03 | 如何使用简洁的架构实现高性能读服务?

在模块一我们采用“目的性”这一维度，将后台系统的架构归类为读业务、写业务及扣减类业务。在实际业务中，绝大多数情况都是读场景高于写场景，可以想象一下，你浏览多少次商品才会下一单？一天看多少条朋友圈、才会发几条朋友圈?

因此，我将先从读业务入手，带你开启本专栏的实战之旅。具体包括读业务在实现上需要满足的功能特点、遵守的基本原则及常见的实现方案存在的优缺点。

> 说明：读业务这个名词偏业务和产品，研发在做系统设计或日常沟通中，会采用偏技术实现的名词，如读接口或者读服务。在本讲的后续内容中，我统一用读服务代指读业务。

读服务的功能实现要求

读服务在实现流程上，基本上是纯粹地从存储中一次或者多次获取原始数据，进行简单的逻辑加工，或者直接返回给用户/前端业务系统。它是无状态的或者是无副作用的，也就是说每一次执行都不会在存储中记录数据或者修改数据，每一次读请求都和上一次无关。

比如，打开资讯类 App，你会看到两类场景，一类是业务后台系统直接从存储中获取今日的新闻列表，另一类是推荐系统生成一个新闻推荐列表，给到业务前台系统并展示给用户。

亦如在电商 App 里，首页展示的商品和促销等信息，是运营根据营销策略配置的，业务后台接收到读请求，然后直接去存储获取数据并进行加工后返回给业务前台系统。其他业务亦然，此处不再赘述。

结合前面两讲和上述介绍，我们可以分析出读服务在实现上需要满足的功能要求，主要有以下 3 点。

1. 保证高可用；其实不管是不是读服务，都需要满足高可用。
2. 保证高性能。我先定义一个较大的指标，TP 999 要在 100ms 以内。比如你去浏览新闻和电商 App，如果首页打开得非常慢，体验一定非常差。
3. 支持的 QPS 非常高（如上万~百万的峰值 QPS）。因为大部分业务场景都是“读多写少”。

针对这些技术功能性指标，下面将讲解如何实现。

架构尽量不要分层

在“**02 讲**”我们介绍了如何利用拆分降低系统架构复杂度，通过水平拆分将一些共性的、不易变的代码逻辑单独封装成一个模块对外服务。这样能够减少重复，提升效率。此时的读服务架构如下图 1 所示：

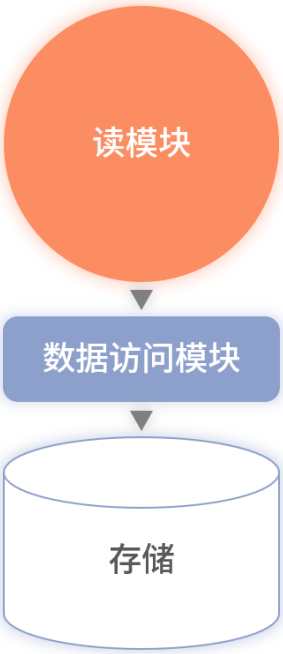

图 1：分层架构的读服务

但在实际的应用中，通过监控图可以发现此种架构下，读服务性能的平均值离 TP99 或 TP999 有较大的差距，通常在一倍以上。另外，性能的毛刺也比较多。产生这种情况有以下两个原因。

- 一方面是因为采用分层架构之后，网络传输相比不分层的架构多了一倍。
- 另一方面，读服务的业务逻辑都比较简单，性能主要消耗在网络传输上。因此，请求查询的数据越少，性能越好，假设为 10ms；数据多时，性能则较差，假设为 50ms。当叠加上分层架构，性能就会翻倍。比如数据少时，从 10ms 变成 20ms；数据多时，从 50ms 变成 100ms。分层后，数据的多与少带来的性能差距达到了 80ms，这也是产生毛刺差的原因。

因此，为了提高查询的性能减少毛刺同时降低部署机器的数量，可以将水平拆分的数据访问层代码工程保留独立，但在实际编译时，直接编译到读服务里。以 Java 举例，直接将数据访问层编译为 JAR 包并由读服务进行依赖。这样在部署时，它们在同一个进程里，去掉网络传输升级后的架构如下图 2 所示：

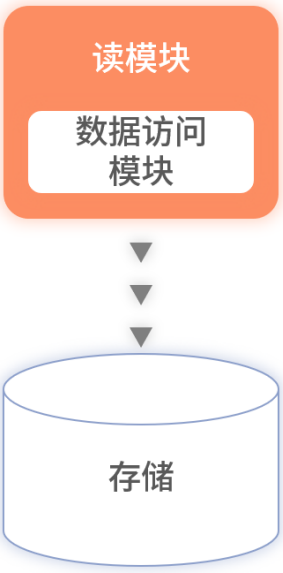

图 2：内嵌的读服务架构

在实际的案例中，当进行此项升级后。性能有了较明显变化，TP999 基本降了一半，平均性能降了 20%~30% 。下图 3 展示了一个大致效果图，你可以感受一下：

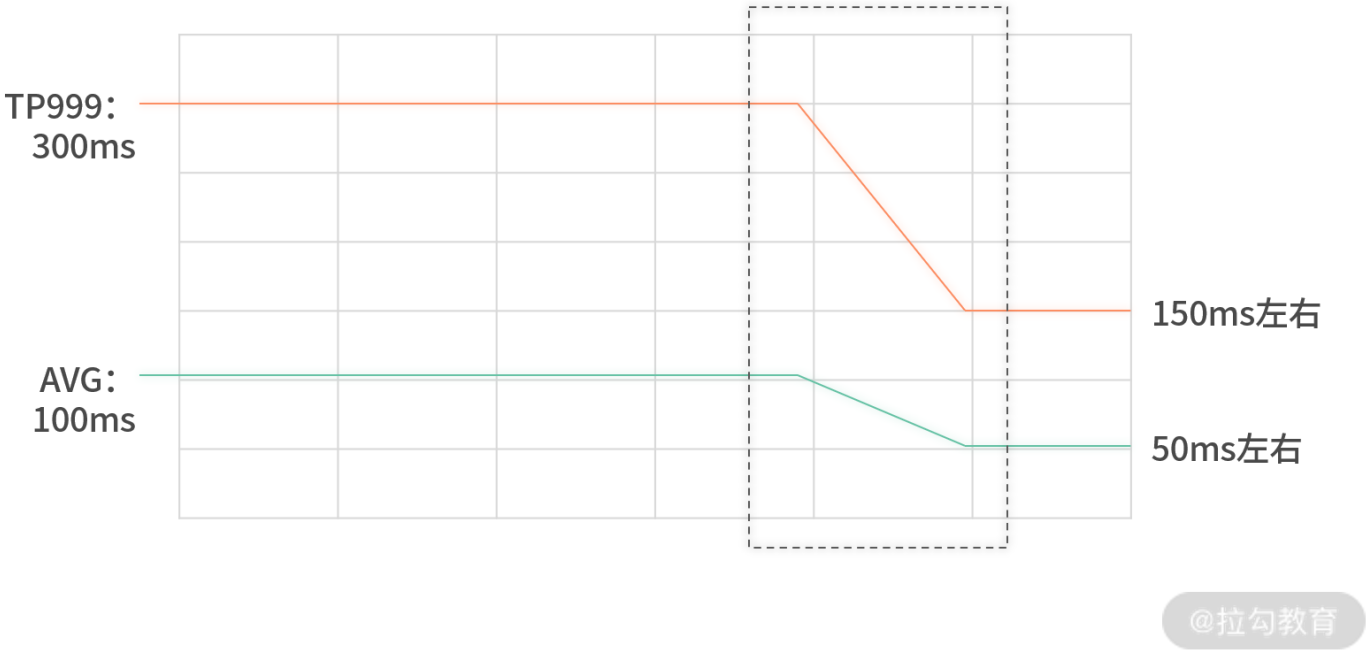

图 3：内嵌后的性能效果图

上述优化的隐含原则是，读服务要尽可能和数据靠近，减少网络传输。此项原则其实已经应用在很多类似的场景里了。比如：

1. 现在很多浏览器都自带本地缓存的功能，你浏览一个网页后，在一段时间内再次访问时，此网页的数据就是从浏览器缓存里获取的，直观感觉就是网页打开速度非常快。
2. 另外 CDN 也是一样的道理，把你需要的数据推送到和你最近的机房里，缩短网络传输的距离。

这两种场景里提升性能的方式又称为**数据前置**。但数据前置也会带来另一个问题——数据更新不及时。关于这块内容，我会在“**04 讲**”里详细介绍，并给出有效的解决方案。

代码尽可能简单

读服务的实现流程非常简单，为了方便你理解，这里我把读服务执行的大致流程画在了一张大图里，如下图 4 所示：

图 4：读服务执行流程图

结合图 4 所展示的内容，我们再来看一下读服务的执行步骤，接口在接受请求时：

1. 首先会将外部的入参解析成内部模型并进行校验；
2. 之后会根据具体的存储类型使用内部模型构建查询条件并请求对应的存储；
3. 获取到数据后，进行反序列化转换为内部模型；
4. 根据业务情况进行适当地处理；
5. 将处理后的数据转换为对外 SDK 的模型并返回。

在上述流程里，有大量的模型映射，比如将外部的模型映射为代码中的内部模型、将内部模型映射为查询条件等。这些不同层次的模型，字段都差不多。为了提升映射效率，有时可能会借助一些框架对相同的字段进行自动化的转换。

对于其他能够提升效率的地方，你可能也会引入一些框架。**在读服务对于性能要求非常严格的情况下，要尽可能地减少引入框架。如果一定要引入，必须经过严格的压测**。在实际的应用中，很多能够提升效率的框架性能都非常差。比如 Java 中的 Bean.copyProperties，它采用了反射的机制进行字段 copy，在数据量较大时，性能较低。

另外，在读服务的处理链路上，为了方便排查问题，经常会直接将请求的入参及从存储中获取的数据，使用JSON进行序列化为字符串，并进行日志打印。这种粗暴的方式，对性能也会有非常大的消耗，建议不要直接全量序列化，而是精细化地按需打印。

最后。对于读取的内容要在存储处按需取，而不是全量取回后再在服务内部进行过滤。如果存储为 MySQL，则不要使用 select *，需要手动指定你要查询的字段。如果存储为 Redis，则使用 Redis 的 hash 结构存储数据，因为 hash 结构可以让你在查询时指定需要返回哪些字段。其他存储结构，如 ElasticSearch 等亦然。

存储的选型和架构

读服务最主要依赖的中间件是存储，因此存储的性能很大程度上决定了读服务的性能。对于 MySQL、HBase等数据库，即使使用分库分表、读写分离、索引优化等手段，在并发量大时，性能也很难达到 200ms 以内。

为了提升性能，实战中的架构通常选用基于内存的、性能更好的 Redis 作为主存储，MySQL 作为兜底来构建，如下图 5 所示的架构：

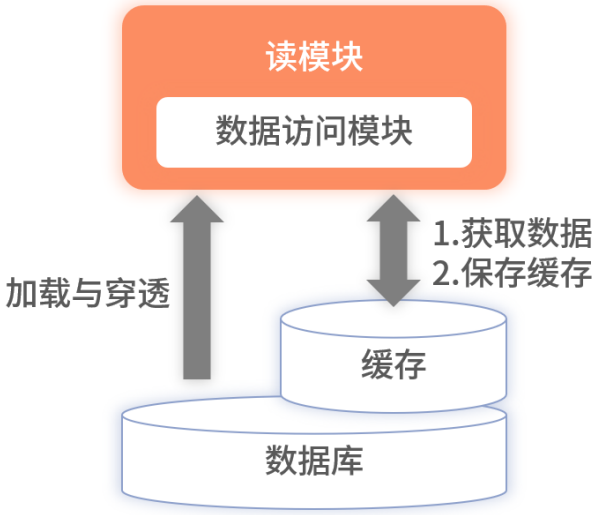

图 5：缓存+数据库的读服务架构

此架构称为懒加载模式。在初始的时候，所有数据都存储在数据库中。当读服务接受请求时，会先去缓存中查询数据，如果没有查询到数据，就会降级到数据库中查询，并将查询结果保存在 Redis 中，以供下一次请求进行查询。保存在 Redis 中的数据会设置一个过期时间，防止数据库的数据变更了，请求还一直读取缓存中的脏数据。

上述的架构设计简单清晰且实现成本较低，但还存在一些潜在的问题，不能满足本讲第一小节里提到的高可用及完全高性能的要求。主要有以下几大类问题：

1. 存在缓存穿透的风险

如果恶意请求不断使用缓存中不存在的数据发送请求，就会导致该请求每次都会被降级到数据库中。因为数据库能够支持的并发有限，如果请求量很大，可能会把数据库打挂，进而引起读服务不可用。这也就不满足高可用这个要求。

针对数据库中没有的数据，可以在缓存中设置一个占位符。在第二次请求处理时，读取缓存中的占位符即可识别数据库中没有此数据，然后直接返回给业务前台系统即可。

使用占位符虽然解决了穿透的问题，但也带来了另外一个问题。如果恶意请求不断变换请求的条件，同时这些条件对应的数据在数据库中均不存在，那么缓存中存储的表示无数据的占位符也会把整个缓存撑爆，进而导致有效数据被缓存清理策略清除或者整个读服务宕机。

对于此种恶意请求，就需要在业务上着手处理。对于请求的参数可以内置一些 token 或者一些验证数据，在读服务中前置进行校验并拦截，而不是透传到缓存或数据库中。

2. 缓存集中过期导致雪崩

对存储在缓存中的数据设置过期时间是为了定期获取数据库中的变更，但如果设置不合理，可能会导致缓存集中过期，进而所有的读请求都会因缓存未命中，而直接请求到数据库。因缓存支持的量级至少是数据库的十倍以上，此类瞬间高并发的流量会直接将数据库打挂，进而宕机。

对于数据库的过期时间，可以在设置时进行加盐操作。假设原先统一是 2 个小时过期，设置时根据随机算法在一个区间内获取一个随机值，在 2 个小时的过期时间上再加上此随机值，这就做到了各个缓存的过期时间不一致，同时过期的缓存数量最可控。

3. 懒加载无法感知实时变更

在缓存中设置过期时间，虽然可以让用户感知到数据的变更。但感知并不是实时的，会有一定延迟。在某些对于数据变更不敏感的场景是可以的，比如编辑新发布了一个新闻，但你没有看到，因为你都不知道编辑新发布了一个新闻。

如果想要做到实时看到数据的变更，可以将架构升级。升级后的架构如下图 6 所示：

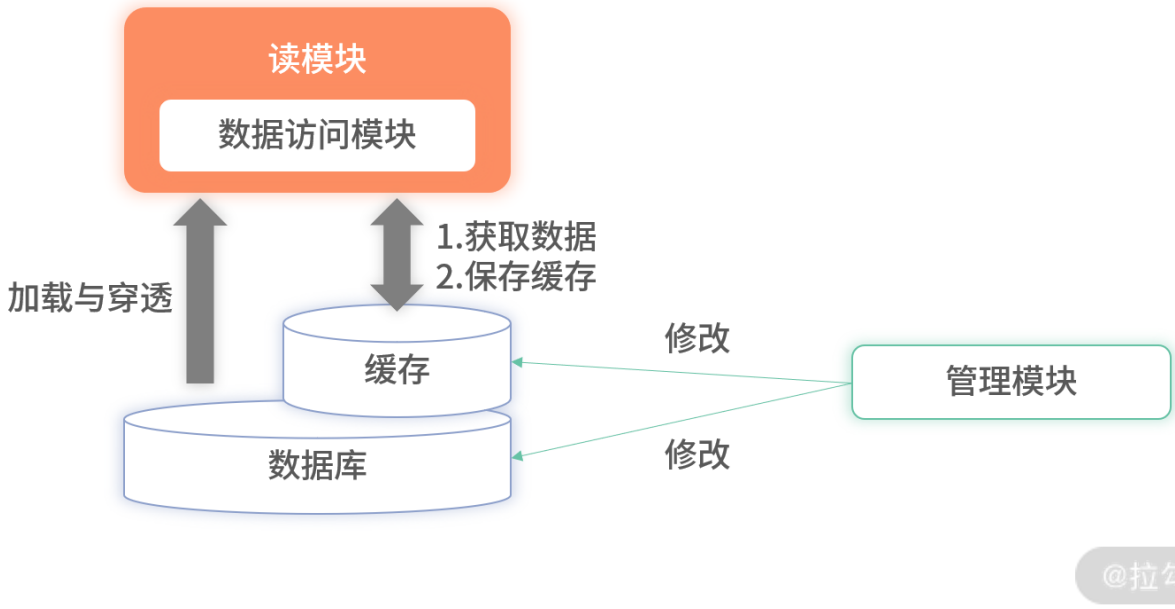

图 6：主动推送变更的架构

在每次修改完数据之后，主动将数据更新至缓存里。此种方案下，缓存里的数据均和数据库保持一致。

但在细节上，还是存在一些问题。**如果你修改完了数据库再更新缓存，在异常情况下，可能出现数据库更新成功了，但缓存更新失败了的情况**。因为数据库和缓存是两个存储，如果没有分布式事务的机制，缓存更新失败了，数据库的数据是不会回滚的。此时，缓存和数据库中的数据依然不一致，因此这个方案并没有完美解决问题。如果先更新缓存，再更新数据库，同样会因为没有分布式事务的保障，出现缓存中存在脏数据的问题。

另外，**在更新数据库后主动更新缓存的模式，在实际的实施层面很容易出现遗漏**。因为你需要在所有更新数据库的地方都加上主动更新缓存的代码，当开发人员不断变更时，很容易出现遗漏的情况，比如在某一个需求里，开发人员只更新了数据库而没有更新缓存。

除了容易遗漏之外，在所有更新数据库的地方，都利用缓存和数据库的分布式事务来保证数据完全一致的成本较高，在实际工作中，成本也是一个必须要考虑的问题。这里做一个预告，我将在 “**04 讲**”介绍一种更简单的方式来解决此遗漏问题。

4. 懒加载无法摆脱毛刺的困扰

使用懒加载的缓存过期方案，还有一个无法避免的问题，就是性能毛刺。当缓存过期时，读服务的请求都会穿透到数据库中，对于穿透请求的性能和使用缓存的性能差距非常大，时常是毫秒和秒级别的差异。

大部分普通业务场景可以容忍此问题，但在一些对性能要求极高的场景里，比如 App 首页，毛刺问题仍需重视和解决。关于此问题的解决方法，我将放在“**04 讲**”进行讲解。

至此，懒加载架构的四个问题及对应的潜在解决方案已讲解完毕。虽然懒加载架构存在一些问题，但在实际应用中，此方案及其变种方案因为实现简单、成本低，仍是使用较多的解决方案。

总结

在本讲里，我们介绍了读服务在实现时应该满足的技术功能性要求，由此确定了读服务实现时应该满足的目标——**应该遵守两个基本原则：架构尽量不要分层、代码尽可能简单**。在此原则之上，我们提供了一个在实战中常见的架构方案，指出了此方案存在的四点不足，并提供了相对应的应对方案。

在理解了上述的方案后，现在我给你留一道思考题。**你所负责过的或者你公司里的读服务架构和本讲的架构有差异吗？对于上述的几个问题，你是如何应对的**？欢迎在留言区留下你的想法，我们一起讨论。

这一讲就到这里，感谢你学习本次课程，下一讲我们将介绍 04 |如何利用全量缓存打造毫秒级的读服务？