05 | 如何做到异构数据的同步一致性?

在上一讲里，我介绍了基于 Binlog 完成数据同步的全量缓存的读服务架构方案，可以实现平均性能在一百毫秒以内的高可用方案。此方案不仅可以满足缓存同步的实时性要求，还能够降低同步的复杂度，以及解决分布式事务问题。

那是不是上述方案已经十分完善，可以直接进行落地复用了呢？其实还有很多重要的点待明确。在上一讲里，我只是介绍了 Binlog 可以实现最终一致性和低延迟，但是具体如何实现及相关细节、实现中有哪些坑需要规避及最佳实践等内容均没有介绍。在本讲里，我将带你把这些内容一一攻破。

基于 Binlog 的全量缓存架构问题分析

为了方便你理解，我首先展示一张基于 Binlog 的数据同步全景图，如下图 1 所示：

注：本讲的后续内容将基于这张架构图来讨论存在的问题。

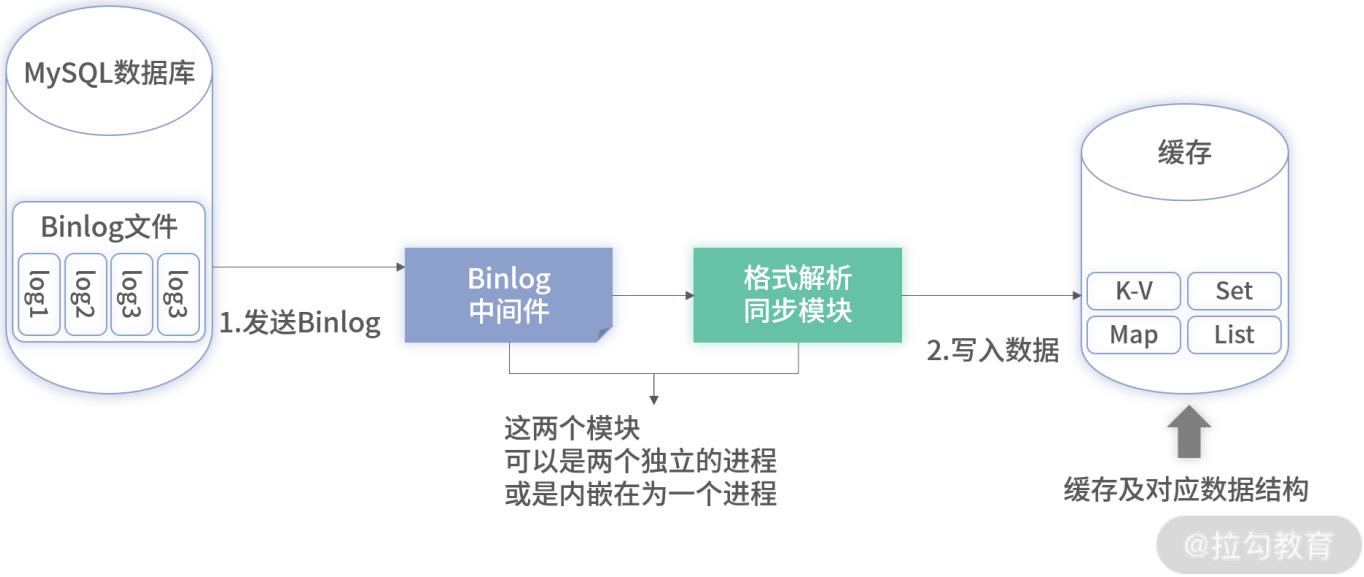

图 1：同步架构全景图

问题一：Binlog 延迟低是指纯 MySQL 的主从同步

从上图 1 中可以看出，基于 Binlog 的缓存数据同步和纯 MySQL 的主从同步在架构上是存在区别的，主要有以下 4 个区别。

1. 因为 MySQL 的主从同步是纯的数据同步，格式和协议完全适配，因此性能损耗极低。而自己使用 Binlog 同步是经过协议转换的，有一定的性能损耗。
2. 其次，上图基于 Binlog的同步比MySQL的主从同步多了两个模块，因此整体链路比也较长。
3. 在实际场景里，为了保持稳定性，同步的是从库的 Binlog，这也会导致延迟进一步加大。
4. 最后，因为 Binlog 是串行的，这会导致同步的吞吐量太低，进一步加大同步的延迟。

以上这 4 个问题，都会导致 Binlog 的实际延迟时间要比预期的要高。

问题二：如何做 Binlog 格式解析?

抽象来看，程序其实是数据和逻辑的组合。所有的程序都要按照一定的业务规则对某种数据处理才能产生价值。

Binlog 的同步程序也是一样，Binlog 同步转换程序处理的是 Binlog 的数据。那 Binlog 的格式是什么样的？是每次变更的 SQL，还是其他维度的数据？这关系到同步程序的设计方案，以及对应的实现的复杂度。

问题三：如何保证数据不丢失或错误?

MySQL 的主从同步逻辑是和业务数据无关的，正式版本发布之后，修改的频率比较低。而基于 Binlog 实现的业务数据同步程序是易变的，因为互联网业务需求迭代周期非常快，在业务高速迭代的过程中，如何保证开发人员写出没有 Bug 的代码？如何保证同步的数据不丢失、不出错呢?

问题四：如何设计缓存数据格式?

最后便是如何设计存储在缓存里的数据格式了。现在主流的数据库（如 Memcache、Redis 等）不只提供 Key-Value 的数据结构，还提供了其他丰富的数据结构类型。如何利用和设计这些数据结构，来提升数据查询和写入时的性能，同时降低代码的复杂度呢?

下面我将逐一讲解解决上述四类问题的场景手段。

如何发送Binlog

此处的方案，我将以互联网中使用最多的 MySQL 作为示例进行讲解，其他类型的数据库可以以此类推。MySQL 的 Binlog 分为三种数据格式：statement、row 及 mixed 格式，我将基于下面展示的示例表来分别介绍上述三种格式：

```
create table demo_table{
  id bigint not null auto_increment comment '主键',
  message varchar(100) not null comment '消息',
  status tinyint not null comment '状态',
  created datetime not null '创建时间',
  modified datetime not null '修改时间',
  primary key ('id') using btree
}
```

1. statement 格式

statement 格式是把每次执行的 SQL 语句记录到 Binlog 文件里，在主从复制时，基于 Binlog 里的 SQL 语句进行回放来完成主从复制。比如执行了如下 SQL 成功后：

```
update demo_table set status='无效' where id =1
```

Binlog 中记录的便是上述这条具体的 SQL。采用 SQL 格式的 Binlog 的好处是内容太少，传输速度快。但存在一个问题，在基于 Binlog 进行数据同步时，需要解析上述的 SQL 获取变更的字段，存在一定的开发成本。

2. row 格式

row 格式的 Binlog 会把当次执行的 SQL 命中的那条数据库行的变更前和变更后的内容，都记录到 Binlog 文件里。以上述 statement 格式里的 SQL 作为示例，该 SQL 在 row 格式下执行后会产生如下的数据：

```
{
  "before":{
    "id":1,
    "message":"文本",
    "status":"有效",
    "created":"xxxx-xx-xx",
    "modified":"xxxx-xx-xx"
  },
  "after":{
    "id":1,
    "message":"文本",
    "status":"无效",
    "created":"xxxx-xx-xx",
    "modified":"xxxx-xx-xx"
  },
  "change_fields":["status"]
}
```

上述案例记录的 Binlog 数据非常全面，包含了 demo_table 中所有字段对应的变更和未变更的数据，同时标记了具体哪些字段发生了变更。在数据同步时，可以完全以它为准。基于上述格式的数据同步的实现代码会非常简单，但缺点是，上述格式产生的数据量较大。

3. mixed 模式

mixed 模式是上述两种模式的动态结合。采用 mixed 模式的 Binlog 会根据每一条执行的 SQL 动态判断是记录为 row 格式还是 statement 格式。比如一些 DDL 语句，如新增加字段的 SQL，就没有必要记录为 row 模式，记录为 statement 即可，因为它本身并没有涉及数据变更。

在实际应用中，推荐使用 row 模式或者 mixed 模式，主要有以下两个原因。

原因一：这两种格式的数据量全，可以让你做更多的逻辑。因为随着业务需求的发展，同步逻辑会出现非常多的个性化需求，越多信息的数据，在编写代码时会越简单。

原因二：row 模式无须解析SQL，实现复杂度非常低。在执行的 SQL 非常复杂时，对 statement 模式里记录的 SQL 的解析需要耗费大量开发精力，越复杂的解析越容易产生 Bug，所以推荐更加简单的 row 模式的数据格式。

Binlog 如何高效消费

在上一小节里确定了 Binlog 发送的格式后，紧接着需要确定的便是 Binlog 数据该如何消费的问题。在技术上，数据消费有两种常见模式：串行和并行。下面将对这两种模式逐一讲解，并对它们存在的优缺点进行讨论。

1. 全串行的方式进行消费

以 MySQL为例，不管是表还是 SQL 维度的数据，都需要将整个实例的所有数据变更写入一个 Binlog 文件。在消费时，对此 Binlog 文件使用 ACK 机制进行串行消费，每消费一条确认一条，然后再消费一条，以此重复。具体消费形式如下图 2 所示：

图 2：基于 ACK 的串行消费图

此类模式的消费存在两个问题。

问题一：串行消费效率低，延迟大。假设一次同步 20ms 左右，同步 10W 条数据就需要 30min 左右。

问题二：单线程无法利用水平扩展，架构有缺陷。当前数据量小，可以满足。但当数据量增大后，此模式是无法通过水平扩展来提升性能。

2. 采用并行的方式提升吞吐量及扩展性

Binlog 的单文件及 ACK 机制，导致我们必须去串行消费。但实际上，通过一些技术手段我们是能够对 Binlog 文件里的不同库、不同表的数据进行并行消费的。因为不同库之间的数据是不相关的，为了在 Binlog 原有的串行机制下完成按库的并行消费，整体架构需要进行一定升级，具体如图 3 所示：

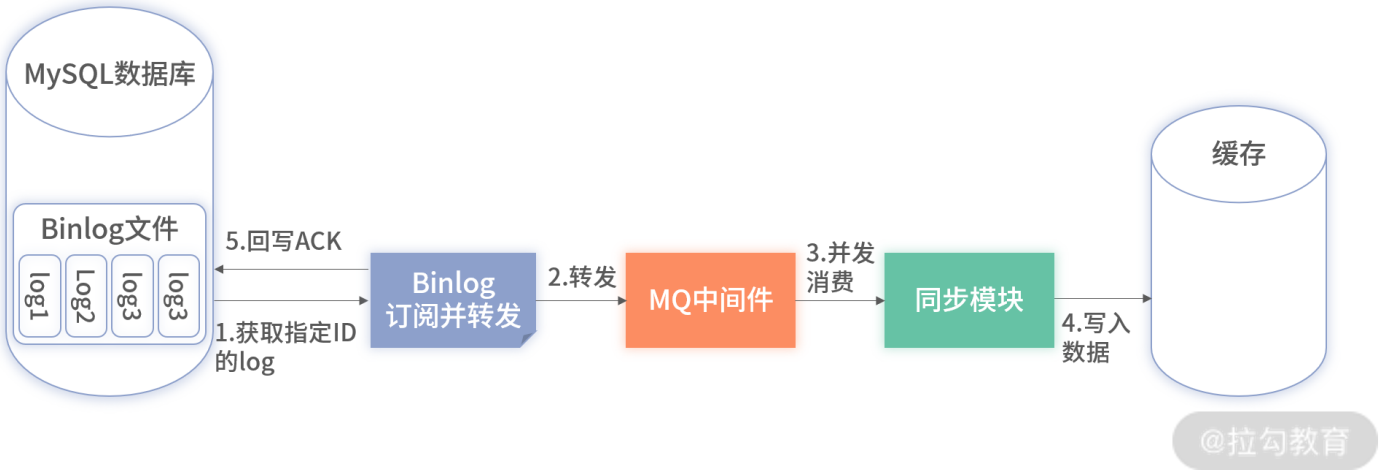

图 3：升级后的并行消费方案

上述架构里，借用了 MQ 进行拆分。在 Binlog 处仍然进行串行消费，但只是 ACK 数据。ACK 后数据直接发送到 MQ 的某一个 Topic 里即可。因为只做 ACK 并转发至 MQ，不涉及业务逻辑，所以性能消耗非常小，大概只有几毫秒或纳秒。

现在大部分的 MQ 中间件都支持数据并行消费，在开发时，上图 3 中的数据转换模块在消费数据时，开启并行乱序消费即可。此时虽然完成了从串行消费到并行消费的升级，提升了吞吐量和扩展性，但也因并行性带来了数据乱序的问题。

比如你对某一条微博连续修改了两次，第一次为 A1，第二次为 A2。如果使用了并行消费，可能因为乱序的原因，先接收到 A2 并写入缓存再接受到 A1。此时，微博中就展示了 A1 的内容，但缓存中的数据 A1 是脏数据，实际数据应该是 A2。

因此我们需要继续对升级后的方案进行改造。对于并行带来的数据错乱问题，有两个解决方案。

方案一：加分布式锁实现细粒度的串行

此方案和 Binlog 的串行区别是粒度。以上述修改微博为例，在数据同步时，只需要保证对同一条微博的多次修改串行消费即可，而多条微博动态之间在业务上没有关系，仍然可以并行消费。在实施时，加锁的维度可以根据数据是否需要串行处理而定，它可以是表中的一个字段，也可以是多个字段的组合。

确定加锁的维度后，数据库中的多张表可根据需要使用此维度进行串行消费，具体示例可参考本讲后面的“**缓存数据结构设计及写入**”小节。此方案虽然可以解决乱序问题，但引入了分布式锁，且需要业务系统自己实现，出错率及复杂度均较高。

方案二：依赖 MQ 中间件的串行通道特性进行支持

采用此方案后，整个同步的实现会更加简单。还是以上述修改微博为例，在“Binlog 订阅及转发模块”转发 Binlog 数据前，会按业务规则判断转发的 Binlog 数据是否在并发后仍需要串行消费，比如上面提到的同一条微博的多次修改就需要串行消费，而多条微博间的修改则可以并行消费，它不存在并发问题。

判断需要串行消费的数据，比如同一条微博数据，都会发送到 MQ 中间件的串行通道内。在同步模块进行同步时，**MQ 中间件里的串行通道的数据均会串行执行，而多个串行通道间则可以并发执行。借助 MQ 中间件的此特性，既解决了乱序问题又保证了吞吐量。**很多开源的 MQ 实现都具备此小节介绍的功能，如 Kafka 提供的 Partition 功能。改造后的架构如图 4 所示：

图 4：并行+串行的架构方案

最后，在采用了 MQ 进行纯串行转并行时，将 Binlog 发送到 MQ 可以根据情况进行调整，当数据量很大或者未来很大时，可以将 Binlog 的数据按表维度发送到不同的 Topic。一是能够实现扩展性；二是可以提升性能；三是通过不同表使用不同的 Topic，可以起到隔离的作用，减少表之间的相互影响。

缓存数据结构设计及写入

现在常用的缓存大多数为 Redis 或者它的变种，所以此处我们就以 Redis 支持的为准，来讨论缓存结构设计。你使用的非 Redis 缓存提供的数据结构可能有所差异，但思路是类似的。

数据库表是按技术的范式来设计的，会将数据按一对一或一对多拆分成多张表，而缓存中则是面向业务设计的，会尽可能地将业务上一次查询的数据存储为缓存中的一个 Value 值。

比如订单至少要包含订单基本信息和用户的购买商品列表。在数据库中会设计订单基本信息表和商品表。而在缓存中，会直接将订单基本信息和商品信息存储为一个 Value 值，方便直接满足用户查询订单详情的需求，减少和 Redis 的交互次数。

这种在数据库中多张表存储，而在缓存中只用 K-V 结构进行冗余存储的数据结构，需要我们在数据同步的时候进行并发控制，防止因为多张表的变更导致并发写入，从而产生数据错乱。

多张表间共享分布式锁进行协调

以上述订单为例，数据库中的订单信息表和商品表均存储了订单号，在数据同步时，可以使用订单号进行加锁。

当订单基本信息或订单中的商品同时发生变更后，因为使用了订单号进行加锁控制，在数据同步时，两张表归属同一订单号的数据实际为串行执行。因缓存中同一个订单的基本信息和商品是存储在一起的，**更新时需要把缓存中的数据读取至同步程序并替换掉此次变更的内容（如某一个发生变更的商品信息），再回写至缓存中即可。在 Redis 中，可以考虑使用 Lua 脚本完成上述过程。**

此方式虽然可以解决因 Redis 和数据库表设计不匹配带来的问题，当多张表之间加锁又降低了吞吐量。

采用反查的方式进行全量覆盖

在同步时，可以采用反查数据库的方式来补齐 Redis 需要的数据。以上述订单为例，当订单基本信息变更时，可以在同步模块通过数据库反查此订单下的所有商品信息，按 Redis 的格式组装后，直接更新缓存即可。

采用反查的方式虽然简单，但反查库会带来一定性能消耗和机器资源（如 CPU、网络等）的浪费。而且在变更量大的情况下，反查的量可能会把数据库打挂。因**此，在采用反查方案时，建议反查发送 Binlog 的从库，从而保障主库的稳定性。**

采用 Redis 的 Hash 结构进行局部更新

参考数据库的多张表设计，缓存中也可以进行多部分存储。在 Redis 中，可以采用 Hash 结构。对于一个订单下的不同表的数据，在 Redis 中存储至各个 field 下即可，同时 Redis 支持对单个 field 的局部更新。结构如下图 5 所示：

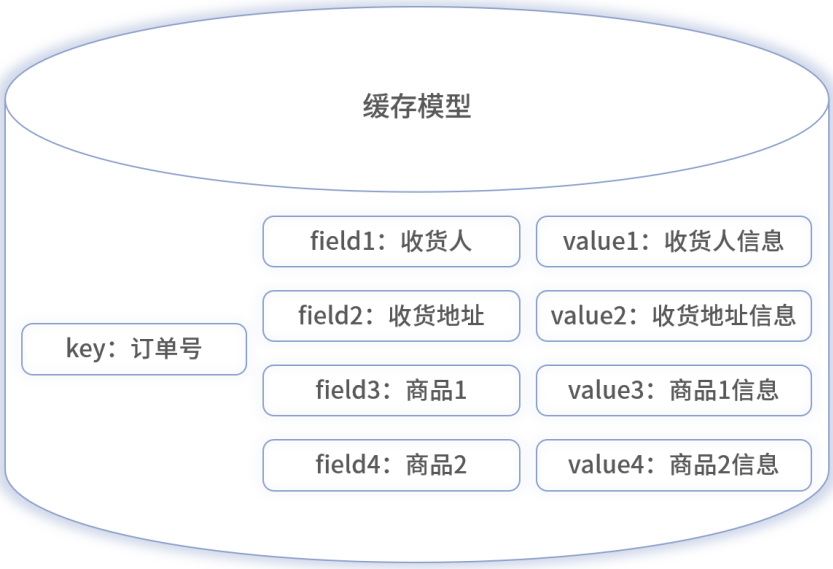

图 5：Hash 结构缓存

在上述订单案例的多张表变更时，同步程序无须对多张表间进行分布式加锁协调，哪张表变更就去更新缓存中对应的局部信息即可。不管是同步性能还是实现难度均较好。

在查询时，直接使用订单号即可查询到所有信息。为什么使用 Hash 结构，而不使用所有缓存都支持的 Key+Value 的结构呢？其中，Key 设计为订单号+子表标识，如 Key 为 OrderId_BaseInfo，表示某一个订单的基本信息，或者 Key 为 OrderId_SkuId，表示某一个订单下的某个商品基本信息。

主要有以下 3 个原因。

1. 首先，使用了 KV 结构后，查询时需要使用多个命令。如果提供了批量命令，也可以使用批量命令解决此问题。
2. 其次，一个订单下的商品是动态的，无法提前固定。如果全部改为 KV 结构，就无法查询到订单详情了。除非再异构一份订单下所有商品的 ID 列表。
3. 最后，现在主流缓存都是分布式部署的。如果采用 KV 的分割设计，很有可能一个订单的基本信息和商品信息被存储在两个分片上，此处查询的性能和复杂度也会上升。因 Redis 是使用 Key 进行分布式路由的，采用 Hash 结构的数据都存储在同一个分片上，不会出现跨分片查询的问题。

数据对比发现错误

数据同步模块是基于业务进行数据转换的，在开发过程中，需要基于业务规则不断地迭代。此外，为了保证吞吐量和性能，整个基于 Binlog 的同步方案在本讲了做了很多升级和改造。在这个不断迭代的过程中，难免会出现一些 Bug，导致缓存和数据库不一致的情况。为了保障数据的一致性，可以采用数据对比进行应对，架构如下图 6 所示：

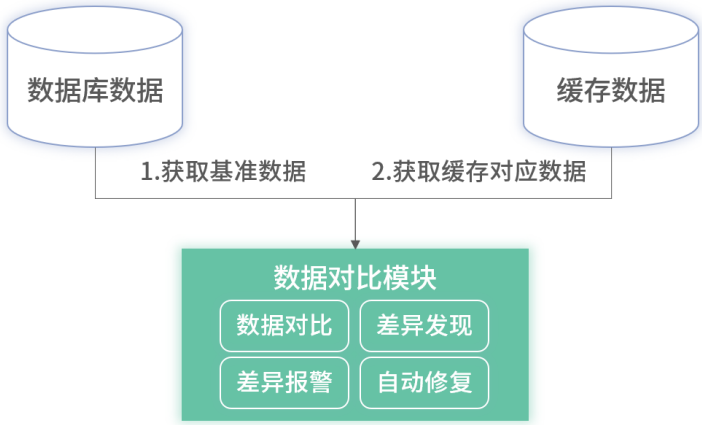

图6：数据对比架构图

数据对比以数据库中的数据为基准，定期轮询对比缓存和数据库的数据。如果发现不一致后，可以增加延迟重试，再次对比。如果多次对比不一致后，可以增加报警并保留当时的数据，之后以数据库中的数据为准刷新缓存。延迟重试是为了防止因同步的时差，出现短暂的数据不一致但最终数据一致的情况。其次，保留出错现场的数据是为了排查定位问题。

最后的兜底，直接写入

虽然上述在提升同步吞吐量上做了非常多地设计，但不可否认延迟总是存在的，即使是纯数据库主从同步间也会因为网络抖动和写入量大的情况出现毫秒或者秒级延迟，本讲基于 Binlog 的改良方案自然不例外。

绝大部分的业务和场景，对于毫秒或秒级延迟无感知。但为了方案的完整性和极端场景的应对，可以在异步同步的基础上，增加主动同步。方案如下图 7 所示：

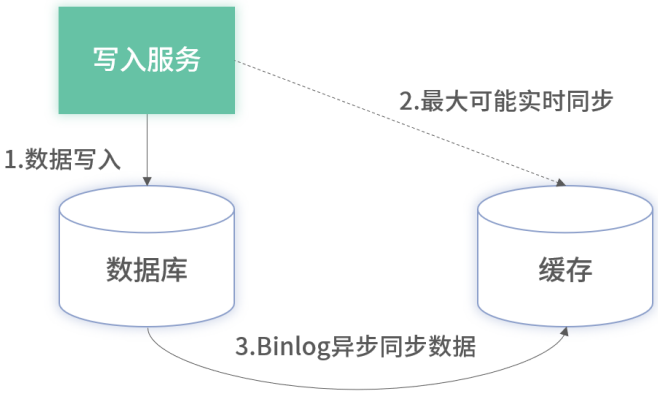

图 7：主动写入的同步方案

上述的架构是对一些关键场景在写完数据库后，主动将数据写入缓存中去。但对于写入缓存可能出现的失败可以不处理，因为主动写入是为了解决缓存延迟的问题，主动写入导致的丢失数据由 Binlog 保障最终一致性。**此架构是一个技术互补的策略，Binlog 保证最终一致性但可能存在延迟，主动写入保障无延迟但存在丢数据。**在架构中，你也可以采用此思路。**一个单项技术无法完美解决问题时，可以对短板寻找增量方案，而不是整个方案完全替换。**

总结

在本讲里详细介绍了采用 Binlog 同步数据存在的延迟、数据丢失、格式解析和缓存数据结构设计这四大类问题。并采用结构化的方式进行了解答，从 Binlog 如何发送、如何消费、数据如何写入缓存，以及如何进行数据对比发现问题这四大步骤进行了解答。相信你看完本讲后，**可以直接在你所负责的模块里，落地基于 Binlog 的数据同步，并根据环境要求，选择适合的最佳组合方案**。

最后我再给你留一个讨论题，你使用过基于 Binlog 或者其他方式的同步方案吗？使用中存在什么问题，你又是如何解决的？欢迎写在留言区，我们一起进行讨论。