从好友中心开始，聊“多对多”类业务数据库水平切分架构实践

本文将以“好友中心”为例，介绍“多对多”类业务，随着数据量的逐步增大，数据库性能显著降低，数据库水平切分相关的架构实践。

一、什么是多对多关系

所谓的**“多对多”**，来自数据库设计中的“实体-关系”ER模型，用来描述实体之间的关联关系，一个学生可以选修多个课程，一个课程可以被多个学生选修，这里学生与课程时间的关系，就是多对多关系。

二、好友中心业务分析

好友关系主要分为两类，**弱好友关系**与**强好友关系**，两类都有典型的互联网产品应用。

弱好友关系的建立，不需要双方彼此同意：

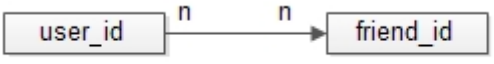
- 用户A关注用户B，不需要用户B同意，此时用户A与用户B为弱好友关系，对A而言，暂且理解为“关注”；
- 用户B关注用户A，也不需要用户A同意，此时用户A与用户B也为弱好友关系，对A而言，暂且理解为“粉丝”；

微博粉丝是一个典型的弱好友关系应用。

强好友关系的建立，需要好友关系双方彼此同意：

- 用户A请求添加用户B为好友，用户B同意，此时用户A与用户B则互为强好友关系，即A是B的好友，B也是A的好友。

QQ好友是一个典型的强好友关系应用。



好友中心是一个典型的多对多业务，一个用户可以添加多个好友，也可以被多个好友添加，其典型架构为：

- friend-service：好友中心服务，对调用者提供友好的RPC接口
- db：对好友数据进行存储

三、弱好友关系-元数据简版实现

通过弱好友关系业务分析，很容易了解到，其核心元数据为：

guanzhu(uid, guanzhu_uid);

fensi(uid, fensi_uid);

其中：

- guanzhu表，用户记录uid所有关注用户guanzhu_uid
- fensi表，用来记录uid所有粉丝用户fensi_uid

需要强调的是，**一条弱关系的产生，会产生两条记录，一条关注记录，一条粉丝记录。**

例如：用户A(uid=1)关注了用户B(uid=2)，A多关注了一个用户，B多了一个粉丝，于是：

- guanzhu表要插入{1, 2}这一条记录，1关注了2
- fensi表要插入{2, 1}这一条记录，2粉了1

> **如何查询一个用户关注了谁呢？**
>
> **回答**：在guanzhu的uid上建立索引：
>
> select * from guanzhu where uid=1;
>
> 即可得到结果，1关注了2。
>
> **如何查询一个用户粉了谁呢？**
>
> **回答**：在fensi的uid上建立索引：
>
> select * from fensi where uid=2;
>
> 即可得到结果，2粉了1。

四、强好友关系-元数据实现一

通过强好友关系业务分析，很容易了解到，其核心元数据为：

friend(uid1, uid2);

其中：

- uid1，强好友关系中一方的uid
- uid2，强好友关系中另一方的uid

uid=1的用户添加了uid=2的用户，双方都同意加彼此为好友，这个强好友关系，**在数据库中应该插入记录{1, 2}还是记录{2,1 }呢**？

回答：都可以，为了避免歧义，可以人为约定，插入记录时uid1的值必须小于uid2。

例如：有uid=1,2,3三个用户，他们互为强好友关系，那边数据库中可能是这样的三条记录

{1, 2}

{2, 3}

{1,3 }

> **如何查询一个用户的好友呢？**
>
> **回答**：假设要查询uid=2的所有好友，只需在uid1和uid2上建立索引，然后：
>
> select * from friend where uid1=2
>
> union
>
> select * from friend where uid2=2
>
> 即可得到结果。

作业：为何不使用这样的SQL语句呢？

select * from friend uid1=2 or uid2=2

供大家思考。

五、强好友关系-元数据实现二

强好友关系是弱好友关系的一个特例，A和B必须互为关注关系（也可以说，同时互为粉丝关系），即也可以使用关注表和粉丝表来实现：

guanzhu(uid, guanzhu_uid);

fensi(uid, fensi_uid);

例如：用户A(uid=1)和用户B(uid=2)为强好友关系，即相互关注：
用户A(uid=1)关注了用户B(uid=2)，A多关注了一个用户，B多了一个粉丝，于是：

- guanzhu表要插入{1, 2}这一条记录
- fensi表要插入{2, 1}这一条记录

同时，用户B(uid=2)也关注了用户A(uid=1)，B多关注了一个用户，A多了一个粉丝，于是：

- guanzhu表要插入{2, 1}这一条记录
- fensi表要插入{1, 2}这一条记录

六、数据冗余是实现多对多关系水平切分的常用实践

对于强好友关系的两类实现：

- friend(uid1, uid2)表
- 数据冗余guanzhu表与fensi表（后文称正表T1与反表T2）

在数据量小时，看似无差异，但数据量大时，数据冗余的优势就体现出来了：

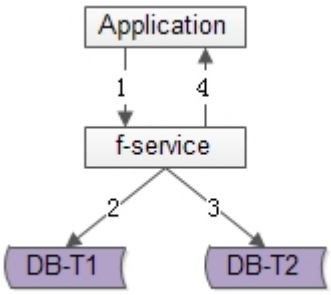
- friend表，数据量大时，如果使用uid1来分库，那么uid2上的查询就需要遍历多库
- 正表T1与反表T2通过数据冗余来实现好友关系，{1,2}{2,1}分别存在于两表中，故两个表都使用uid来分库，均只需要进行一次查询，就能找到对应的关注与粉丝，而不需要多个库扫描

数据冗余，是多对多关系，在数据量大时，数据水平切分的常用实践。

七、如何进行数据冗余

接下来的问题转化为，好友中心服务如何来进行数据冗余，常见有三种方法。

方法一：服务同步冗余



顾名思义，由好友中心服务同步写冗余数据，如上图1-4流程：

1. 业务方调用服务，新增数据
2. 服务先插入T1数据
3. 服务再插入T2数据
4. 服务返回业务方新增数据成功

优点：

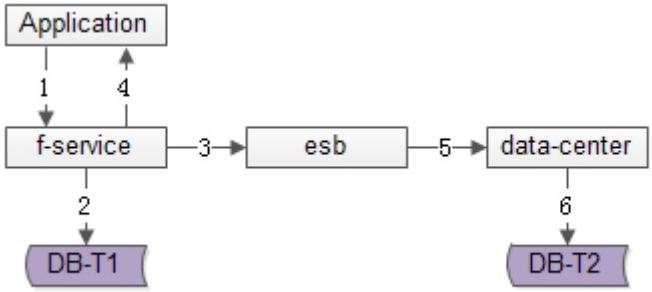
1. 不复杂，服务层由单次写，变两次写
2. 数据一致性相对较高（因为双写成功才返回）

缺点：

1. 请求的处理时间增加（要插入次，时间加倍）
2. 数据仍可能不一致，例如第二步写入T1完成后服务重启，则数据不会写入T2

如果系统对处理时间比较敏感，引出常用的第二种方案

方法二：服务异步冗余



数据的双写并不再由好友中心服务来完成，服务层异步发出一个消息，通过消息总线发送给一个专门的数据复制服务来写入冗余数据，如上图1-6流程：

1. 业务方调用服务，新增数据
2. 服务先插入T1数据
3. 服务向消息总线发送一个异步消息（发出即可，不用等返回，通常很快就能完成）
4. 服务返回业务方新增数据成功
5. 消息总线将消息投递给数据同步中心
6. 数据同步中心插入T2数据

优点：

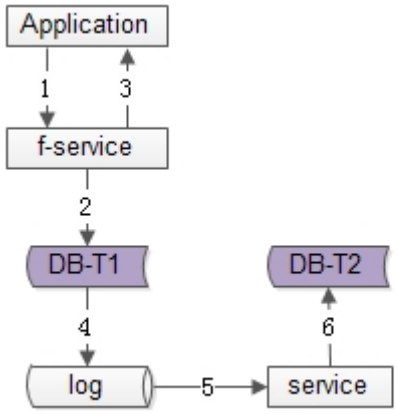
请求处理时间短（只插入1次）

缺点：

1. 系统的复杂性增加了，多引入了一个组件（消息总线）和一个服务（专用的数据复制服务）
2. 因为返回业务线数据插入成功时，数据还不一定插入到T2中，因此数据有一个不一致时间窗口（这个窗口很短，最终是一致的）
3. 在消息总线丢失消息时，冗余表数据会不一致

如果想解除“数据冗余”对系统的耦合，引出常用的第三种方案

方法三：线下异步冗余



数据的双写不再由好友中心服务来完成，而是由线下的一个服务或者任务来完成，如上图1-6流程：

1. 业务方调用服务，新增数据

2. 服务先插入T1数据

3. 服务返回业务方新增数据成功

4. 数据会被写入到数据库的log中

5. 线下服务或者任务读取数据库的log

6. 线下服务或者任务插入T2数据

优点：

1. 数据双写与业务完全解耦

2. 请求处理时间短（只插入1次）

缺点：

1. 返回业务线数据插入成功时，数据还不一定插入到T2中，因此数据有一个不一致时间窗口（这个窗口很短，最终是一致的）

2. 数据的一致性依赖于线下服务或者任务的可靠性

上述三种方案各有优缺点，可以结合实际情况选取。

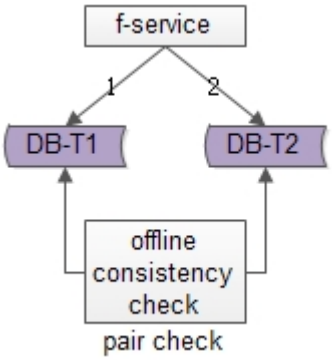
数据冗余固然能够解决多对多关系的数据库水平切分问题，但又带来了新的问题，**如何保证正表T1与反表T2的数据一致性呢？**

八、如何保证数据的一致性

上一节的讨论可以看到，不管哪种方案，因为两步操作不能保证原子性，总有出现数据不一致的可能，**高吞吐分布式事务是业内尚未解决的难题**，此时的**架构优化方向，并不是完全保证数据的一致，而是尽早的发现不一致，并修复不一致。**

最终一致性，是高吞吐互联网业务一致性的常用实践。更具体的，保证数据最终一致性的方案有三种。

方法一：线下扫面正反冗余表全部数据



如上图所示，线下启动一个离线的扫描工具，不停的比对正表T1和反表T2，如果发现数据不一致，就进行补偿修复。

优点：

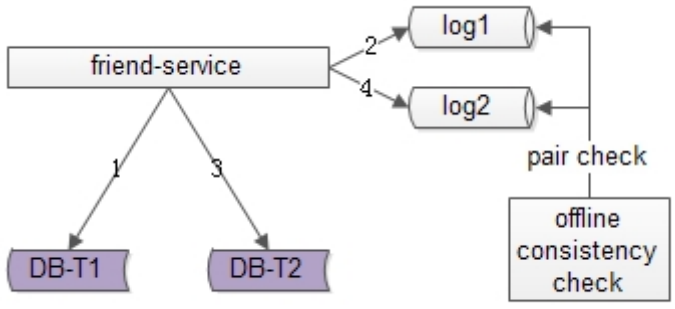
1. 比较简单，开发代价小
2. 线上服务无需修改，修复工具与线上服务解耦

缺点：

1. 扫描效率低，会扫描大量的“已经能够保证一致”的数据
2. 由于扫描的数据量大，扫描一轮的时间比较长，即数据如果不一致，不一致的时间窗口比较长

有没有只扫描“可能存在不一致可能性”的数据，而不是每次扫描全部数据，以提高效率的优化方法呢？

方法二：线下扫描增量数据



每次只扫描增量的日志数据，就能够极大提高效率，缩短数据不一致的时间窗口，如上图1-4流程所示：

1. 写入正表T1
2. 第一步成功后，写入日志log1
3. 写入反表T2

4. 第二步成功后，写入日志log2

当然，我们还是需要一个离线的扫描工具，不停的比对日志log1和日志log2，如果发现数据不一致，就进行补偿修复

优点：

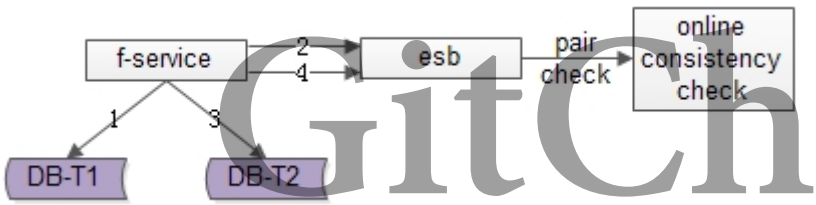
1. 虽比方法一复杂，但仍然是比较简单的
2. 数据扫描效率高，只扫描增量数据

缺点：

1. 线上服务略有修改（代价不高，多写了2条日志）
2. 虽然比方法一更实时，但时效性还是不高，不一致窗口取决于扫描的周期

有没有实时检测一致性并进行修复的方法呢？

方法三：实时线上“消息对”检测



这次不是写日志了，而是向消息总线发送消息，如上图1-4流程所示：

1. 写入正表T1
2. 第一步成功后，发送消息msg1
3. 写入反表T2
4. 第二步成功后，发送消息msg2

这次不是需要一个周期扫描的离线工具了，而是一个实时订阅消息的服务不停的收消息。

假设正常情况下，msg1和msg2的接收时间应该在3s以内，如果检测服务在收到msg1后没有收到msg2，就尝试检测数据的一致性，不一致时进行补偿修复

优点：

1. 效率高
2. 实时性高

缺点：

1. 方案比较复杂，上线引入了消息总线这个组件
2. 线下多了一个订阅总线的检测服务

however，技术方案本身就是一个投入产出比的折衷，可以根据业务对一致性的需求程度决定使用哪一种方法。

九、总结

文字较多，希望尽量记住如下几点：

- 好友业务是一个典型的多对多关系，又分为强好友与弱好友 。
- 数据冗余是一个常见的多对多业务数据水平切分实践。
- 冗余数据的常见方案有三种：
  - 服务同步冗余
  - 服务异步冗余
  - 线下异步冗余
- 数据冗余会带来一致性问题，高吞吐互联网业务，要想完全保证事务一致性很难，常见的实践是最终一致性。
- 最终一致性的常见实践是，尽快找到不一致，并修复数据，常见方案有三种。
  - 线下全量扫描法
  - 线下增量扫描法
  - 线上实时检测法

十、还有哪些未尽事宜

以订单中心为典型的“多KEY”类业务的水平拆分架构又应该怎么处理，敬请期待下期。