04 | 如何利用全量缓存打造毫秒级的读服务?

上一讲我们介绍了一个简单易实现，且成本较低的高性能读服务方案及其升级方案，但其中仍有两个问题暂未完全解决：

- 第一个问题是为了保证缓存更新实时性而带来的分布式事务的问题；
- 第二个问题是懒加载导致的毛刺问题。

在本讲里，我将针对上述两个问题，和你一起利用全量缓存打造一个无毛刺、平均性能在 100ms 以内的读服务。

全量缓存的基本架构

全量缓存是指将数据库中的所有数据都存储在缓存中，同时在缓存中不设置过期时间的一种实现方式，此实现的架构如下图 1 所示：

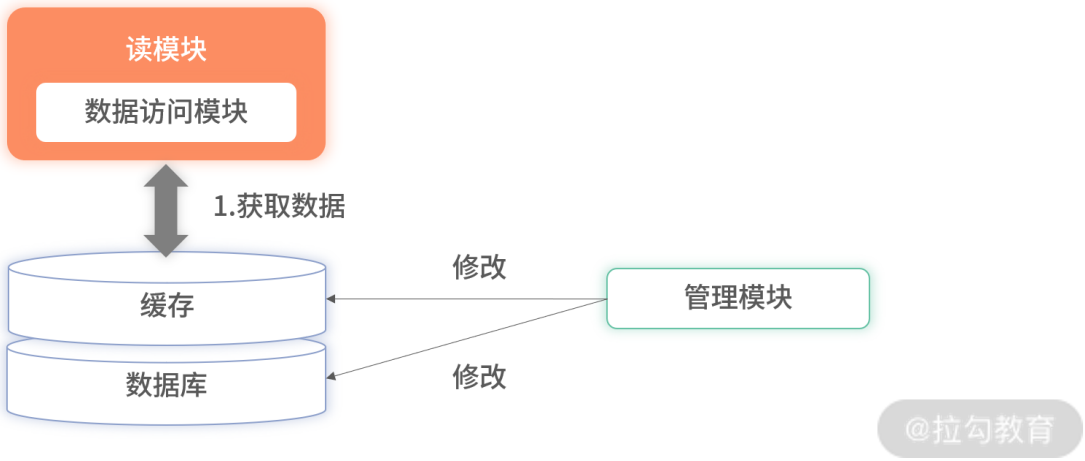

图 1：全量缓存的架构图

因为所有数据都存储在缓存里，读服务在查询时不会再降级到数据库里，所有的请求都完全依赖缓存。此时，因降级到数据库导致的毛刺问题就解决了。

但全量缓存并没有解决更新时的分布式事务问题，反而把问题放大了。因为全量缓存对数据更新要求更加严格，要求所有数据库已有数据和实时更新的数据必须完全同步至缓存，不能有遗漏。

对于此问题，一种有效的方案是采用订阅数据库的 Binlog 实现数据同步。

基于 Binlog 的全量缓存架构

在实施基于 Binlog 的架构方案前，我先简单介绍下 Binlog，更加详细的介绍我将在“**05 讲**”里和你讨论。首先看下 Binlog 的原理，如下图 2 所示：

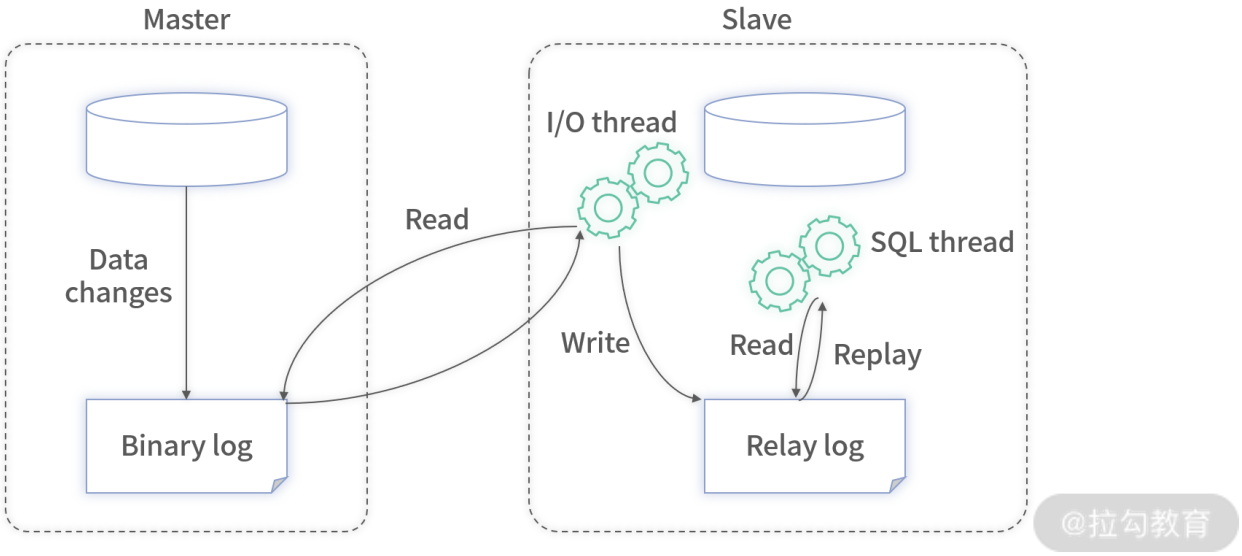

图 2：Binlog 原理图

Binlog 是 MySQL 及大部分主流数据库的主从数据同步方案。主数据库会将所有的变更按一定格式写入它本机的 Binlog 文件中。在主从同步时，从数据库会和主数据库建立连接，通过特定的协议串行地读取主数据库的 Binlog 文件，并在从库进行 Binlog 的回放，进而完成主从复制。

现在很多开源工具（如阿里的 Canal、MySQL_Streamer、Maxwell、Linkedin 的 Databus 等）可以模拟主从复制的协议。通过模拟协议读取主数据库的 Binlog 文件，从而获取主库的所有变更。对于这些变更，它们开放了各种接口供业务服务获取数据。

基于 Binlog 的全量缓存架构正是依赖此类中间件完来成数据同步的，架构如下图 3 所示：

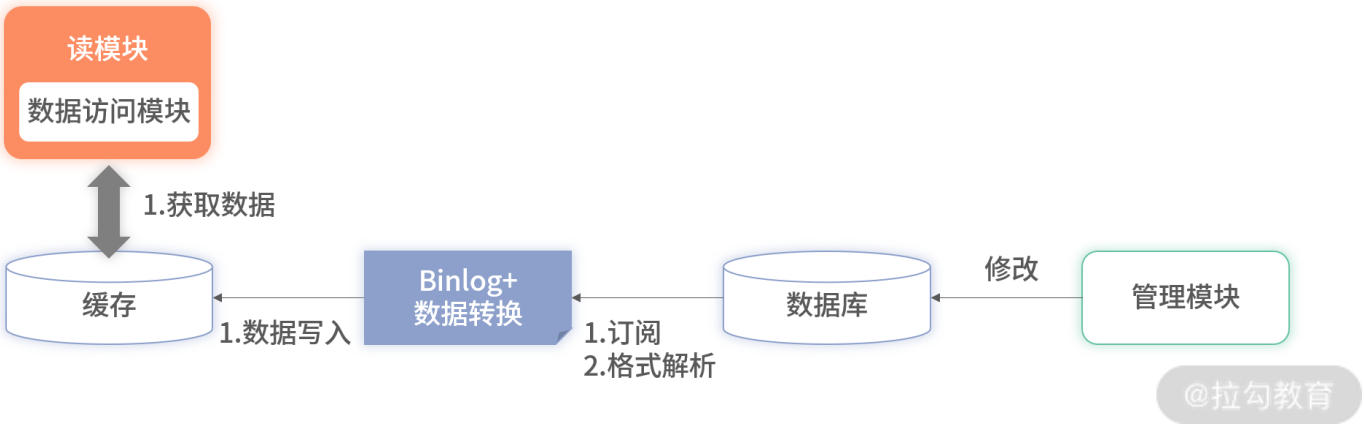

图 3：基于 Binlog 的缓存同步架构图

将 Binlog 的中间件挂载至目标数据库上，就可以实时获取该数据库的所有变更数据。对这些变更数据解析后，便可直接写入缓存里。

采用了 Binlog 的同步方案后，全量缓存的架构变得更加完整，主要表现在以下 3 个方面。

1. 降低了延迟

缓存基本上是准实时的，数据库的主从同步保持在毫秒级别，数据库的数据变更可以实时地反映到缓存里。

2. 解决了分布式事务的问题

Binlog 的主从复制是基于 ACK 机制，如果同步缓存失败了，被消费的 Binlog 不会被确认，下一次会重复消费，数据最终会写入缓存中。这就解决了因无法满足分布式事务而导致的丢数据问题，保障了数据的最终一致性。

3. 提升了代码的简洁性和可维护性

因为所有对数据库的修改最终都会反映到 Binlog 里，只要数据库的表结构不变更，对 Binlog 数据的处理程序就能保持固定。回想一下，如果采用在上一讲里提到的在代码里添加主动更新缓存的方式，那么每增加一个对数据库修改的接口，都需要加上更新缓存的代码。相比 Binlog 的方式，它的维护成本和出错概率就高得多。

基于 Binlog 的全量缓存带来了这么多提升，那它是否存在一些缺陷呢？答案是肯定的，**任何方案在带来某一方面的提升时，必然是在其他方面做出了一些取舍，架构其实是一门平衡的艺术**。

使用 Binlog 的全量缓存存在的问题

在使用的了 Binlog 的全量缓存时，会带来两个问题。

第一个问题：提升了系统的整体复杂度。当架构中只存在一个数据库中间件时，系统相对比较简单。当使用了 Binlog 后，整个数据同步的流程变长，且关注点和出错点由一个中间件变为了两个。

第二个问题：缓存的容量会成倍上升，相应的资源成本也大幅上升。 在一些对性能要求极致且实时性高的场景下，只能进行取舍，为了获取这些增强的能力，需要付出一定的代价。除了取舍之外，在技术上还有几点可以进行提升。

首先是存储在缓存中的数据需要经过筛选，有业务含义且会被查询的才进行存储。比如数据库常见的修改时间、创建时间、修改人、数据有效位等一些记录性字段可以不存储在缓存中。

其次是存储在缓存中的数据可以进行压缩。可以采用 Gzip、Snappy 等一些较常见的压缩算法进行处理，但压缩算法通常较消耗 CPU。在实际选型时，可以先压测再评估是否选择。如果无法承受压缩带来的 CPU 消耗，希望在缓存中直接存储 JSON 格式的数据或 Redis 的 Hash 结构的数据。这里我再和你分享 3 个节约缓存的小技巧。

技巧一：将数据按 JSON 格式序列化时，可以在字段上添加替代标识，表示在序列化后此字段的名称用替代标识进行表示。假设有一个 DemoClass 类，采用了替代标识后的格式，如下所示：

```
Class DemoClass{

  @Field("1")
  private field1;

  @Field("2")
  private field2;
}
```

采用了此方式序列化后的数据如下所示：

```
{"1":field1Value,"2":field2Value}
```

而没有采用此标识的数据如下所示：

```
{"field1":field1Value,"field2":field2Value}
```

从上面的示例来看，虽然只节约了 field1 和 field2 两个 key 的长度，数据量并不大。但如果你要在缓存中存储上千万、上亿条类似的数据，整体数据量还是非常可观的。另外，当前主流的 JSON 序列化工具均已支持此技巧，比如 Java 里的 Gson、FastJSON 等。你可以去对应工具的官网查看具体如何使用。

技巧二：如果你使用的缓存是 Redis 且使用了其 Hash 结构存储数据。其 Hash 结构的 Field 字段，也可以使用和上述 JSON 标识一样的模式，使用一个较短的标识进行代替。在使用全量缓存时，节约的数据也是非常可观的。

技巧三：使用全量缓存承接读服务所有的请求时，会出现无法感知缓存丢失的问题。 比如 Redis 等缓存实现虽然提供了持久化、主从备份等功能，但它为了性能，并没有提供类似数据库的 ACID 等功能，在极端情况下仍然会丢失数据。

为了保留全量缓存的优点同时解决此极端问题，可以采用异步校准加报警及自动化补齐的方式来应对。此方案的架构如下图 4 所示：

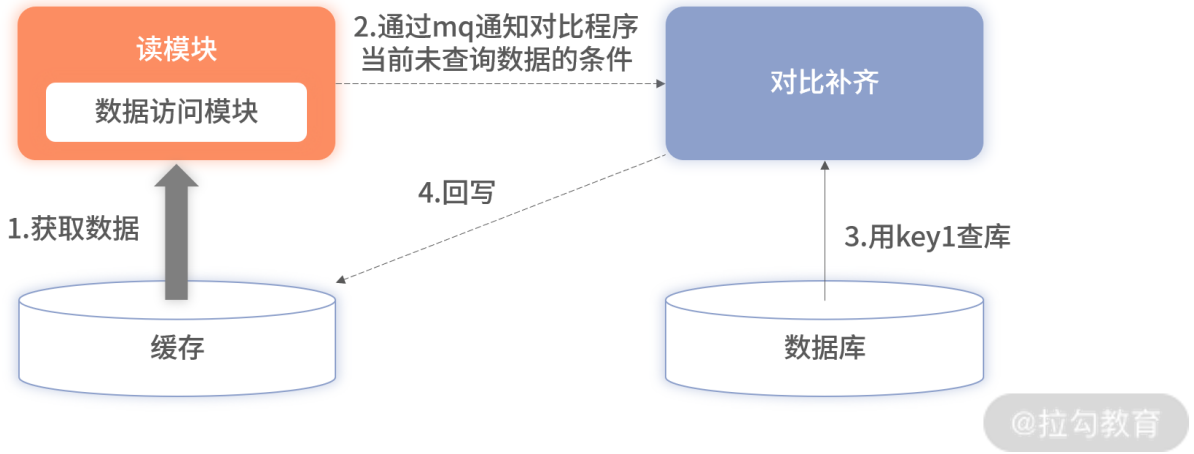

图 4：异步校准+自动化架构方案图

当读服务查询缓存中无数据后，会直接返回空数据给到调用方（见上图 4 中标记 1 处）。与此同时，它会通过 MQ 中间件发送一条消息（见上图的标记 2）。此消息的消费程序会异步查询数据库（见上图的标记 3），如果数据库确实存在数据，则会进行一次告警或者一次记录，并自动把数据刷新至缓存中去（见上图的标记 4）。

此方案是一个有损方案，如果数据在数据库中真实存在而在缓存中不存在，调用方第一次调用请求获取到的是空数据，那我们为什么还要使用此方案呢?

其实这种情况在现实场景中出现的概率极低。在我的实战经历里，在线上已经关闭了此异步校准方案，主要从以下 4 个方面来考虑。

1. 根据数据统计， 数据在数据库中存在而在缓存中不存在的概率几乎为零。
2. 对数据库大量无效的异步校准查询会导致数据库性能变差。
3. 即使缓存里数据丢失，只要此条数据存在变更，Binlog 都会把它再次刷新至缓存里。如果此条数据一直不存在变更，说明它是死数据，价值也不会太大。
4. 如果你将此方案应用到生产环境里，同时开启了异步校准，依然存在大量数据丢失的情况，那说明对于缓存中间件的使用和调优还有很大的提升空间。毕竟，此类数据丢失大多都是中间件自身导致的。我们不应该本末倒置，为了弥补缓存中间件的问题，而让业务团队做太多的补偿工作。

虽然最后我们没有采用此有损补偿方案，但这个思考和论证过程非常值得你学习和参考。当你在工作中遇到类似的问题，需要决定是否采用某个技术方案时，你可以类比上述方法，通过推理和数据验证来做最终决定。

其他优化点

在使用了 Binlog 的同步方案后，整个数据同步变得非常简单。数据同步模块接到 Binlog 的数据后，进行一定规则的数据转换后，便可直接写入缓存。

多机房实时热备

为了提升性能和可用性，可以将数据同步模块写入的缓存由一个集群变成两个集群，此时的架构演化为如下图 5 的所示：

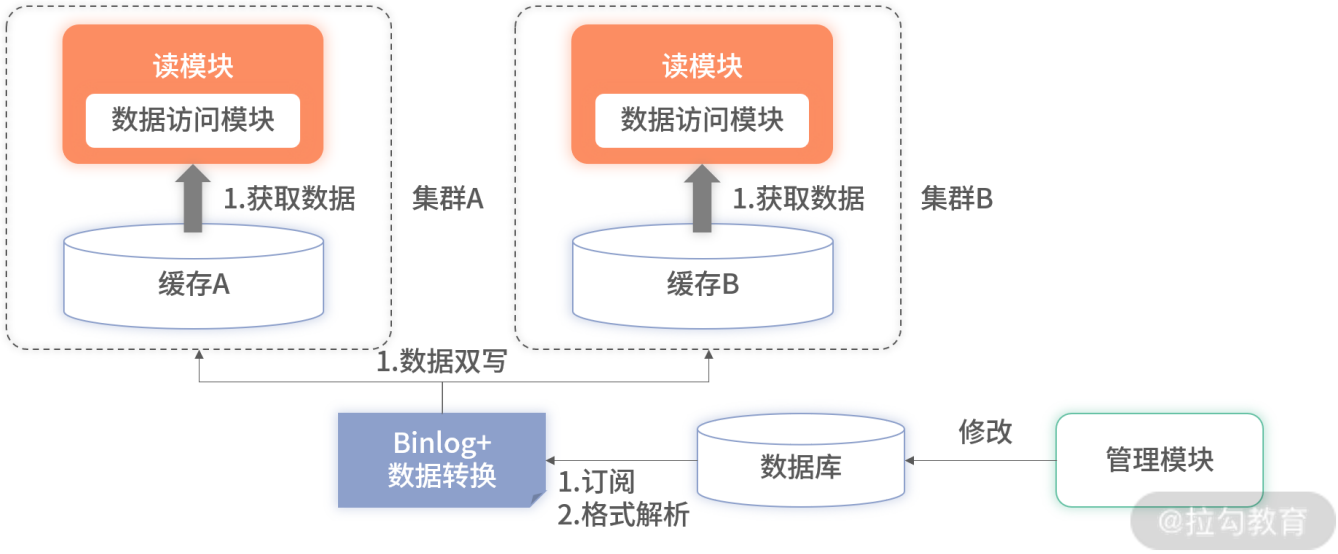

图 5：双缓存集群架构图

在部署上，如果资源允许，两套缓存集群可以分别部署到不同城市的机房或者同城市的不同分区。另外，读服务也相应地部署到不同城市或不同分区。在承接请求时，不同机房或分区的读服务只依赖同样属性的缓存集群。此方案有两个好处。

第一：提升了性能。此方式和上一讲里提到的原则——读服务不要分层，服务要尽可能地和数据靠近其实是一个逻辑。

第二：增加了可用性。当单机房出现故障时，可以无缝地将所有流量都切换至存活的机房或分区。此方案的切换时间可以达到分钟级或秒级，高可用毋庸置疑。

此方案虽然带来了性能和可用性的提升，但代价是资源成本的上升。

异步并行化

最简单的读服务场景是一次请求只和存储交互一次，但实际上很多时候交互都不止一次。对于需要多次和存储交互的场景，可以采用异步并行化的方式——接收到一次读请求后，在读服务内部，将串行与存储交互的模式改为异步并行与存储进行交互，形式如下图 6 所示：

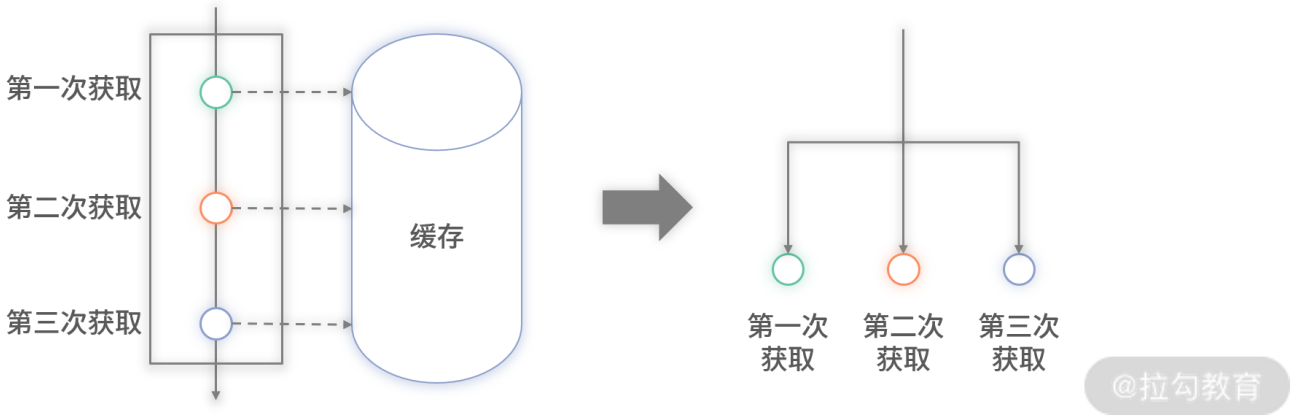

图 6：异步并行化读取数据

如果一次读请求和存储需要交互三次，假设每次交互时间为 10ms，采用串行的方式总耗时为 30ms，而采用了异步并行的方式后，三次交互为并行执行，总耗时仍为 10ms。整体的性能提升了很多。但**异步并行也带来了一些问题和局限**：

1. 首先，异步并行增加了线程的消耗，每一个异步并行都对应一个线程，进而带来 CPU 的消耗；
2. 其次，异步并行的多线程开发也带来的编程复杂度和维护难度；
3. 最后，异步并行化只能应用在每一次和存储交互都是独立的、无先后关系的场景里。

除了上述场景可以采用异步并行化外，对于一次请求查询一批数据的场景也可以进行异步并行化。当查询的一批次数据较多时，大部分性能都消耗在串行的等待网络传输上。可以将这个批次拆分成多个子批次，对每个子批次使用异步并行化的方式和存储交互，性能也会有很大的提升。具体子批次设置为多少，可以在实践中根据压测来决定。

总结

在这一讲里，**介绍了一种能够解决毛刺和更新实时性问题的全量缓存的完整架构方案。此方案提供的性能和高可用的能力，基本上可以满足各大互联网的业务场景里对于读服务的性能和容灾指标的要求**。即使是近些年越来越热闹、流量瞬间并发非常大的电商大促活动，只要对方案稍加改造，也能够轻松应对。

但上述方案也带来了另外的问题，即成本的消耗。从上述的文字中，相信你也会有这样的感受——任何新的架构和应对方案都不是完美的，架构是解决问题的方案，也是取舍的艺术。这也是本讲所传达的另一个核心思想。

最后，留一道思考题给你，此整套的全量缓存的读服务方案是否完全适合你当前负责的读类型的业务？如果不是完全适合，你会做怎样的裁剪？可以说一说你为什么这么做。