的场景。

针对上述问题，这里不卖关子，解决的架构方案如下图 6 所示：

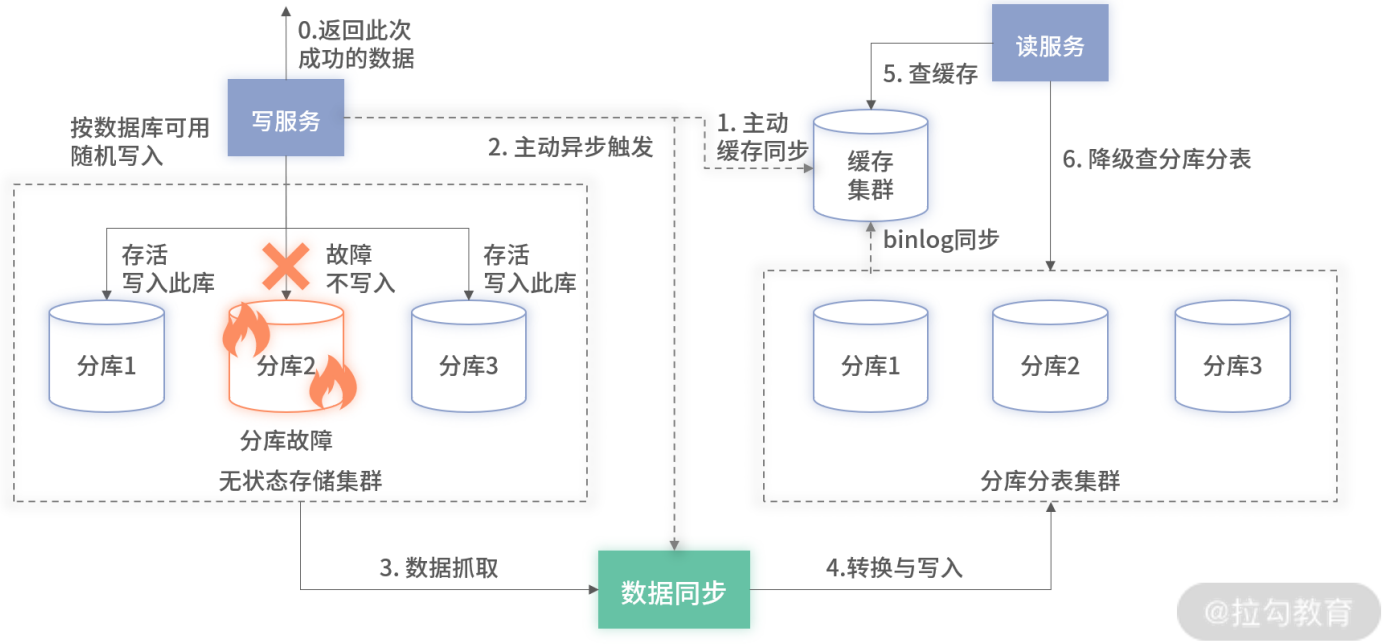

图 6：解决数据延迟的架构

在数据写入后，用户需要立即查看写入内容的场景并不太多。比如上传完论文后，你只要立刻确定论文上传成功且查看系统里的论文内容和你上传的一致即可。

对于这些有时延要求的场景，我们在图 5 的架构里已经进行了单独预处理。当数据写入随机存储成功后，可以在请求返回前，主动地将数据写入缓存中，同时将此次写入的数据全部返回给前台。但此处并不强制缓存一定要写成功，缓存写入失败也可以返回成功。对时延敏感的场景，可以直接查询此缓存。

对于无状态存储中的数据，可以在写入请求中主动触发同步模块进行迁移，如上图 5 中标号 X 的流程。同步模块在接收到请求后，立刻将数据同步至分库分表及缓存中，后续流程就和上一讲保持一致了。

主动触发同步模块的请求及同步模块本身运行都有可能出现异常，对于可能出现的异常情况，可以设计兜底策略进行处理。兜底策略和同步模块比较类似，主要架构如下图 7 所示：

图 7：兜底同步策略

兜底的同步对于无状态存储中的数据按创建时间进行不断轮询，轮询会对超过设置的时间阈值（如 5S）仍未得到同步的数据进行主动同步。此兜底方案保证了当上述缓存预写入和主动同步故障时，数据仍然可以写入分库分表。此外，如果兜底策略的时间阈值设置得过小，有可能和主动同步产生重复同步。**对于重复同步，在分库分表处可以设置数据库唯一索引、插入前查询进行简单防重即可。**

缓存可降级

因为主动写入缓存可能存在异常，导致数据未写入缓存，且主动数据同步和兜底同步是先写分库分表再通过 Binlog 刷新缓存，存在一定的延迟。因此在查询时需要具备降级功能，当缓存中未查询到时，可以主动降级到数据库进行一次兜底查询，并将查询到的值存储至缓存中。后续再有数据变更，和原有保持一致即可。可降级的架构方案如下图 8 所示：

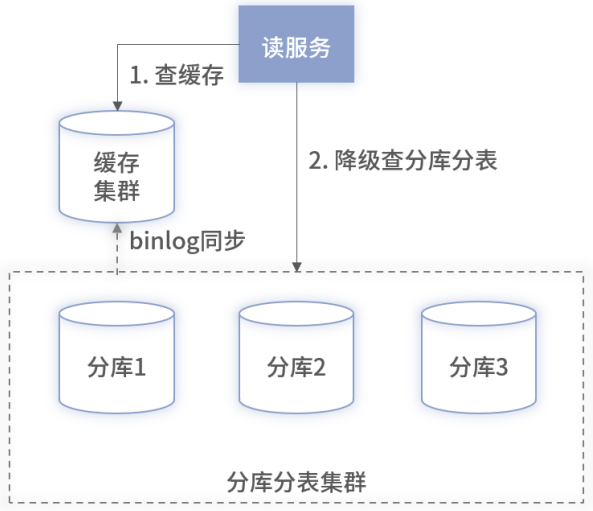

图 8：缓存可降级架构方案

其他功能流程保持复用

你可能会疑惑，采用无状态存储后，原有分库分表及无状态存储集群，除了上述之外的一些架构细节，是否还会有什么变化？比如，假设某一台无状态存储的数据库故障后，如果该故障数据库中仍有数据未同步如何处理？

其实此数据库故障后的处理流程和任何线上数据库故障一样，都是经由 DBA 确认该数据库的从库和主库数据是否一致。如一致，升级该库的从库为主库，并将其加回无状态集群即可。对于其余的一些架构细节亦是如此，这里不再赘述。

总结

本讲**基于无状态存储打造了一个可以随时切库的高可用写服务**。虽然当某一台无状态存储出现故障时，其中遗留的数据会出现短时的同步延迟，但**此方案可以将出问题的无状态存储立刻移除，保障了写入的 7*24 高可用**。

对于一些对写入有极致要求的场景，比如在电商的大促零点时刻，每一秒钟都会产生上百万、千万的下单金额。可以进行适当地体验降级，比如让用户晚几秒看到历史订单列表，但一定要保障在此时刻用户可以下单并看到此订单的结果，以及对应的收件信息（如收货地址、电话、收货人）。对于上述场景，此方案也可以很好地应对。

此方案虽然应对了对写入有极致要求的场景，但它会将系统变得更加复杂且落地的开发成本和部署成本都更高，这也是它的弊端。还是回到本课程一直秉承的理念，架构是为了达到解决问题的目标而做的平衡，而不是技术比拼。

最后留一个讨论题给你。此方案适合或不适合你经历/知道过的那些场景？欢迎留言区留言，我们一起讨论。