迁移方案、开发方案复盘 & 总结

存储层差异

- mysql 字段值的大小写

表的 collate 配置影响 mysql 对待字段值的方式，默认的 collate 与 the version of mysql server 有关，认为都是不区分大小写的，使用下面这些值来避免这个问题。

- utf8_general_cs (case sensitive)
- utf8_bin (binary) -- sort by byte sequence
- mysql 串字段右侧空白字符会被忽略

CHAR, VARCHAR, and TEX` values 的右侧空格会被忽略，包括查询及插入时对 duplicate key 的检查

All MySQL collations are of type `PAD SPACE`. This means that all `CHAR`, `VARCHAR`, and `TEXT` values are compared without regard to any trailing spaces. “Comparison” in this context does not include the `LIKE` pattern-matching operator, for which trailing spaces are significant.

https://dev.mysql.com/doc/refman/5.7/en/char.html

- 调研真实数据的 schema
- 确认串字段的最长长度
- 确认每个字段的数据类型是否唯一
- 确认边界数据，考虑清洗或丢弃
- 按主键排序

mongo._id 映射到 mysql 的一个新的串字段后，在查询时如果不指定查询顺序，其顺序会依赖 mysql 记录的插入顺序，对存量数据则是依赖于迁移平台/脚本的扫描迁移逻辑，还可能与增量数据交叉在一起，需要考虑接口上层是否在意顺序，必要时应该考虑 order by oid。

- 索引

mongodb 也有类似 mysql 的前缀查询优化，但除此之外还有更多较为复杂的规则和内容。

参阅：

The ESR (Equality, Sort, Range) Rule — MongoDB Manual

Index Intersection — MongoDB Manual

Compound Indexes — MongoDB Manual

这可能导致在同样顺序的联合索引下，同一个查询在 mongo 中可能比 mysql 要快得多。

所以，在映射到 mysql schema 时需要额外参考上面这些内容，充分考虑 mongo 与 mysql 在索引上的差异。

为了在过去所有查询 case 上能够达到与 mongo 同等或更高的性能，可能涉及表的关系拆分，或对索引的精心挑选。

- json

在没有多值索引特性 (mysql@8) 的情况下，json_contains 的性能非常差，百万数量级的全表 json 扫描接近 10s。

在最初的 mongodb collection -> mysql table 的结构映射工作中，应该充分考虑这个问题，并做一些测试来验证性能。

db 层的设计、开发和迭代

最小接口（接口隔离）：

- 接口应该尽量正交

单一职责：

- 在设计时，接口应该（几乎）完全不与具体的数据库相关。

db 层的重要设计目标之一是屏蔽数据在存储层的差异。

- 在设计以及实现时，单一接口内不允许同时包含读写两种数据库操作。
- 接口的实现应该类似 fp 的纯函数。

写接口的调用不应引起任何除向数据库网络 IO 外的其他副作用，例如 uuid 生成，password 带 salt hash 等。

写接口对同样的入参执行无限次应该永远发出相同的数据库写请求。

- 尽量少或不对入参做修改。

仅包含必要的到存储层数据类型的转换，这一转换也应是抽象的，对上层应该做到不被感知。

- 不应与业务相关，不包含任何业务逻辑。
- 入参的合法性，由调用方保证。
- 异常封装为通用的业务异常。
- 返回的复杂类型使用通用的 DTO。

工程：

- 保持单一写入口，对外不暴露数据库接口，保证数据写入完全可控。
- 更新数据时尽量不要使用 upsert，存在并发同步问题。

迁移方案

指导原则：

分离读写流量，分阶段在运行时切换控制流状态，从而在保持对上层服务无感的前提下平滑迁移，并维持双库数据的一致性。

需求能力：

- 存量数据从源库到目标库的迁移
- 运行时切换读写流量的控制流

方案细节：

我们把对数据库的访问抽象为四种操作：

mysql 读、mysql 写，mongodb 读、mongodb 写

根据这四种操作，可以用一个读写状态来描述系统在某一时刻的状态：

例如：(mysql 读, mongodb & mysql 写)。

我们还称在某一时刻之前，数据库中的所有数据为 "存量数据"，之后新增的对数据库的写操作导致的数据变化称 "增量数据"。

存量数据和增量数据分别通过通过源库到目标库的迁移能力和运行时读写流量切换来同步，并共同保持双库的数据一致性。

于是我们用当下时刻的系统状态来描述一个流量切换的时序安排。

1. mongo 读，mongo 写
2. mongo 读，mongo & mysql 写
3. mongo 读，mongo & mysql 写，开始同步存量数据
4. mongo 读，mongo & mysql 写，存量数据同步完成
5. mysql 读，mongo & mysql 写
6. mysql 读，mysql 写

说明：

- 节点 1 是服务的原始状态。
- 从节点 1 - 5 的任何一个操作都是可逆的，原 mongo 读写的功能与最初完全一致且允许随时回滚。
- 节点 2 - 5 可以用来观察写逻辑是否如预期。
- 节点 5 主要用来观察读逻辑是否如预期。
- 操作 5 - 6 标志着迁移结束，不可逆。

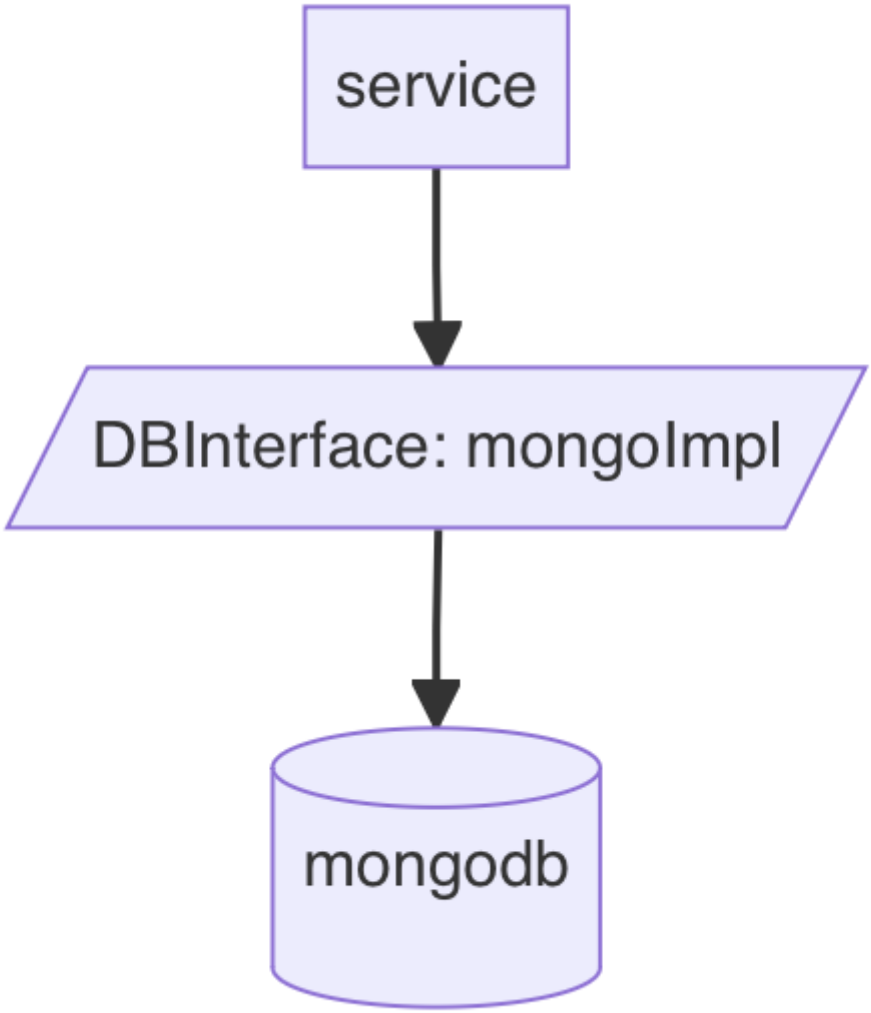
开发方案

这里将归纳提炼一些内容，包括：

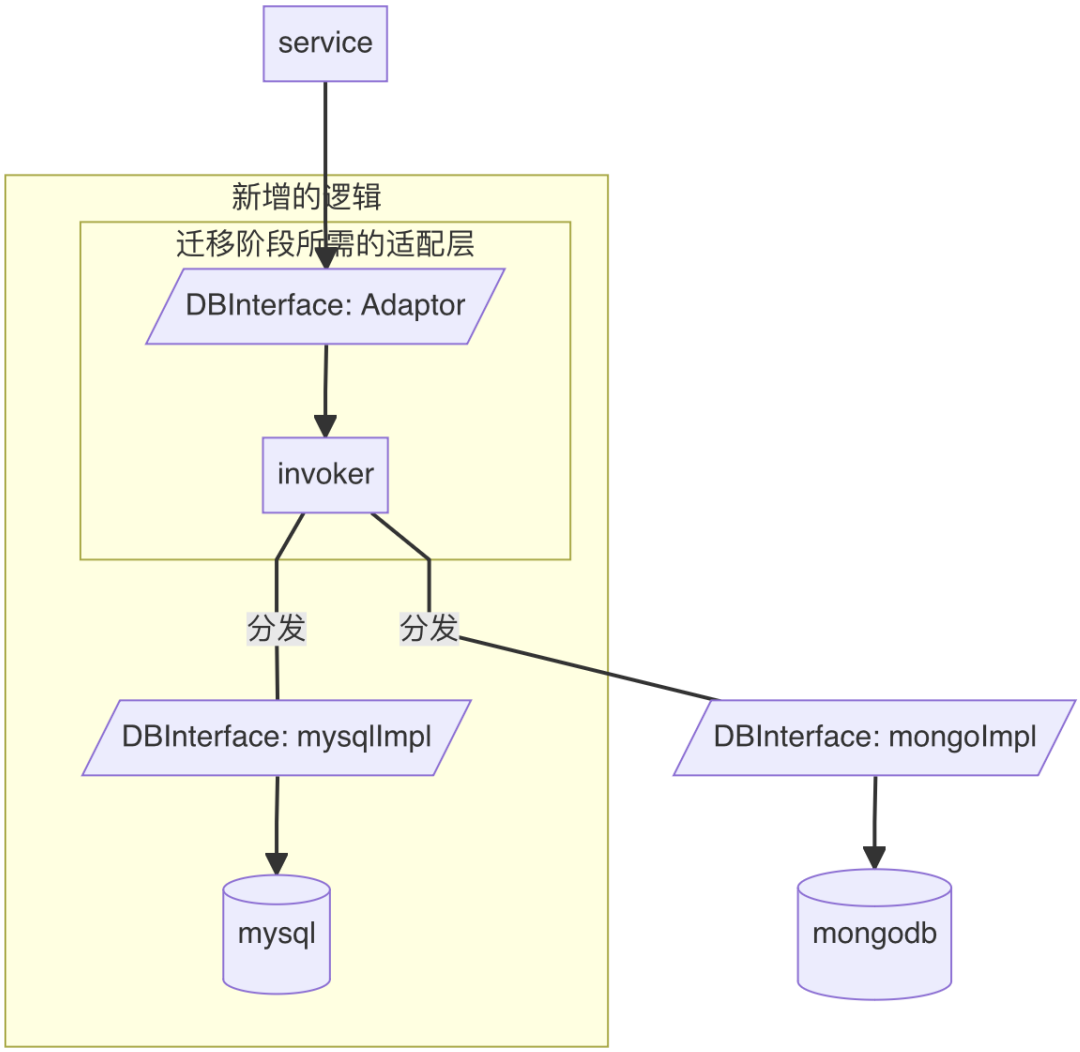
- 面对不同质量的 db 层代码的抽象方案 √
- 可行的另一条开发路线 √
- 在缺少测试支持的情况下对数据、逻辑的验证方案 √
- 在迁移过程中遇到的问题

- 在开发过程中遇到的问题

主要方案



首先是开发的主线，我们要实现一套 mysql 的 db 层 `mysqlImpl`，最终会使用这一层完全代替原有的使用 mongo 的 db 层实现。



为了能在运行时控制读写调用的控制流，我们引入了一个新的 db 层实现，它在迁移过程中将直接接收来自上层的调用，并负责根据开关状态将调用分发到具体的 mongo/mysql 实现中。

这个分发可以简单地在 `DBInterface: Adaptor` 层的每个方法中根据系统状态手动调用到对应的方法。

从调用 adaptor 到从该层返回，其分发行为应当仅受同一状态影响，否则控制流会在中途被打乱，跑到奇怪的地方。

这很好实现，在开关配置的实现中，确保：

- 调用方只读一次就能拿到完整状态
- adaptor 在分发调用时对配置仅读一次

例如 这里，这是一个通过反射实现的通用的调用分发逻辑，

另一条路线

如果先前的 db 层设计非常糟糕，出现了不少上文提到过的问题，例如读写伴随、复杂业务逻辑、有副作用等等，最坏情况下需要细粒度地在 mongoImpl/mysqlImpl 的实现中逐行拆分和控制读写，此时我们有两个选择：

- 将 mongoImpl/mysqlImpl 实现内容抽到 adaptor 中细粒度地控制。
- 重构 mongoImpl 的逻辑，回归验证后再实现 mysql 的逻辑。

大部分情况下，对 mongoImpl 进行重构的成本会比抽取逻辑到 adaptor 中小得多，但要额外承担 mongo 重构本身带来的风险 —— 例如排查迁移问题时需要额外考虑可能存在的 mongo 重构的缺陷。

如果完全采用抽取逻辑到 adaptor 的方案，则要承担在撤掉 adaptor 时的复杂度复杂度上升带来的风险。

这需要根据实际项目 db 层的代码质量来做取舍。

数据验证

第一件事是确保数据抽象一致，例如，在经过谨慎评估和调研的情况下，我们可以认为 mongodb 中的 null or not exists 与 mysql 的 "" 是一致的。

数据验证可以由内部平台支持，也可以在缺少基础建设的情况下自行实现。

这一步是必要的，在数据验证的过程中会发现很多预料之外的事情，其中大部分是 **存储层差异** 导致的问题。

逻辑监控

在运行时合适的时机进行 db 层接口级别的 返回值比较，当发生不一致时在 log 中表现出来（对相关对象的比较的逻辑大部分情况下是需要自定义的）。

- 方便在迁移过程中排查问题时获取更多的上下文信息。
- 保持 mysqlImpl 与 mongoImpl 的行为一致。

这个 "行为一致" 是抽象的，只要做好对调用方的调研，确认两侧接口存在的差异对上层控制流无感即可。

需要考虑的点：

- created_at 这类时间字段，大部分情况下对业务无感，不需要比较，但需要调研。

这个合适的时机是：

- 节点 2 - 5（双写时），将写调用分发到两边的写接口中
- 节点 4 - 5（存量数据同步后），将读调用分发到两边的读接口中

这里要注意阻塞 IO 的时间效率问题，与双写必须保持写一致不同，额外的读调用有必要考虑并发请求。

并发请求时，为了联系上下文，可能需要将 log 上下文的 trace 标识与新的异步上下文进行关联，这里的具体方案取决于具体的技术栈。

参考 这里的实现