16 张图吃透 Redis 架构演进

ImportNew 2021-09-08 20:30

The following article is from 水滴与银弹 Author Magic Kaito

水滴与银弹

7年资深后端，擅长Redis、基础架构、中间件、异地多活，QConf+合作讲师，只写硬核、…

现如今 Redis 变得越来越流行，几乎在很多项目中被用到。不知道大家有没有思考过：**Redis 到底是如何稳定、高性能地提供服务的?**

你可以先尝试回答一下这些问题：

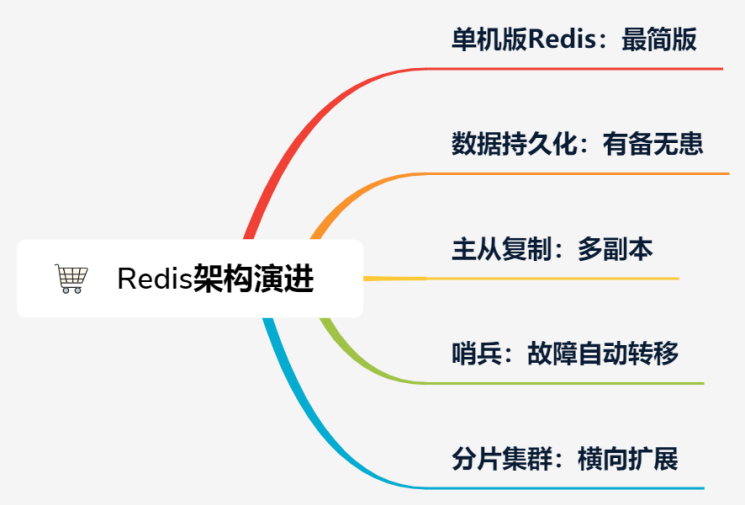
- 我使用 Redis 的场景很简单，只使用单机版 Redis 会有什么问题吗?
- 我的 Redis 故障宕机了，数据丢失了怎么办？如何能保证我的业务应用不受影响?
- 为什么需要主从集群？它有什么优势?
- 什么是分片集群？我真的需要分片集群吗?
- ...

此外，如果你对 Redis 已经有所了解，肯定也听说过**数据持久化、主从复制、哨兵**这些概念，它们之间又有什么区别和联系呢?

如果你存在这样的疑惑，这篇文章，我将带你掌握：**如何从 0 到 1，再从 1 到 N，一步步构建出一个稳定、高性能的 Redis 集群?**

在这个过程中，你可以了解到 Redis 为了做到稳定、高性能，都采取了哪些优化方案，以及为什么要这么做?

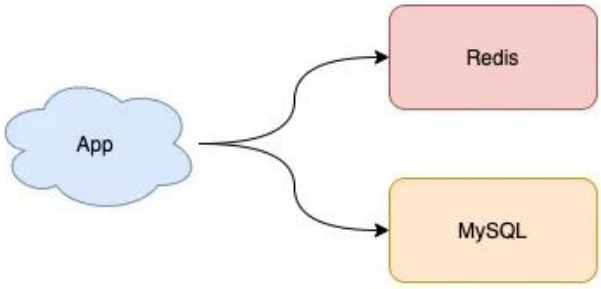
掌握了这些原理，这样平时你在使用 Redis 时，就能够做到「游刃有余」。



01 从最简单的开始：单机版 Redis

首先，我们从最简单的场景开始。

假设现在你有一个业务应用，需要引入 Redis 来提高应用的性能，此时你可以选择部署一个单机版的 Redis 来使用，就像这样：

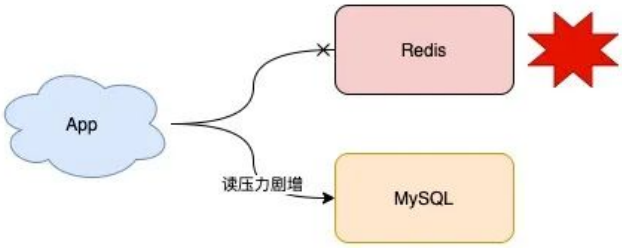


这个架构非常简单，你的业务应用可以把 Redis 当做缓存来使用，从 MySQL 中查询数据，然后写入到 Redis 中，之后业务应用再从 Redis 中读取这些数据，由于 Redis 的数据都存储在内存中，所以这个速度飞快。

如果你的业务体量并不大，那这样的架构模型基本可以满足你的需求。是不是很简单？

随着时间的推移，你的业务体量逐渐发展起来了，Redis 中存储的数据也越来越多，此时你的业务应用对 Redis 的依赖也越来越重。

但是，突然有一天，你的 Redis 因为某些原因宕机了，这时你的所有业务流量，都会打到后端 MySQL 上，这会导致你的 MySQL 压力剧增，严重的话甚至会压垮 MySQL。



这时你应该怎么办？

我猜你的方案肯定是，赶紧重启 Redis，让它可以继续提供服务。

但是，因为之前 Redis 中的数据都在内存中，尽管你现在把 Redis 重启了，之前的数据也都丢失了。重启后的 Redis 虽然可以正常工作，但是由于 Redis 中没有任何数据，业务流量还是都会打到后端 MySQL 上，MySQL 的压力还是很大。

这可怎么办？你陷入了沉思。

有没有什么好的办法解决这个问题？

既然 Redis 只把数据存储在内存中，那是否可以把这些数据也写一份到磁盘上呢？

如果采用这种方式，当 Redis 重启时，我们把磁盘中的数据快速**恢复**到内存中，这样它就可以继续正常提供服务了。

是的，这是一个很好的解决方案，这个把内存数据写到磁盘上的过程，就是「数据持久化」。

02 数据持久化：有备无患

现在，你设想的 Redis 数据持久化是这样的：

但是，数据持久化具体应该怎么做呢？

我猜你最容易想到的一个方案是，Redis 每一次执行写操作，除了写内存之外，同时也写一份到磁盘上，就像这样：

没错，这是最简单直接的方案。

但仔细想一下，这个方案有个问题：客户端的每次写操作，既需要写内存，又需要写磁盘，而写磁盘的耗时相比于写内存来说，肯定要慢很多！这势必会影响到 Redis 的性能。

如何规避这个问题？

我们可以这样优化：Redis 写内存由主线程来做，写内存完成后就给客户端返回结果，然后 Redis 用另一个线程去写磁盘，这样就可以避免主线程写磁盘对性能的影响。

这确实是一个好方案。除此之外，我们可以换个角度，思考一下还有什么方式可以持久化数据？

这时你就要结合 Redis 的使用场景来考虑了。

回忆一下，我们在使用 Redis 时，通常把它用作什么场景？

是的，缓存。

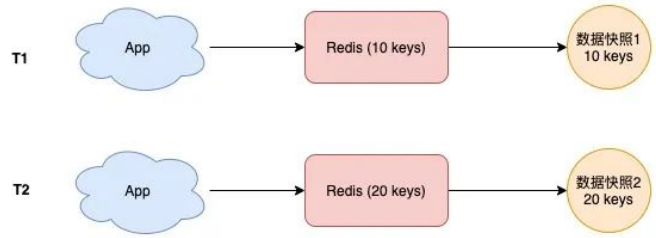
把 Redis 当做缓存来用，意味着尽管 Redis 中没有保存全量数据，对于不在缓存中的数据，我们的业务应用依旧可以通过查询后端数据库得到结果，只不过查询后端数据的速度会慢一点而已，但对业务结果其实是没有影响的。

基于这个特点，我们的 Redis 数据持久化还可以用「数据快照」的方式来做。

那什么是数据快照呢？

简单来讲，你可以这么理解：

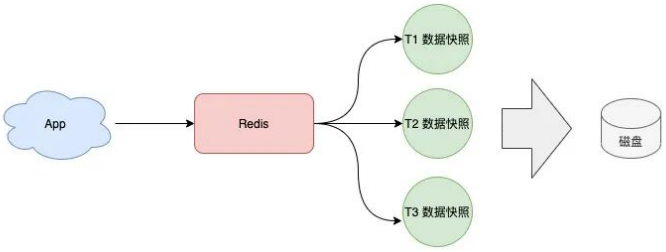
> 1、你把 Redis 想象成一个水杯，向 Redis 写入数据，就相当于往这个杯子里倒水。
>
> 2、此时你拿一个相机给这个水杯拍一张照片，拍照的这一瞬间，照片中记录到这个水杯中水的容量，就是水杯的数据快照。



也就是说，Redis 的数据快照，是记录某一时刻下 Redis 中的数据，然后只需要把这个数据快照写到磁盘上就可以了。

它的优势在于，只在需要持久化时，把数据「一次性」写入磁盘，其它时间都不需要操作磁盘。

基于这个方案，我们可以**定时**给 Redis 做数据快照，把数据持久化到磁盘上。



其实，上面说的这些持久化方案，就是 Redis 的「RDB」和「AOF」：

- RDB：只持久化某一时刻的数据快照到磁盘上（创建一个子进程来做）
- AOF：每一次写操作都持久到磁盘（主线程写内存，根据策略可以配置由主线程还是子线程进行数据持久化）

它们的区别除了上面讲到的，还有以下特点：

> 1、RDB 采用二进制 + 数据压缩的方式写磁盘，这样文件体积小，数据恢复速度也快。
>
> 2、AOF 记录的是每一次写命令，数据最全，但文件体积大，数据恢复速度慢。

如果让你来选择持久化方案，你可以这样选择：

> 1、如果你的业务对于数据丢失不敏感，采用 RDB 方案持久化数据。
>
> 2、如果你的业务对数据完整性要求比较高，采用 AOF 方案持久化数据。

假设你的业务对 Redis 数据完整性要求比较高，选择了 AOF 方案，那此时你又会遇到这些问题：

> 1、AOF 记录每一次写操作，随着时间增长，AOF 文件体积会越来越大。
>
> 2、这么大的 AOF 文件，在数据恢复时变得非常慢。

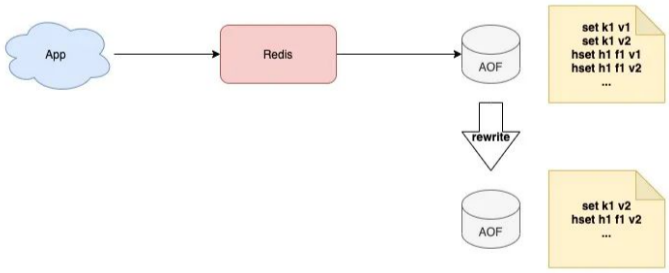
这怎么办？数据完整性要求变高了，恢复数据也变困难了？有没有什么方法，可以缩小文件体积？提升恢复速度呢？

我们继续来分析 AOF 的特点。

由于 AOF 文件中记录的都是每一次写操作，但对于同一个 key 可能会发生多次修改，我们只保留最后一次被修改的值，是不是也可以？

是的，这就是我们经常听到的「AOF rewrite」，你也可以把它理解为 AOF 「瘦身」。

我们可以对 AOF 文件定时 rewrite，避免这个文件体积持续膨胀，这样在恢复时就可以缩短恢复时间了。



再进一步思考一下，还有没有办法继续缩小 AOF 文件？

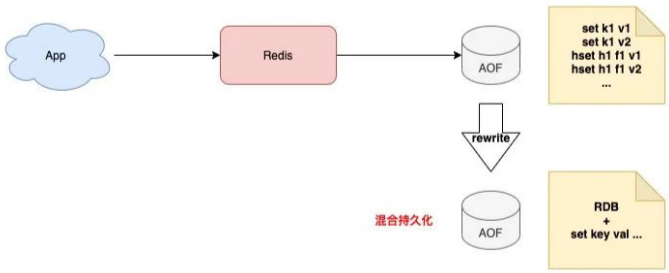
回顾一下前面讲到的，RDB 和 AOF 各自的特点：

> 1、RDB 以二进制 + 数据压缩方式存储，文件体积小。
>
> 2、AOF 记录每一次写命令，数据最全。

我们可否利用它们各自的优势呢？

当然可以，这就是 Redis 的「混合持久化」。

具体来说，当 AOF rewrite 时，Redis 先以 RDB 格式在 AOF 文件中写入一个数据快照，再把在这期间产生的每一个写命令，追加到 AOF 文件中。因为 RDB 是二进制压缩写入的，这样 AOF 文件体积就变得更小了。



此时，你在使用 AOF 文件恢复数据时，这个恢复时间就会更短了！

> Redis 4.0 以上版本才支持混合持久化。

这么一番优化，你的 Redis 再也不用担心实例宕机了，当发生宕机时，你就可以用持久化文件快速恢复 Redis 中的数据。

但这样就没问题了吗？

仔细想一下，虽然我们已经把持久化的文件优化到最小了，但在恢复数据时依旧是需要时间的，在这期间你的业务应用还是会受到影响，这怎么办？

我们来分析有没有更好的方案。

一个实例宕机，只能用恢复数据来解决，那我们是否可以部署多个 Redis 实例，然后让这些实例数据保持实时同步，这样当一个实例宕机时，我们在剩下的实例中选择一个继续提供服务就好了。

没错，这个方案就是接下来要讲的「主从复制：多副本」。

03 主从复制：多副本

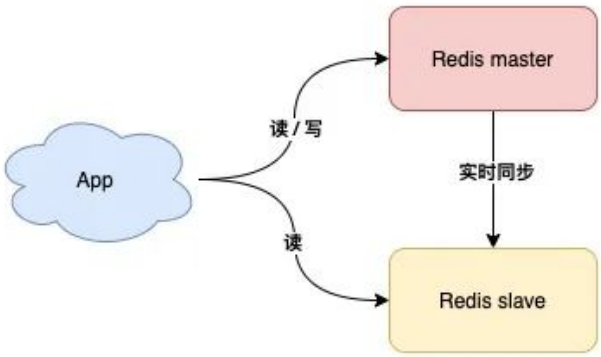
此时，你可以部署多个 Redis 实例，架构模型就变成了这样：

我们这里把实时读写的节点叫做 master，另一个实时同步数据的节点叫做 slave。

采用多副本的方案，它的优势是：

1、缩短不可用时间：master 发生宕机，我们可以手动把 slave 提升为 master 继续提供服务。

2、提升读性能：让 slave 分担一部分读请求，提升应用的整体性能。



这个方案不错，不仅节省了数据恢复的时间，还能提升性能，那它有什么问题吗？

你可以思考一下。

其实，它的问题在于：当 master 宕机时，我们需要「手动」把 slave 提升为 master，这个过程也是需要花费时间的。

虽然比恢复数据要快得多，但还是需要人工介入处理。一旦需要人工介入，就必须要算上人的反应时间、操作时间，所以，在这期间你的业务应用依旧会受到影响。

怎么解决这个问题？我们是否可以把这个切换的过程，变成自动化呢？

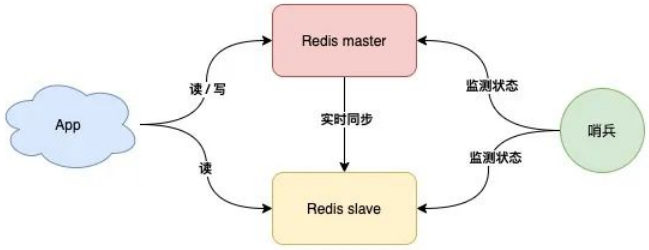
对于这种情况，我们需要一个「故障自动切换」机制，这就是我们经常听到的「哨兵」所具备的能力。

04 哨兵：故障自动切换

现在，我们可以引入一个「观察者」，让这个观察者去实时监测 master 的健康状态，这个观察者就是「哨兵」。

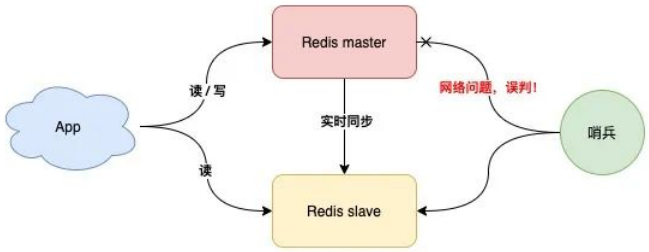
具体如何做?

> 1、哨兵每隔一段时间询问 master 是否正常。
>
> 2、master 正常回复，表示状态正常，回复超时表示异常。
>
> 3、哨兵发现异常，发起主从切换。



有了这个方案，就不需要人去介入处理了，一切就变得自动化了，是不是很爽?

但这里还有一个问题，如果 master 状态正常，但这个哨兵在询问 master 时，它们之间的网络发生了问题，那这个哨兵可能会误判。



这个问题怎么解决?

答案是，我们可以部署多个哨兵，让它们分布在不同的机器上，它们一起监测 master 的状态，流程就变成了这样：

> 1、哨兵每隔一段时间询问 master 是否正常。

> 2、master 正常回复，表示状态正常，回复超时表示异常。
>
> 3、一旦有一个哨兵判定 master 异常（不管是否是网络问题），就询问其它哨兵，如果多个哨兵（设置一个阈值）都认为 master 异常了，这才判定 master 确实发生了故障。
>
> 4、多个哨兵经过协商后，判定 master 故障，则发起主从切换。

所以，我们用多个哨兵互相协商来判定 master 的状态，这样一来，就可以大大降低误判的概率。

哨兵协商判定 master 异常后，这里还有一个问题：**由哪个哨兵来发起主从切换呢？**

答案是，选出一个哨兵「领导者」，由这个领导者进行主从切换。

问题又来了，这个领导者怎么选？

想象一下，在现实生活中，选举是怎么做的？

是的，投票。

在选举哨兵领导者时，我们可以制定这样一个选举规则：

> 1、每个哨兵都询问其它哨兵，请求对方为自己投票。
>
> 2、每个哨兵只投票给第一个请求投票的哨兵，且只能投票一次。
>
> 3、首先拿到超过半数投票的哨兵，当选为领导者，发起主从切换。

其实，这个选举的过程就是我们经常听到的：分布式系统领域中的「共识算法」。

什么是共识算法？

我们在多个机器部署哨兵，它们需要共同协作完成一项任务，所以它们就组成了一个「分布式系统」。

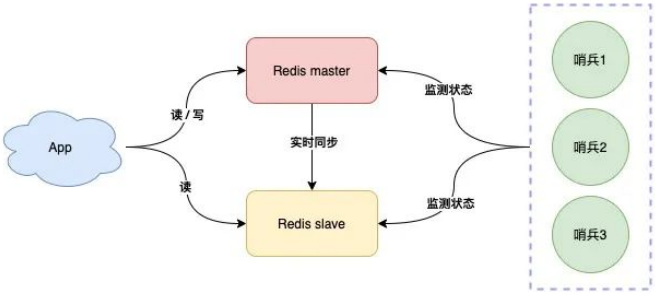
在分布式系统领域，多个节点如何就一个问题达成共识的算法，就叫共识算法。

在这个场景下，多个哨兵共同协商，选举出一个都认可的领导者，就是使用共识算法完成的。

这个算法还规定节点的数量必须是奇数个，这样可以保证系统中即使有节点发生了故障，剩余超过「半数」的节点状态正常，依旧可以提供正确的结果，也就是说，这个算法还兼容了存在故障节点的情况。

> 共识算法在分布式系统领域有很多，例如 Paxos、Raft，哨兵选举领导者这个场景，使用的是 Raft 共识算法，因为它足够简单，且易于实现。

现在，我们用多个哨兵共同监测 Redis 的状态，这样一来，就可以避免误判的问题了，架构模型就变成了这样：



好了，到这里我们先小结一下。

你的 Redis 从最简单的单机版，经过数据持久化、主从多副本、哨兵集群，这一路优化下来，你的 Redis 不管是性能还是稳定性，都越来越高，就算节点发生故障，也不用担心了。

你的 Redis 以这样的架构模式部署，基本上就可以稳定运行很长时间了。

...

随着时间的发展，你的业务体量开始迎来了爆炸性增长，此时你的架构模型，还能够承担这么大的流量吗？

我们一起来分析一下：

> 1、稳定性：Redis 故障宕机，我们有哨兵 + 副本，可以自动完成主从切换
>
> 2、读性能：读请求量增长，我们可以再部署多个 slave，读写分离，分担读压力
>
> 3、写性能：写请求量增长，但我们只有一个 master 实例，这个实例达到瓶颈怎么办？

看到了么，当你的写请求量越来越大时，一个 master 实例可能就无法承担这么大的写流量了。

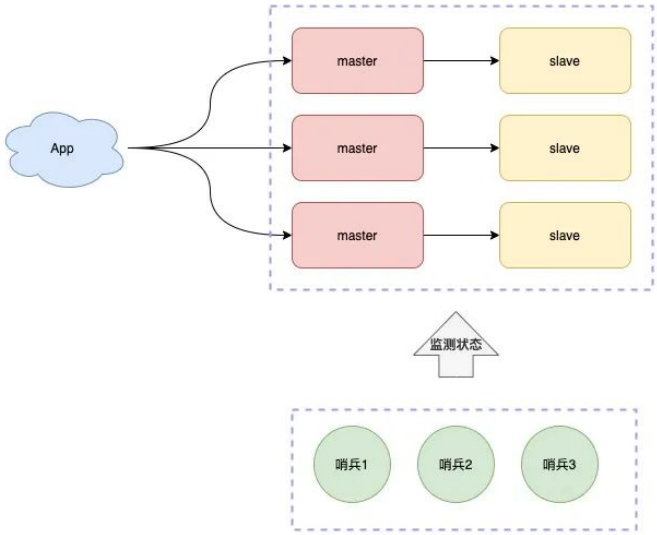
要想完美解决这个问题，此时你就需要考虑使用「分片集群」了。

05 分片集群：横向扩展

什么是「分片集群」？

简单来讲，一个实例扛不住写压力，那我们是否可以部署多个实例，然后把这些实例按照一定规则组织起来，把它们当成一个整体，对外提供服务，这样不就可以解决集中写一个实例的瓶颈问题吗？

所以，现在的架构模型就变成了这样：



现在问题又来了，这么多实例如何组织呢？

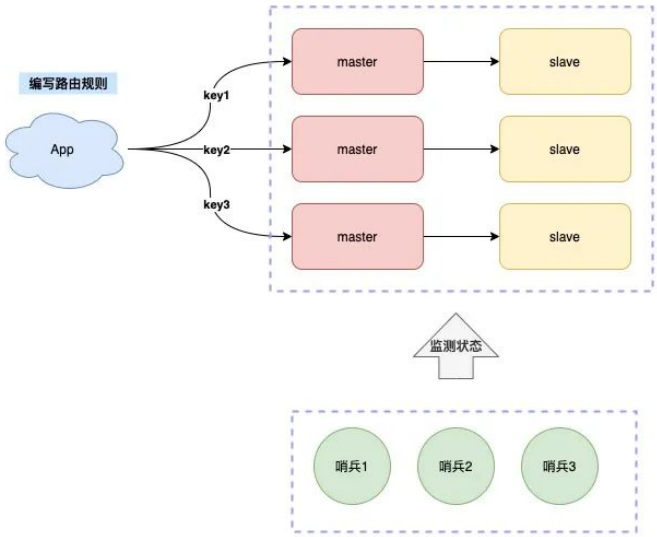
我们制定规则如下：

> 1、每个节点各自存储一部分数据，所有节点数据之和才是全量数据。
>
> 2、制定一个路由规则，对于不同的 key，把它路由到固定一个实例上进行读写。

而分片集群根据路由规则所在位置的不同，还可以分为两大类：

> 1、客户端分片
>
> 2、服务端分片

客户端分片指的是，key 的路由规则放在客户端来做，就是下面这样：

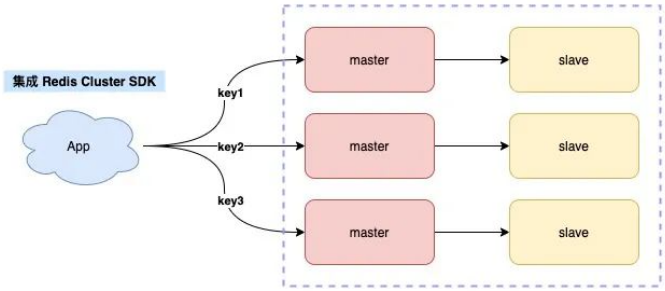


这个方案的缺点是，客户端需要维护这个路由规则，也就是说，你需要把路由规则写到你的业务代码中。

如何做到不把路由规则耦合在业务代码中呢？

你可以这样优化，把这个路由规则封装成一个模块，当需要使用时，集成这个模块就可以了。

这就是 Redis Cluster 的采用的方案。



> Redis Cluster 内置了哨兵逻辑，无需再部署哨兵。

当你使用 Redis Cluster 时，你的业务应用需要使用配套的 Redis SDK，这个 SDK 内就集成好了路由规则，不需要你自己编写了。

再来看服务端分片。

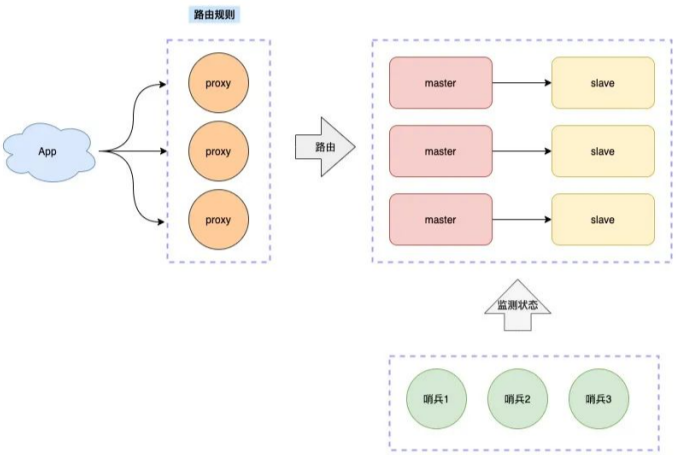
这种方案指的是，路由规则不放在客户端来做，而是在客户端和服务端之间增加一个「中间代理层」，这个代理就是我们经常听到的 Proxy。

而数据的路由规则，就放在这个 Proxy 层来维护。

这样一来，你就无需关心服务端有多少个 Redis 节点了，只需要和这个 Proxy 交互即可。

Proxy 会把你的请求根据路由规则，转发到对应的 Redis 节点上，而且，当集群实例不足以支撑更大的流量请求时，还可以横向扩容，添加新的 Redis 实例提升性能，这一切对于你的客户端来说，都是透明无感知的。

业界开源的 Redis 分片集群方案，例如 Twemproxy、Codis 就是采用的这种方案。



> 分片集群在数据扩容时，还涉及到了很多细节，这块内容不是本文重点，暂不详述。

至此，当你使用分片集群后，对于未来更大的流量压力，都可以从容面对了！

06 总结

好了，我们来总结一下，我们是如何一步步构建一个稳定、高性能的 Redis 集群的。

首先，在使用最简单的单机版 Redis 时，我们发现当 Redis 故障宕机后，数据无法恢复的问题，因此我们想到了「数据持久化」，把内存中的数据也持久化到磁盘上一份，这样 Redis 重启后就可以从磁盘上快速恢复数据。

在进行数据持久化时，我们又面临如何更高效地将数据持久化到磁盘的问题。之后我们发现 Redis 提供了 RDB 和 AOF 两种方案，分别对应了数据快照和实时的命令记录。当我们对数据完整性要求不高时，可以选择 RDB 持久化方案。如果对于数据完整性要求较高，那么可以选择 AOF 持久化方案。

但是我们又发现，AOF 文件体积会随着时间增长变得越来越大，此时我们想到的优化方案是，使用 AOF rewrite 的方式对其进行瘦身，减小文件体积，再后来，我们发现可以结合 RDB 和 AOF 各自的优势，在 AOF rewrite 时使用两者结合的「混合持久化」方式，又进一步减小了 AOF 文件体积。

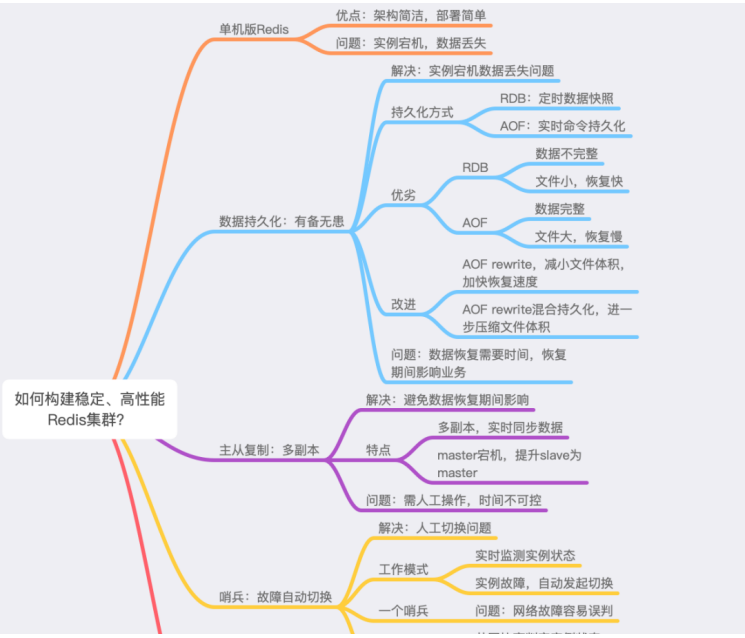
之后，我们发现尽管可以通过数据恢复的方式还原数据，但恢复数据也是需要花费时间的，这意味着业务应用还是会受到影响。我们进一步优化，采用「多副本」的方案，让多个实例保持实时同步，当一个实例故障时，可以手动把其它实例提升上来继续提供服务。

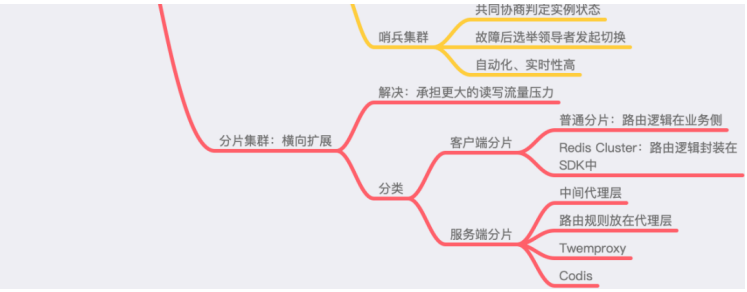
但是这样也有问题，手动提升实例上来，需要人工介入，人工介入操作也需要时间，我们开始想办法把这个流程变得自动化，所以我们又引入了「哨兵」集群，哨兵集群通过互相协商的方式，发现故障节点，并可以自动完成切换，这样就大幅降低了对业务应用的影响。

最后，我们把关注点聚焦在如何支撑更大的写流量上，所以，我们又引入了「分片集群」来解决这个问题，让多个 Redis 实例分摊写压力，未来面对更大的流量，我们还可以添加新的实例，横向扩展，进一步提升集群的性能。

至此，我们的 Redis 集群才得以长期稳定、高性能的为我们的业务提供服务。

这里我画了一个思维导图，方便你更好地去理解它们之间的关系，以及演化的过程。





07 写在最后

看到这里，我想你对如何构建一个稳定、高性能的 Redis 集群问题时，应该会有自己的见解了。

其实，这篇文章所讲的优化思路，围绕的主题就是「架构设计」的核心思想：

- 高性能：读写分离、分片集群
- 高可用：数据持久化、多副本、故障自动切换
- 易扩展：分片集群、横向扩展

当我们讲到哨兵集群、分片集群时，这还涉及到了「分布式系统」相关的知识：

- 分布式共识：哨兵领导者选举
- 负载均衡：分片集群数据分片、数据路由

当然，除了 Redis 之外，对于构建任何一个数据集群，你都可以沿用这个思路去思考、去优化，看看它们到底是如何做的。

例如当你在使用 MySQL 时，你可以思考一下 MySQL 与 Redis 有哪些不同？MySQL 为了做到高性能、高可用，又是如何做的？思路类似。

本文的思考过程，也是做「架构设计」的思路。在做软件架构设计时，你面临的场景就是发现问题、分析问题、解决问题，一步步去演化、升级你的架构，最后在性能、可靠性方面达到一个平衡。

虽然各种软件层出不穷，但架构设计的思想不会变，我希望你真正吸收的是这些思想，这样才可以做到以不变应万变。

- EOF -

推荐阅读　点击标题可跳转