「腾讯云 NoSQL 技术」之 Redis 篇 | 揭晓腾讯云Redis水平扩缩容极致流畅背后的技术玄机

图片

> 凭借高吞吐与低延迟，Redis 已成为互联网缓存的事实标准，承担了互联网业务90%的请求，其弹性扩展能力是应对业务增长与突发流量的关键保障，社区原生扩缩容方案在性能、可靠性及扩容速度等方面存在局限。为了解决这一痛点，腾讯云 NoSQL 团队基于多年技术沉淀与实践验证，创新性地提出基于slot原子化迁移的水平扩缩容方案，实现了无缝平滑的分片数量调整。本文将从技术原理、业界方案对比及开源贡献等多维度展开分析，重点解读腾讯云如何通过内核级优化解决Redis弹性扩缩容的长期痛点。该方案自上线后已广泛服务于腾讯集团的内部客户和公有云上的外部客户，经历了大规模实践的检验，稳定性有保障，助力业务侧实现资源利用率提升与运维成本降低。
>
> 作者：腾讯云NoSQL团队宋平凡，朱彬彬

在当今互联网应用架构中，Redis 凭借其极低的访问延迟、优异的高并发处理能力、丰富的数据结构支持、完善的功能生态以及成熟的生态，已成为不可或缺的KV缓存和数据库存储解决方案。在权威的 DB-Engines 数据库流行度排名中，Redis 长期位列键值型数据库首位，并稳定居于全部数据库类别的前十名，如图1所示。

Rank			DBMS	Database Model	Score		
Sep 2025	Aug 2025	Sep 2024			Sep 2025	Aug 2025	Sep 2024
1.	1.	1.	Oracle	Relational, Multi-model	1170.62	-50.08	-115.97
2.	2.	2.	MySQL	Relational, Multi-model	891.77	-23.69	-137.72
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	717.32	-36.84	-90.45
4.	4.	4.	PostgreSQL	Relational, Multi-model	657.17	-14.08	+12.81
5.	5.	5.	MongoDB	Document, Multi-model	380.50	-15.08	-29.74
6.	6.	↑ 7.	Snowflake	Relational	190.19	+11.29	+56.47
7.	7.	↓ 6.	Redis	Key-value, Multi-model	145.17	-2.02	-4.25
8.	8.	↑ 9.	IBM Db2	Relational, Multi-model	124.19	-3.12	+1.14
9.	9.	↑ 14.	Databricks	Multi-model	124.06	+8.25	+39.82
10.	10.	↓ 8.	Elasticsearch	Multi-model	118.26	+3.99	-10.53

▲图1.DB-Engines 数据库流行度排行榜(2025.09)

然而，随着业务发展，几乎所有客户都会面临这样一个难题：业务上升期需要快速扩容，下降期需要及时缩容。但令人头疼的是，Redis 的命令执行是单线程模型，一旦主线程打满 CPU，无法像其他数据库一样通过增加 CPU 核心来纵向提升单个节点的处理能力。基于这样的业务困境，构建高效、平稳的水平扩缩容机制成为保障业务弹性的关键需求。

为应对这一挑战，业界提出了多种水平扩缩容方案。本文首先给大家介绍 Redis 社区提供的原生水平扩缩容方案，包括方案的底层实现原理以及方案本身的局限性；随后，本文会进一步介绍业界针对 Redis 水平扩缩容的需求所提出的另外两种大相径庭的解决方案，这两种方案虽然解决了社区原生方案的不足，但又引入了新的痛点，很难让客户满意；最后，本文会分享我们腾讯云 NoSQL 团队针对这一业界难题的思考，以及最终提出的 slot 原子化迁移方案，该方案有效解决了现有方案的共性痛点，为用户带来平滑、高效的扩缩容体验。

那这一切是怎么做到的呢？让我们带着问题，跟着我一起来一探究竟吧！

1 Redis 社区原生的水平扩缩容方案

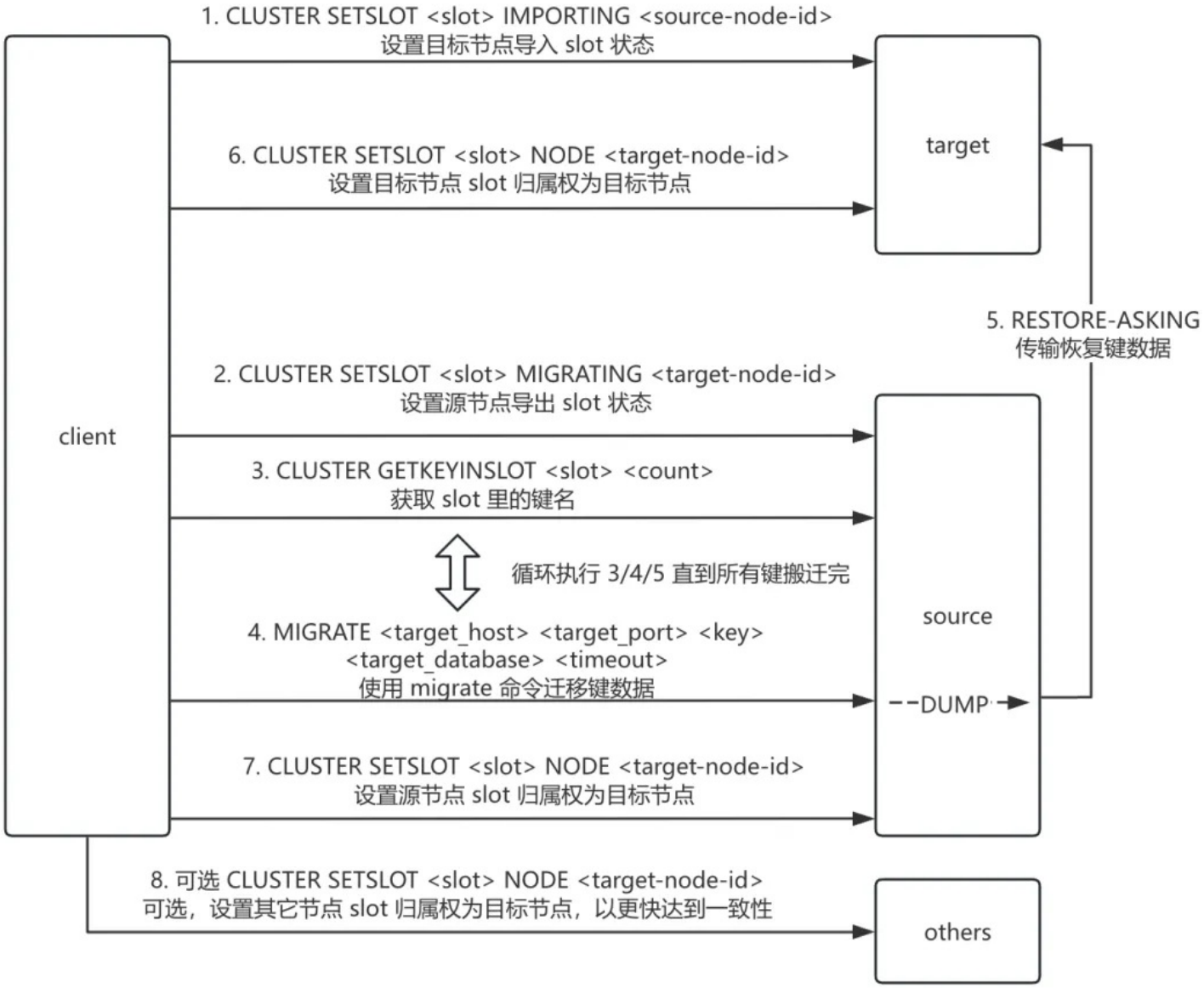
1.1 Redis 社区原生水平扩缩容方案的原理和实现

（1）基于 key 粒度的 slot 迁移

Redis 从3.0版本开始支持集群功能，并且为了简化集群元数据的复杂度，新引入了哈希槽(hash slot，后文简称slot)的概念，整个集群中的全量数据被

划分为16384个 slot 。在集群相关的逻辑中，无论是节点间数据的分布策略，还是客户请求的路由规则，都以 slot 为判断依据。

Redis 3.0 也为集群提供了原生的水平扩缩容方案，这个方案逻辑上是通过在节点之间迁移 slot 来达到目的，但从底层实际的实现上来说，它的 slot 迁移过程并没有原子性，而是以 key 为基本粒度，通过分批搬 key 来达到搬 slot 的目的。社区迁移方案的详细流程梳理如图2所示：



▲图2. Redis 社区基于key粒度的原生slot迁移流程

其中，1/2是状态标记阶段，主要是在目标节点和源节点分别标记 slot 的 IMPORTING (导入)和 MIGRATING (导出)状态；3/4/5是数据搬迁阶段，循环通过 MIGRATE 命令搬迁一批 key ，直至完成所有 key 的搬迁，这也是耗时最长的一个阶段；6/7/8则是归属转移阶段，负责清理 slot 的 IMPORTING 和 MIGRATING 状态，并将 slot 的归属权更新为目标节点。可以看到，在流程的标记和转移阶段，处理的粒度是整个 slot ；但在数据搬迁阶段，处理的粒度则是 slot 中的 key 。

（2）迁移过程中对业务请求的处理

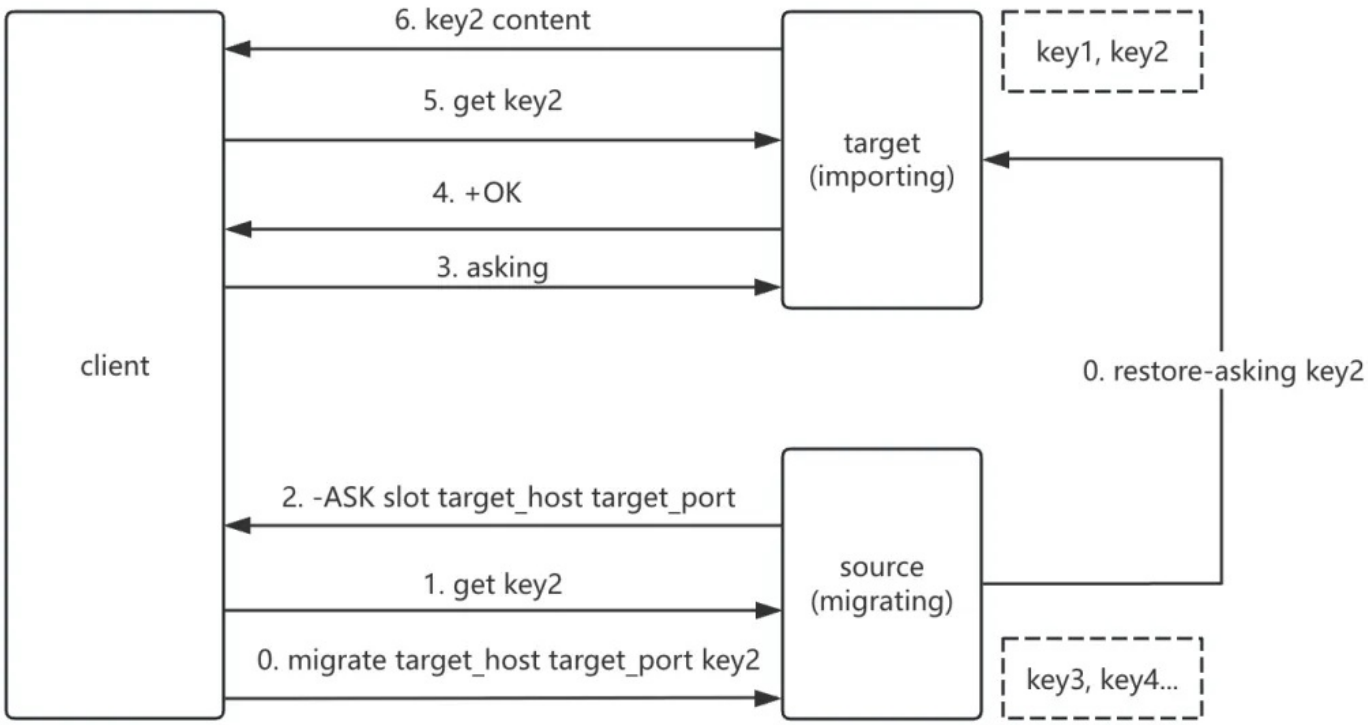
因为迁移过程的非原子性，所以在迁移的过程中会有一个时间段， slot 中的一部分 key 已经搬迁到了目标节点，而另一部分 key 还在源节点等待搬迁。而社区提供的是一个在线的热迁移方案，过程中无需业务停服。所以，就面临一个问题，如果迁移中有客户端要访问这个 slot 中的 key ，它也不知道这个 key 是在源节点还是目标节点，那要怎么办呢？社区方案给出的方法是 ASK 重定向，大致的原理如图3所示，说明如下：

1. 客户端优先将请求发送到源节点，源节点如果能找到这个 key ，就直接给客户端返回结果。

2. 源节点如果没找到，但是检测到这个 key 所属的 slot 在本节点处于 MIGRATING 状态，则这个 key 可能已经被搬到了目标节点；此时源节点会给客户端返回 ASK 重定向错误，格式为：-ASK <slot> <target-ip:target-port>

3. 客户端收到 ASK 重定向后，提取出目标节点的地址信息，先给目标节点发送 ASKING 命令，给连接打上特殊的 CLIENT_ASKING 标记，再将原始请求重发给目标节点。

4. 目标节点检测到这个 key 所属的 slot 在本节点处于 IMPORTING 状态，且这个请求的连接有 CLIENT_ASKING 标记，才会临时允许执行这个请求，并在执行结束后立刻清理这个连接的 CLIENT_ASKING 标记。



▲图3. Redis集群slot迁移过程中的ASK重定向原理

1.2 Redis 社区原生水平扩缩容方案面临的挑战

我们已经详细介绍了 Redis 社区原生的水平扩缩容方案，这个方案在多数普通场景下可以正常使用，并且它还是有不少优点的：不用业务停服，增减的分片数很灵活等等。但这个方案也有不少问题：影响业务请求，自身不够健壮，扩缩容速度太慢等等，方方面面都面临不少的挑战。

（1）社区方案影响业务请求带来的挑战

a. 大 key 阻塞问题

前文有提到，社区方案中实际的 key 搬迁是通过 MIGRATE 命令完成的，这个命令大致可以等价于：DUMP + RESTORE + DELETE。它的底层实现如下：

1. 源节点复用 DUMP 命令的逻辑，将 key 的内容序列化为一个 RDB 格式的二进制串；

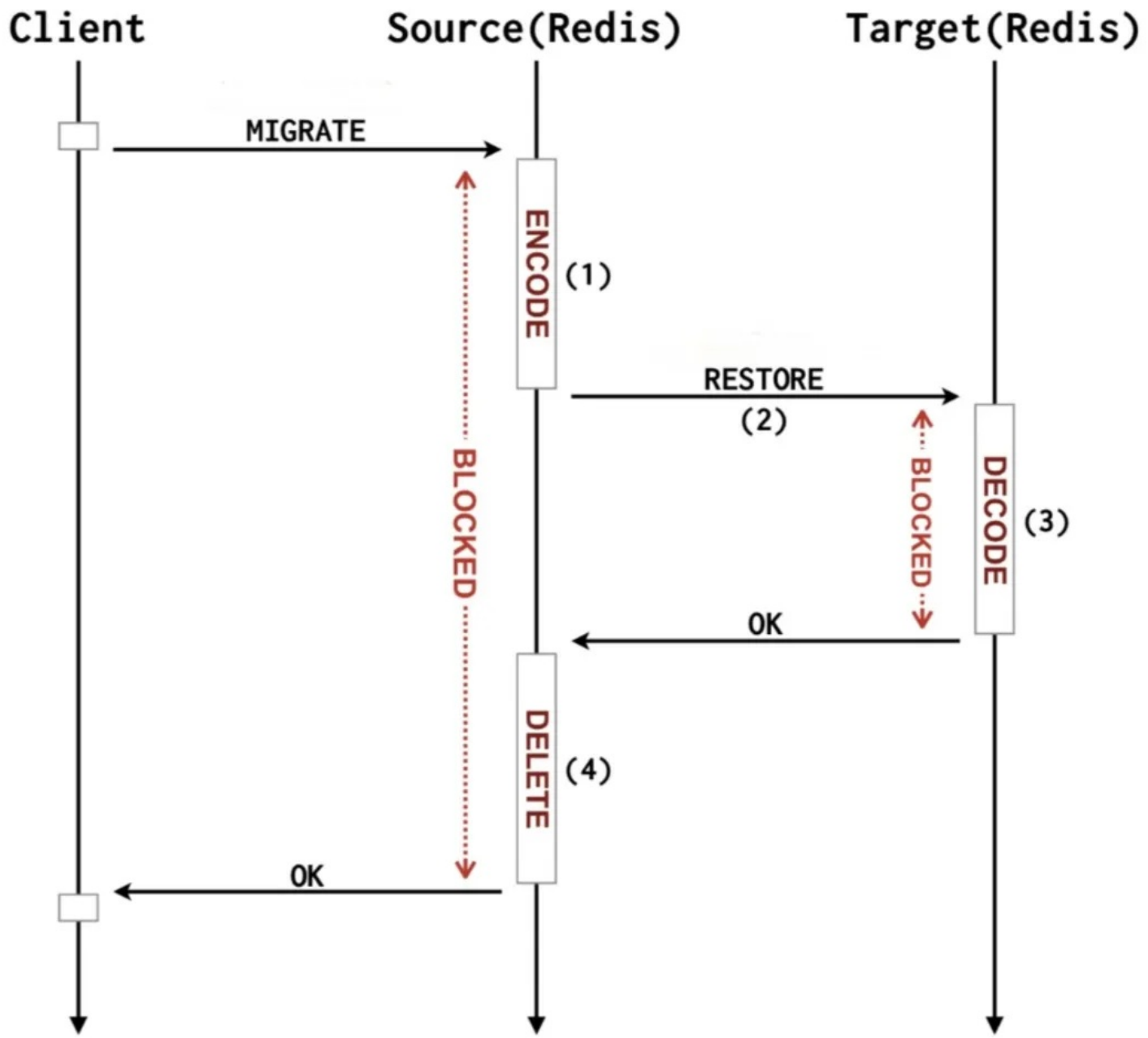
2. 源节点通过 RESTORE 命令将二进制串传输给目标节点；

3. 目标节点执行 RESTORE 命令将二进制串反序列化加载到DB；

4. 源节点收到确认后删除这个key。

其中，RESTORE 命令在集群模式下会被替换成 RESTORE-ASKING 命令，这两个命令的底层实现基本一致，区别可以简单理解为：RESTORE-ASKING = ASKING + RESTORE。

在 Redis 的设计里，命令的执行是单线程模型。而 MIGRATE 和 RESTORE 都是同步阻塞的命令，也就意味着，在单个 key 搬迁的整个过程中，源节点会阻塞所有其它请求；同理，目标节点在反序列化加载的过程中，也会阻塞所有其它请求。如图4所示：



▲图4. MIGRATE 命令在执行过程中对源节点和目标节点的阻塞

MIGRATE 的这种阻塞特性，对于较小的 key 影响不大。但因为 Redis 支持 zset/hash/list 等复杂结构，很多业务在使用中可能会产生含有几十万上百万元素的大 key ，这种大 key 无论是 DUMP ， RESTORE 还是 DELETE ，都需要好几秒，会给业务带来严重的时延抖动和超时报错。更糟糕的是，如果集群中出现含有几百万甚至上千万元素的大 key ， MIGRATE 可能会阻塞20多秒，已经超出了节点心跳的15s超时设置，会导致搬迁中的节点被判死，集群出现不可用。

b. 多key请求异常

如前所述，社区方案在迁移过程中，提供了 ASK 重定向机制来为业务请求持续提供服务。这个机制对于单 key 命令是够用的，但如果业务使用的是类似 MSET/MGET 这种多key命令，那情况就复杂了：

- 如果命令访问的所有 key 都在源节点，或者所有 key 都已经搬到了目标节点，那还是能正常响应的；
- 如果命令中访问的一部分 key 还在源节点，但另一部分 key 已经搬迁到了目标节点，此时源节点就会返回 -TRYAGAIN 错误，让客户端重试，一直等到此命令要访问的全部 key 搬迁到目标节点，或者客户端重试超时，会给业务带来严重的时延抖动和超时报错；
- 还有一种更坑的情况，命令中访问的一部分 key 在源节点，但还有一部分 key 根本不存在，这种情况下客户端会反复重试，只能等到重试超时报错。

c. LUA 请求异常

Redis 的 LUA 为用户提供了类似关系数据库存储过程的高级功能，让业务在服务端原子性执行复杂逻辑，深受广大用户欢迎。虽然 Redis 也支持用户通过 EVAL 命令将 LUA 脚本的内容和参数一并传递，但为了减少每次重复传输脚本的代价，更好的做法是，业务启动的时候通过 SCRIPT LOAD 命令将LUA脚

本的内容加载到数据库并得到脚本对应的 SHA1 ，后续都只需通过 EVALSHA 命令传递 SHA1 和参数即可调用相应的 LUA 脚本。

但是社区的 slot 迁移方案有一个特点，就是只会搬 key ，不会搬迁源节点的 LUA 脚本。这就导致一旦发起后，无论是过程中还是结束后在新节点执行 EVALSHA 命令都会报 -NOSCRIPT 错误，需要业务在新节点重新执行 SCRIPT LOAD 命令加载对应 LUA 脚本才行。

d. 业务性能受损

Redis是一个主打高并发和低时延的内存数据库，不幸的是，社区方案对 QPS 和时延这两个维度都有明显的影响：

- 从 QPS 的维度来说，因为搬迁过程本质上也是在源和目标分别执行 MIGRATE 和 RESTORE 命令，而 Redis 的命令执行是单线程的，搬迁命令占多了自然业务请求能用的就少了，这对于一些本身就因为 QPS 不够用而扩容的业务来说，简直就是雪上加霜。
- 从时延的维度来说，受到影响有两个原因，
- 一个是命令执行的单线程模型，原本只需要处理用户请求，现在插入大量搬迁相关的命令，用户的请求不可避免地要排队，平均时延增加；甚至搬迁的 key 稍大一些还会因主线程阻塞带来时延的大毛刺。
    - 另一个则是 ASK 重定向机制。原本只需要直接对应节点，网络只有一跳；现在可能因 key 不在源节点而额外增加一跳，带来平均时延的增加。

（2）社区方案健壮性不足带来的挑战

a. 迁移中断难以处理

因为社区的迁移方案不是原子的，一旦某个 slot 开始迁移，必定slot中有一部分 key 在目标节点，另一部分 key 在源节点。此时无论发生何种情况，想要无损地中断迁移都是不可能的。只能看哪一边的 key 更多，如果目标节点上已经有大多数 key ，就硬着头皮继续把剩下的 key 也搬迁到目标节点；如果源节点上还有大多数 key 没搬，那就把目标节点上的 key 再想办法搬回来，但是搬回来的过程中如何保证这些 key 不会有写冲突也很麻烦。

b. 迁移状态丢失问题

社区方案的迁移状态只维护在主节点里，从节点里没有这个信息。如果数据搬迁阶段，源节点发生故障，它的从节点自动提主，此时新源节点没有 MIGRATING 状态，它不知道这个 slot 在迁移。一旦有这个 slot 的命令打到新源节点且命令访问的 key 已搬迁到目标节点，它会按照这个 key 不存在进行处理，造成源和目标 key 冲突。后续即使 DBA 介入，继续或者中断流程，也很难决定如何合并或者覆盖这个 key ，可能造成数据丢失或不一致的严重问题。

此外，新源主节点原本会把 slot 归属权更新的消息传播给集群中所有的节点，但在目标节点的视角里，这个 slot 还处于 IMPORTING 状态，按照集群协议，它会忽略这个归属权更新。所以在目标节点的视角里，这个 slot 的 owner 始终是 fail 的老源节点，从而判定集群 fail ，所有打到目标节点的请求都会收到- CLUSTERDOWN 报错，业务请求受损。

c. 迁移状态乱序问题

社区方案里还有一点需要特别注意，迁移状态的标记和清理都需要遵守特定的顺序。

首先，标记阶段步骤1和2顺序不能换(参见图2)，必须是先目标节点后源节点。如果先给源节点设置了 MIGRATING 状态，一旦有这个 slot 的命令打到源节点且命令访问的 key 不存在，源节点就会返回 ASK 重定向，让客户端去请求目标节点；但此时目标节点因没有标记迁移状态，无法正常处理被重定向的请求，会给客户端返回 MOVED 重定向，又把请求踢回给源节点。二者来回踢皮球，直到目标节点设置了 IMPORTING 状态，或者客户端超时报错。

其次，转移阶段步骤6和7顺序也不能换(参见图2)，状态清理也必须是先目标节点后源节点。如果先清理源节点的 MIGRATING 状态，源节点就会认为这个 slot 永久归属为目标节点，一旦有这个 slot 的命令打到源节点，源节点会返回 MOVED 重定向，让客户端去请求目标节点；但 MOVED 重定向不会带 ASKING 命令，目标节点因为有 IMPORTING 状态，无法处理没前置 ASKING 命令的请求，也会返回 MOVED 重定向，又把请求转回给源节点。二者陷入重定向死循环，直到目标节点清理了 IMPORTING 状态，或者客户端超时报错。

那是不是发起迁移的人员或者工具遵守了特定的顺序就没有这个问题了呢？很遗憾，答案是否定的。原因是迁移过程中一旦有节点挂掉，结合上一点，迁移状态会丢失，此时只有一边有迁移状态，顺序的前提就被打破了。

（3）社区方案扩容速度慢带来的挑战

最后一点是扩容速度。前文提到，社区的 slot 迁移流程中，key 搬迁是最耗时的一个阶段。在不考虑大 key 的情况下，可以简单认为：搬迁耗时 = key 总数 / 搬迁命令 QPS 。

业务大多数情况下，是因为性能不足才发起水平扩容，而 slot 迁移时对业务请求的时延和 QPS 都有负面影响，简直是雪上加霜，所以业务肯定希望 slot 迁移越快完成越好；但我们也知道，Redis 的命令执行是单线程模型，为了让业务请求的性能受到的影响尽量小，搬迁命令的 QPS 肯定要做限制，限制太狠的话搬迁耗时又会过长，这就是一个经典的鱼和熊掌难题。

2 业界其他常见的 Redis 水平扩缩容方案

Redis 的水平扩缩容本身是很多业务的刚需，但是社区原生的方案又有很多的不足和痛点。为了在满足业务需求的同时能够规避社区方案的不足，业界也提

出了一些其他的方案。

2.1 基于旁路迁移的水平扩缩容方案

（1）旁路迁移扩缩容方案的实现原理

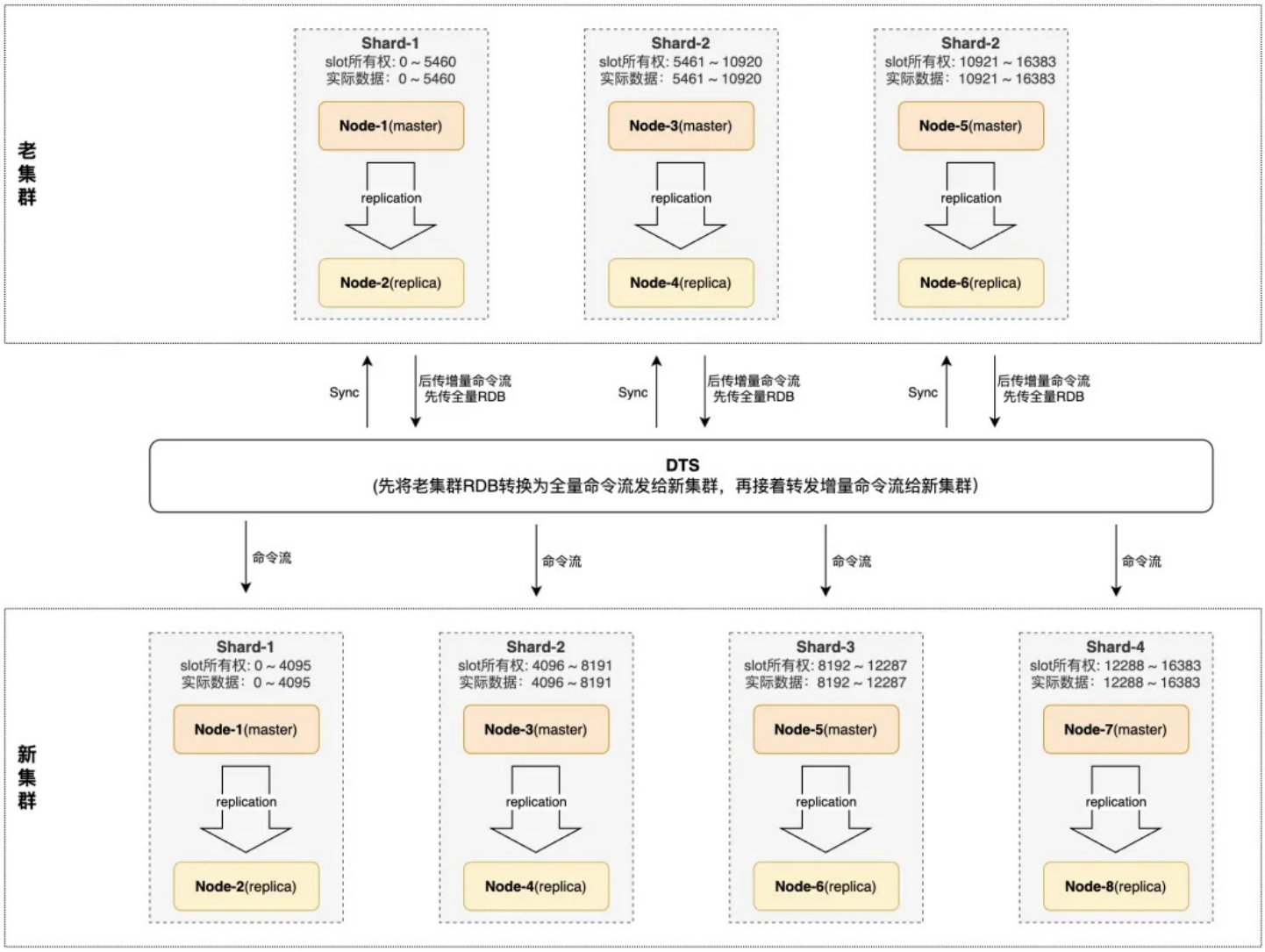
这个方案的思路比较简单，就是基于业务扩缩容的目标分片数创建一个新集群，然后借用旁路的数据传输组件，将老集群的数据传输给新集群，最后再找一个合适的时机，把流量也从老集群切到新集群。

这里提到的数据传输组件，业界有开源的 redis-shake，redis-port 等，各家云厂商也都有相应的 DTS (数据传输服务)可以用。

a. 数据传输

这些数据传输组件的原理都很类似，就是为老集群的每一个分片模拟一个从节点，通过 SYNC 或者是 PSYNC 命令从源端分片发起一次全量同步，然后收到全量的 RDB 文件(其中也带有源端节点中所有的 LUA 脚本)；然后借鉴 AOF 重写的机制把 RDB 转换成命令流发送给目标端的新集群；发完全量命令流后再继续把源端老集群的增量命令流也继续转发给新集群，维持数据同步状态稳定。

因为有着旁路组件在中间做“翻译官”，发给新集群的都是命令流，新集群的分片数就和老集群完全解耦了，只要数据能放得下，比老集群分片数多点少点都OK。以一个3分片集群扩为4分片集群为例，数据同步的示意如图5所示：



▲图5. 基于 DTS 的集群水平扩缩容方案原理

b. 流量切换

两个集群之间进入稳定的数据同步状态之后，就可以准备切流量了。一般是先要观察是不是所有子任务(老集群的每个分片对应一个子任务)的同步延迟都在一个可接受的阈值内；如果满足条件，就让业务将老集群的写流量停掉，等待新集群的数据完全和老集群追齐，因为这是旁路的数据传输组件，这个等待时间往往是分钟级的；接着再把业务的读写流量完全重定向到新集群；最后释放老集群的资源。至此，一次扩缩容任务就算完成了。

（2）旁路迁移扩缩容方案和 Redis 社区原生案的对比

对比 Redis 社区原生的水平扩缩容方案，这个方案的优点很明显：

- 切流前，请求全部访问老集群，老集群中始终有全量的 key 和 LUA 脚本；切流后，全部请求转为访问新集群中，新集群中也已有全量的 key 和 LUA

脚本；切流前后，业务的多 key 请求和 LUA 请求都能正常处理。

- 切流前，仅相当于老集群中每个分片多挂了一个从节点，全量阶段由子进程生成 RDB ，完全不影响主进程的命令处理，序列化大 key 也不阻塞；增量阶段，只需把写命令转发一份给数据传输组件，损耗也很小；而对于新集群，本身不处理业务请求，回放数据同步的写命令对业务不影响。
- 过程中，没有 ASK 重定向，业务请求不会因增加网络跳数而时延上升。
- 过程中，没有迁移状态，无论是老集群还是新集群有节点发生故障，正常将从节点提主就好，业务访问可以保持高可用；不用关心旁路组件的状态，由旁路组件自行检查数据同步状态并恢复。
- 过程中，只要没有切流，随时可以中断，断掉旁路组件和老集群的数据同步即可。
- 迁移过程中新集群完全不处理业务请求，可以全速完成数据同步，迁移速度会更快。

相应的，这个方案也有一些缺点。一方面，为了保障新老集群之间的数据一致性，在切流前需要老集群停写几十秒到几分钟，这个对于部分在线业务是难以接受的；另一方面，在迁移过程中，需要同时提供新老两个集群的机器资源，这个成本压力太大了。

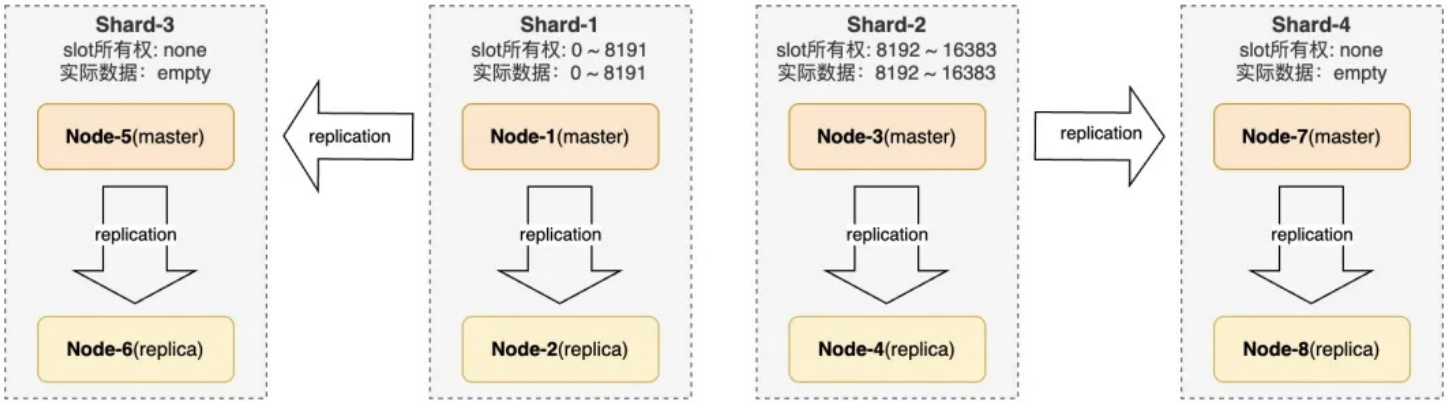
2.2 基于节点分裂的水平扩容方案

（1）节点分裂扩容方案的实现原理

这是一个很有特点的方案，有的地方也称之为分裂式扩容，或者翻倍式扩容。它的思路也不复杂，大致流程就分三步：节点复制，归属权分裂，数据清理。下面我们以一个2分片集群翻倍扩为4分片集群举例，分步骤详细说明。

a. 节点复制

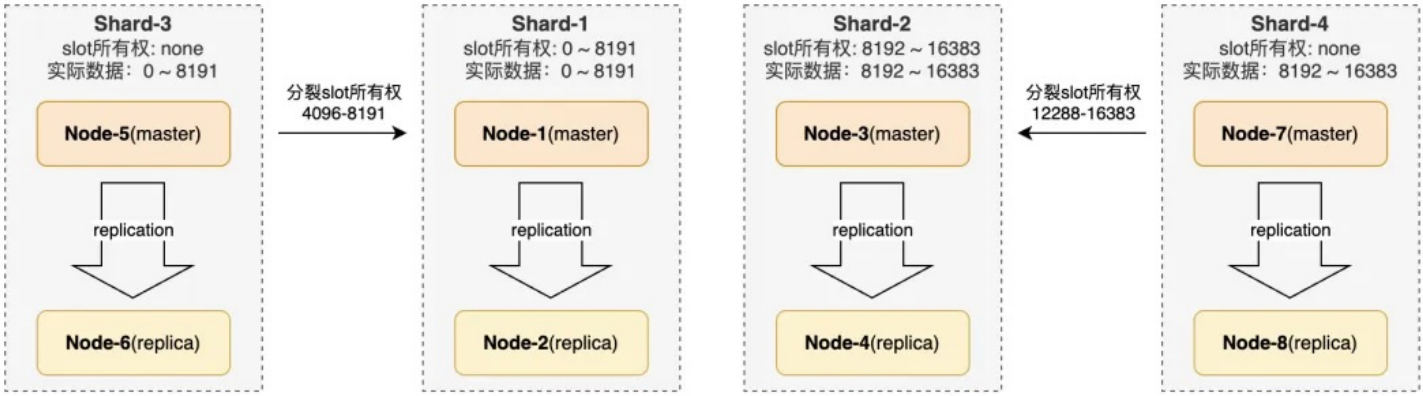
为集群中现有的每个老分片分别创建对应的新分片，因为这里是扩一倍，所以每个老分片只对应一个新分片，如果是扩其它倍数，这里就需要创建对应倍数的新分片。新分片中的节点此时没有数据，也不负责任何 slot ，将每个新分片中的主节点以挂从的方式连上对应老分片的主节点，直接复用 Redis 主从复制的流程，将老分片的数据(也包含所有的 LUA 脚本)全量同步到对应的新分片，并维持增量传播状态，过程如图6-1所示。



▲图6-1. 节点分裂扩容的原理 -- 节点复制

b. 归属权分裂

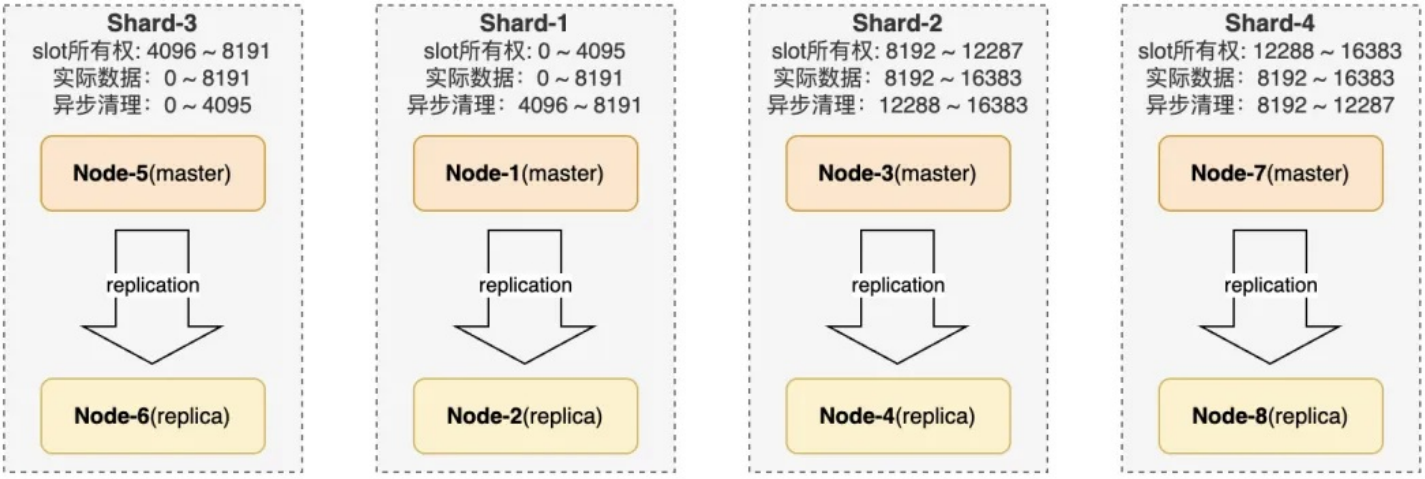
节点复制完成后，两组对应的新老分片之间数据基本是一模一样的(只是增量写入部分新分片有一些滞后)，只是新分片还不负责任何 slot 。此时，每个新分片依次向对应的老分片发起分裂一半 slot 归属权的请求，如图6-2所示。归属权分裂的底层实现方式有很多种，比较优雅的一种是借鉴 Redis 自身的 manual failover 实现机制，区别只是说，分裂只获取对应老分片一部分 slot 的归属权，分裂后老分片的主节点角色还是 master 。



▲图6-2. 节点分裂扩容的原理 -- 归属权分裂

c. 数据清理

上一步完成后，对于每一组的新老分片而言，它们各自获得老分片原先一半 slot 的归属权，但是新老分片都还有老分片原先的所有数据。换言之，新老分片都有一半的数据其实已经没用了，但还占用着内存空间。此时新老分片需要各自发起一个异步的后台清理任务，将已经不属于自己负责的 slot 数据清理掉，如图6-3所示。等清理任务结束，本次扩容任务就完成了。



▲图6-3. 节点分裂扩容的原理 -- 数据清理

（2）节点分裂扩容方案和 Redis 社区原生案的对比

对比 Redis 社区原生的水平扩缩容方案，这个方案也有类似旁路迁移方案的一些优点：

- 分裂前，所有请求访问老分片，老分片上也有全量的 key 和 LUA 脚本；分裂后，一半请求转为访问新分片，新分片上也已经有对应 slot 的全量 key 和 LUA 脚本；分裂前后，业务的多 key 请求和 LUA 请求都能正常处理。
- 复制阶段，每个老分片相当于多挂了N个从节点(N取决于扩容的倍数)，全量阶段由子进程生成 RDB，完全不影响主进程的命令处理，序列化大 key 也不阻塞；增量阶段只需把写命令转发给新分片，N不大的情况下损耗也小；而对于新分片来说，完全不处理业务请求，加载 RDB 和回放增量命令都不影响业务请求。
- 分裂阶段，为了保障新老分片完全一致，需要临时阻塞老分片，但它的时间量级和 manual failover 相近，只是亚秒级的时延抖动。
- 清理阶段，后台清理任务会占用主线程一些资源，不过好在此时集群总处理能力已提升，清理只是为了释放内存空间，稍微慢一点也没事，可以限速严格一些，并且 Redis 4.0 的 lazy free 特性也能减少清理过程对主线程的占用。
- 过程中，没有 ASK 重定向，业务请求不会因增加网络跳数而时延上升。
- 过程中，没有迁移状态，老分片节点异常，正常将从节点提主就好，业务请求可以保持高可用；新分片节点异常更简单，拉起新的空节点，重新发起数据同步即可。
- 过程中，只要没有分裂归属权，随时可以中断，断掉新分片和老分片的数据同步即可。

- 复制过程中新分片完全不处理业务请求，可以全速完成数据同步，速度会更快。

但是，这个方案相比社区方案的缺点也很明显。首先，它的灵活性很差，只能翻倍扩，这个对于本身就已经有几十上百个分片的集群来说就很难接受了，成本一次性上升太多了；而且它还只能扩不能缩，对一些流量弹性很大的业务来说也很难接受。其次，当扩的倍数很大时，对老分片来说，挂的从节点就太多了，这个对老分片的处理能力会有一些损耗。

3 腾讯云Redis基于slot原子化迁移的水平扩缩容方案

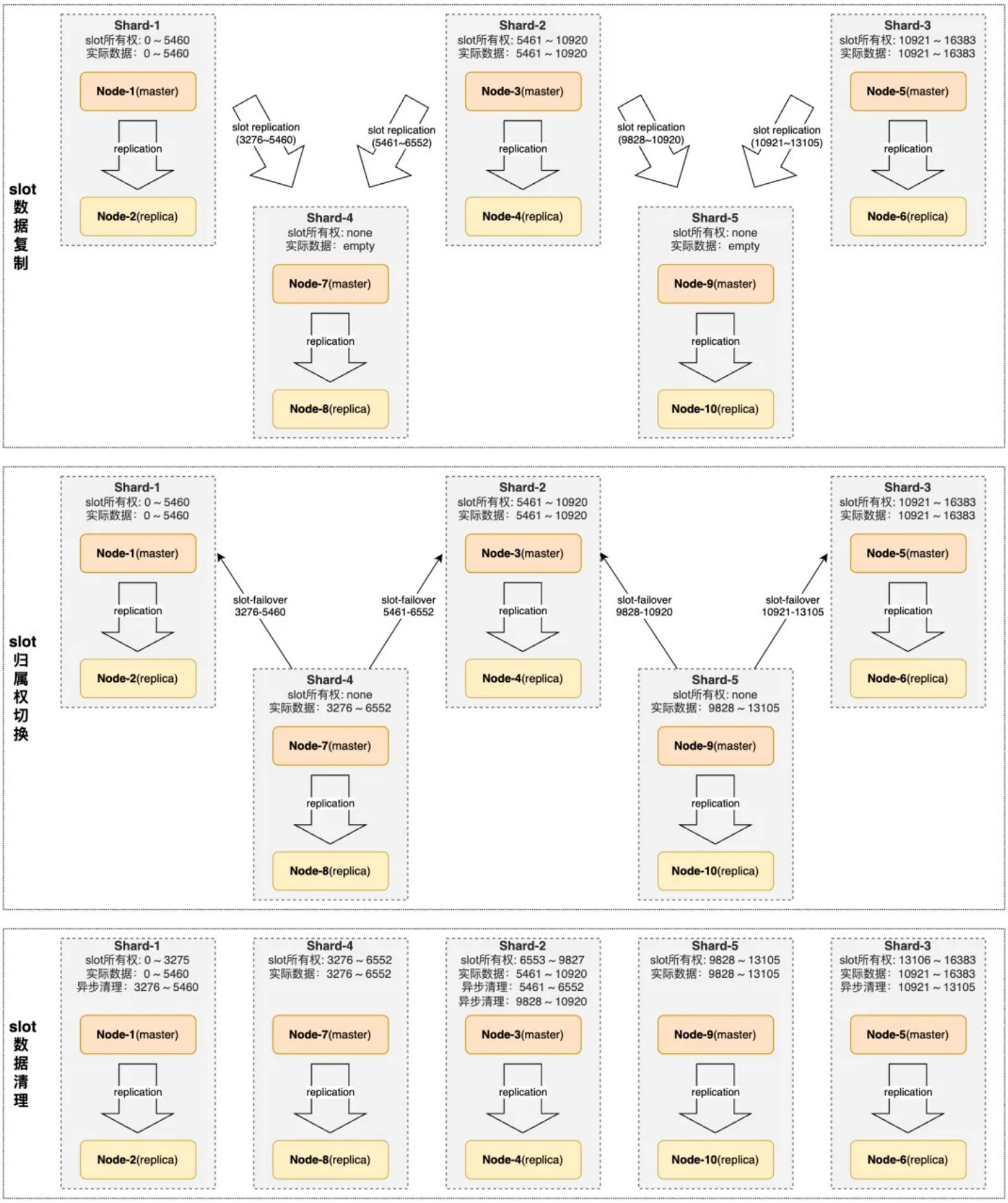
当我们多年前打算在公有云上售卖 Redis 托管服务的时候，摆在我们面前的就是这样的现状： Redis 社区提供的原生水平扩缩容方案槽点多多，运营难度高；旁路迁移方案的切流停写许多客户难以接受，迁移的额外成本对我们平台方来说负担也很重；节点分裂方案其他点都还不错，就是目标分片数限制太死，灵活性太差，使用场景严重受限。

云上运营需要一个新的方案，这个方案应该对业务请求的影响尽量少，本身足够健壮，扩缩容的速度要够快，对目标分片数不做限制，不需要额外的成本，并且尽量不依赖三方组件。基于这样的目标，并吸收了业界已有方案的精髓，我们最终推陈出新，实现了基于 slot 原子化迁移的全新水平扩缩容方案。

3.1 腾讯云 slot 原子化迁移方案的原理和实现

我们的方案也是通过在节点之间迁移 slot 来达到目的，但和社区原生方案不同的是，我们在 slot 迁移的时候，搬迁数据不是以 key 的粒度逐步搬迁，而是以 slot 为最小粒度整体做搬迁，搬迁具备原子性；另外搬迁过程不是同步阻塞的，而是在 fork 的子进程中异步执行。

我们的方案核心流程分为三个阶段：数据复制，归属权切换，数据清理，如图7所示。但为了保障方案的成功率，我们还专门在正式流程发起前，加了一个容量估算的阶段。



▲图7. 腾讯云 slot 原子化迁移方案 -- 核心流程

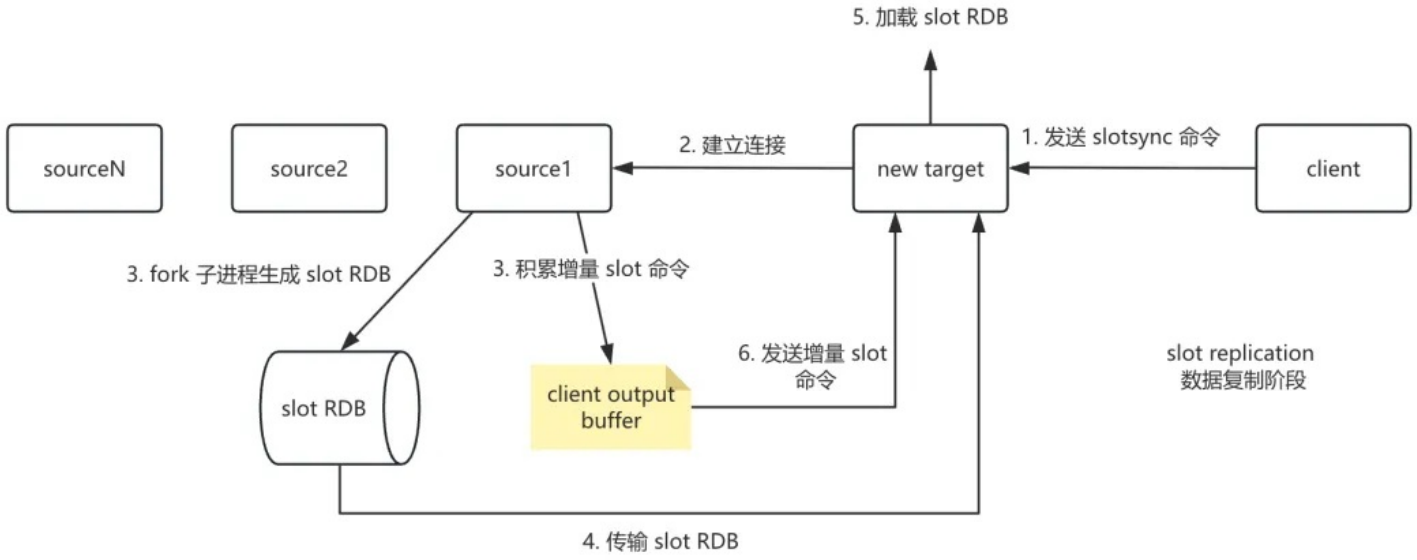
a. 容量估算

所谓容量估算，就是预先对要搬迁的 slot 所占据的内存容量进行估算。有了这一步，就不会等搬迁进行到一半的时候才突然发现目标节点根本放不下这些 slot，就能避免回滚等额外开销。对于每个 slot 的数据分布非常均匀的业务来说，这个步骤是没必要的，直接用节点总容量除以 slot 数即可得到 slot 的大致容量；但因为复杂结构的存在，对很多业务来说，不同 key 的大小差异可能是很巨大的，所以一个更细粒度的 slot 容量估算方案就很必要了。

我们的容量估算实现也很简单，客户端通过 STARTSLOTCALC 命令发起一个异步的计算任务，它会在 Redis 每次运行周期任务的时候，分出一些时间片，根据 slots_to_keys 渐进地遍历指定 slot 中的每个 key ，累加它们的长度。遍历中，对于 string 类型， value 长度可直接获取；而对于复杂结构类型，会通过类似MEMORY USAGE底层的实现方式，基于采样估算 value 的总长度。在此过程中，客户端会通过 GETSLOTCALCSTAT 命令轮询，获取估算的进度。

b. 数据复制

这里的数据复制，只是针对特定 slot ，将这些 slot 的数据从源节点搬迁到目标节点，这也是我们的方案和前面两个业界方案相比起来最大的不同。在这个阶段，目标节点会针对指定的 slot 范围(最少1个)，发起 slot replication 流程。其主要流程如图8-1所示：



▲图8-1. 腾讯云 slot 原子化迁移方案原理 -- slot replication

为了减少开发代价，并减少后续版本的移植难度，我们的 slot replication 大量复用了 Redis 原生 replication 机制的代码实现，并主要做了如下改动：

1. 目标节点发送给源节点的是改造过的 SYNC 命令，可以指定slot范围。

2. 源节点的复制连接对象中增加了相应的字段，保存指定的slot范围。

3. 源节点 fork 生成 RDB 的时候，rdbSaveInfo 也会携带 slot 范围信息，这个关键信息会让子进程遍历数据的时候，由遍历主字典改为遍历特定 slot 的 slot_to_keys 结构，让 RDB 中只有指定 slot 范围的数据，另外还会附带源节点上所有的 LUA 脚本，这个特殊的RDB我们称之为 slot RDB 。

4. 源节点向目标节点传播增量命令的时候，也会基于检查复制连接对象中的上下文，如果是指定了 slot 范围，那就基于命令的 key 做过滤，只传播指定 slot 范围的增量命令。

5. 目标节点可能要加载来自不同源节点的多个 slot RDB ，所以加载前不会清空已有数据。

c. 归属权切换

这里的归属权切换，是将这些已搬迁数据的 slot 的归属权从源节点切换到目标节点。这里额外补充一句，在这个切换操作之前，源节点上是一直有这些 slot 的全量数据的，业务对这些 slot 的访问全部打向源节点即可，完全不需要类似 ASK 这种临时重定向机制。

为了满足目标节点和源节点之间完全的数据一致性，我们借鉴 Redis 原生的 manual failover 机制实现了针对 slot 归属权切换场景的 slot failover 机制，其主要流程如图8-2所示：

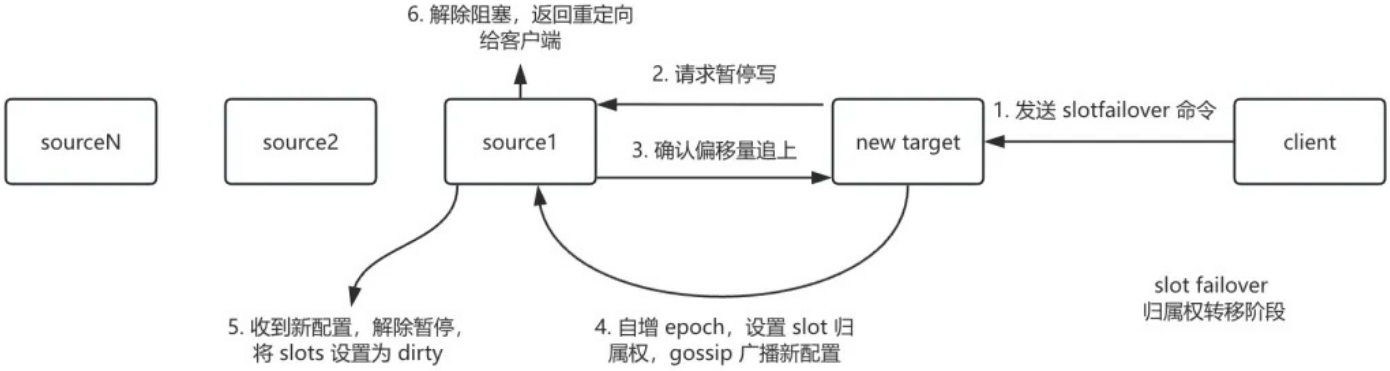
1. 等待目标节点和源节点的差异足够小，给目标节点发送 CLUSTER SLOTFAILOVER 命令，发起切换任务；

2. 目标节点给源节点发起切换请求；

3. 源节点临时阻塞所有客户端，将复制偏移量发给目标节点；

4. 目标节点确认自身复制偏移量已和源节点对齐，先自增 epoch ，设置指定 slot 的归属权属于自己，并广播通知其余节点；

5. 源节点收到 slot 归属权已切换的通知，解除阻塞，将指定 slot 的请求重定向到目标节点，只处理剩余的那些归属权仍为自身的 slot 请求。



▲图8-2. 腾讯云 slot 原子化迁移方案原理 -- slot failover

可以看到，slot failover 和 Redis 原生的 manual failover 很像，区别在于 slot failover 只获取源节点一部分 slot 的归属权，源节点角色还是 master ，仍旧负责剩余的 slot 。

d. 数据清理

这里为啥还需要做数据清理呢？原因在于，这些 slot 的归属权切换到目标节点后，源节点上这些slot的数据就变成无效的了，源节点上这些 slot 会被标记为 dirty ，放在一个队列里，这些 dirtyslot 的数据需要清理以释放源节点的空间。

我们在 Redis 的 serverCron 中加了一个后台任务，它会周期运行，每次取出 dirty slot 队列头部的 slot ，根据 slots_to_keys 渐进地摘除其中的 key ，实际删 key 的时候，对于小 key 会直接删除，但对于大 key 会调用 lazy free 的相关接口，转到 BIO 线程去实际释放内存，避免阻塞业务请求。当一个 dirty slot 中的所有key都被删除后，清理任务会把它从 dirty slot 队列中移除，下个周期继续处理下一个 slot ，直到 dirty slot 队列为空。为了减少清理过程对业务请求的影响，每个周期我们都会严格限制清理任务占用的时间片。

3.2 腾讯云 slot 原子化迁移方案与其他方案的对比

在上一节中，我们详细介绍了云上自研的 Redisslot 原子化迁移方案，一个对业务无损且灵活的方案。相比社区原生方案和业界的另外两个方案，我们的完美解决了它们的所有痛点，在各个维度上的表现都堪称优秀！

- 在业务可用性影响方面，源节点的数据扫描和序列化都在 fork 的子进程中进行，没有大 key 阻塞问题；slot RDB 中会携带LUA脚本，不会有 LUA 请求异常； slot 数据搬迁是原子化的，不会有多 key 请求异常； slot 所有权切换的时候，只需要一个类似主从切换的亚秒级阻塞，不需要分钟级停写(旁路迁移方案需要)。

- 在业务性能影响方面，全程没有 ASK 重定向，不会有网络跳数增加带来的时延增大问题； slot 所有权切换前，业务请求都访问源节点，而源节点主线程不管数据搬迁，全力处理业务请求， QPS 不受影响； slot 所有权切换后，源节点会有异步清理任务，但因为降内存没有升性能紧急，限流可以做得更严格，另外也会用 lazy free 功能规避大 key 删除的卡顿，整体影响可控。

- 在方案的健壮性方面，源节点或目标节点始终有全量的数据，中断处理方便；没有那么复杂的迁移状态，没有状态丢失问题和状态乱序问题。

- 在方案的灵活性方面，只要目标节点内存容量放得下，目标分片数没有任何限制。

- 在方案的速度方面，因为数据搬迁是整体打包成 slot RDB 传输，且搬迁中目标节点不处理业务请求，可以全力加载 slot RDB 与回放增量 slot 命令，速度比起逐个搬 key 会快很多；而且相比节点分裂方案，只传输需要迁移 slot 的数据，数据量少速度会更快。

- 在方案的额外成本方面，客户扩容需要扩几个节点，我们的方案就新拉起几个节点，不用像旁路迁移方案一样同时维护新老两个集群，没有额外机器成本。

- 在方案的外部依赖方面，我们的方案内核自闭环，不会引入外部组件，架构复杂度低。

我们把云上自研方案和业界其他几个方案的详细对比结果总结到了一张表，如表1所示：

	业务请求影响		健壮性	灵活性	扩缩容速度	额外成本低	外部依赖少
	可用性	性能					
社区原生方案	★	★	★	★★★	★	★★★	★★★
旁路迁移方案	★★	★★★	★★★	★★★	★★★	★	★
节点分裂方案	★★★	★★	★★★	★	★★	★★★	★★★
腾讯云优化方案	★★★	★★	★★★	★★★	★★☆	★★★	★★★

表1. 腾讯云 slot 原子化迁移方案与业界其他方案的对比

3.3 腾讯云将 slot 原子化迁移方案贡献给开源社区

我们自研的 slot 原子化迁移方案，解决了长期以来困扰 Redis 用户的水平扩缩容难题，实现了真正意义上的平滑伸缩。该方案自上线后，已广泛服务于腾讯集团的内部客户和公有云上的外部客户，经历了大规模实践的检验，稳定性有保障。

作为开源社区的受益者之一，我们深知众人拾柴火焰高的道理，开源社区的繁荣依赖于生态中的每一个人和组织能够积极参与。我们团队的朱彬彬同学(朱彬彬同学 GitHub 主页：https://github.com/enjoy-binbin) 连续多年在Redis项目踊跃提交代码，长期都是Redis社区最活跃的贡献者之一，也是 Redis 项目的 committer 成员；在2024年 Redis 修改开源许可证后，我们也是加入了linux基金会发起的替代项目 ValKey ，是新项目的发起团队之一，彬彬同学目前也是 ValKey 项目的 core team 成员。

早在24年规划的 ValKey9.0 road map 中，我们就开始推动社区增加slot原子化迁移的相关功能，并在今年年初，我们将内部方案开源，并联合谷歌云(GCP)的同学一起合作，完善这个方案以适配社区相对云上不一样的要求。

目前，这个PR (https://github.com/valkey-io/valkey/pull/1949) 已经合入 Valkey 代码主干，并作为Valkey9.0最重要的功能优化之一，随着 RC1 于今年8.15正式面向所有用户发布。我们也希望，有更多的用户可以在新版本中体验到这一改进，让 Redis 的水平扩缩容难题不再成为业务发展的瓶颈！

前往查看

redis 集群 nosql 迁移 腾讯云

redis 集群 nosql 迁移 腾讯云