数据高可用架构设计与实现

大型企业如何实现 MySQL 到 Redis 的同步

前面曾提到过 Read/Write Through 和 Cache Aside 这几种更新缓存的模式或者说策略，这几种策略都存在缓存不命中的可能性，如果缓存没有命中，就需要直接访问数据库以获取数据。

一般情况下，只要提前做好缓存预热，使缓存的命中率保持在一个相对比较高的水平上，那么直接访问数据库的请求比例就会非常低，这种情况下。一般没有什么问题。但是，如果是一个超大规模的系统或极高并发的情况下那就又不一样了。

缓存不命中

构建 Redis 集群后，由于集群可以水平扩容，因此只要集群足够大，理论上支持海量并发就不是问题。但是如果并发请求数量的基数过大，那么即便只有很小比率的请求直接访问数据库其绝对数量也仍然不小，再加上促销活动的流量峰值，还是会存在系统雪崩的风险。

那么，这个问题该如何解决呢？其实方法很简单，让所有请求都落在缓存上。硬件的价格一般总体是向下的，只要预算足够，Redis 集群的容量理论上就是无限的。我们可以把全量数据都放在 Redis 集群中，处理读请求的时候，只需要读取 Redis，而不用访问数据库，这样就完全没有“缓存不命中”的风险了。实际上，很多大型互联网公司都在使用这种方法。

不过在 Redis 中缓存全量数据，又会引发一个新的问题。那就是缓存中的数据应该如何更新呢？因为现在是从缓存中直接读到数据，则可以直接返回，如果没能读到数据，那就只能返回错误了。所以，当系统更新数据库的数据之后,必须及时更新缓存。

至此，我们又要面对一个老问题：如何保证 Redis 中的数据与数据库中的数据一致性？分布式事务当然可以来解决数据一致性的问题，但是不太适合用来更新缓存，至少大部分的分布式事务实现对数据更新服务有很强的侵入性，而且如果 Redis 本身出现了故障，写入数据失败则还会导致事务全部失败的问题，相当于是降低了服务的性能和可用性。

一个可行的方法是，启动一个更新订单缓存的服务接收数据变更的消息队列(Message Queue，MQ)中的消息，然后注意解决消息的可靠性问题即可，这种方式实现起来很简单，也没有什么侵入性。

使用 Binlog 实时更新 Redis 缓存

但是如果我们要缓存的数据原本就没有一份数据更新的消息队列可以订阅，又该怎么办呢？其实很多大型互联网企业所采用的更通用的解决方案是使用 Binlog 实时更新 Redis 缓存。

数据更新服务只负责处理业务逻辑，更新 MySQL 中的数据，完全不用考虑如何更新缓存。负责更新缓存的服务，把自己伪装成一个 MySQL 的从节点。从 MySQL 接收并解析 Binlog 之后，就可以得到实时的数据变更信息，然后该服务就会根据这个变更信息去更新 Redis 缓存。

订阅 Binlog 更新缓存的方案，相较于上文中接收消息更新 Redis 缓存的方案，两者的实现思路其实是一样的，都是异步实时订阅数据变更信息以更新 Redis 缓存。只不过,直接读取 Binlog 这种方式通用性更强。

除此之外，由于在整个缓存更新链路上，减少了一个收发消息队列的环节，从 MySQL 更新到 Redis 更新的时延变得更短，出现故障的可能性也更低，这也是为什么很多大型互联网企业更青睐于采用这种方案的原因。

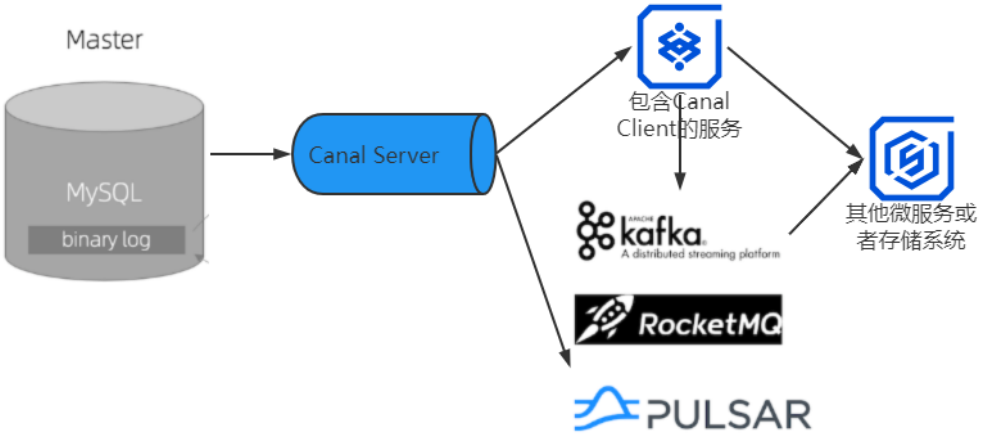
订阅 Binlog 更新缓存的方案唯一的缺点是：自行实现比较复杂，该方案毕竟不像接收消息那样,收到的直接就是订单数据，解析 Binlog 需要对 MySQL 的底层相当熟悉，还是挺麻烦的。

很多开源的项目都提供了订阅和解析 MySQL Binlog 的功能，在我们的商城项目中就使用了比较常用的开源项目 Canal 来实时接收 Binlog 更新 Redis 缓存。

Canal 详解

Canal 是阿里开源的一个项目，官方主页：https://github.com/alibaba/canal。

它通过模拟 MySOL 主从复制的交互协议，把自己伪装成一个 MySOL 的从节点，向 MySOL 主节点发送 dump 请求。MySOL 收到请求后，就会向 Canal 开始推送 Binlog，Canal 解析 Binlog 字节流之后，将其转换为便于读取的结构化数据，供下游程序订阅使用。实际运用后的运行架构如图：



可以看到 Canal 有个服务端，在模拟 MySOL 从节点获得数据库服务器的数据后，我们可以使用一个包含 Canal Client 的服务程序获得 Canal 服务端解析出的数据，也可以通过配置让 Canal 服务端直接将数据发送给 MQ，当然我们的 Canal Client 程序经过数据处理后也可以发送给 MQ。不管是经过 Canal Client 程序还是直接发给 MQ，接下来还可由第三方的服务或者存储系统进行后续处理。

商城项目的实现

安装运行 Canal 服务

Canal 服务端是用 Java 编写的，所以 Windows 和 Linux 下都可以运行，考虑到大多数同学会在本机进行测试，我们以 Windows 下的安装运行为例进行说明，目前环境为操作系统 Windows11，MySQL 8.0.19。

MySQL 配置

既然 Canal 模拟 MySQL 从节点获得数据库服务器的数据，很明显，对 MySQL 服务器的配置完全可以参考 MySQL 的主从复制中主节点的配置。怎么样让 MySQL 成为主节点？

先开启 Binlog 写入功能，配置 binlog-format 为 ROW 模式

```
2  show VARIABLES like "%log_bin%"
```

信息 | 结果 1 | 剖析 | 状态

Variable_name	Value
log_bin	ON

```
2  show VARIABLES like "%binlog_format%"
```

信息 | 结果 1 | 剖析 | 状态

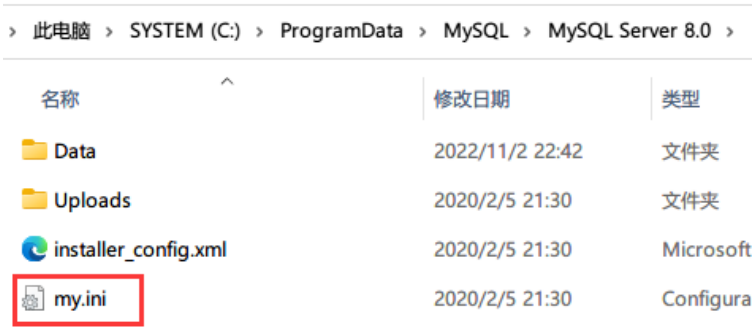
Variable_name	Value
binlog_format	ROW

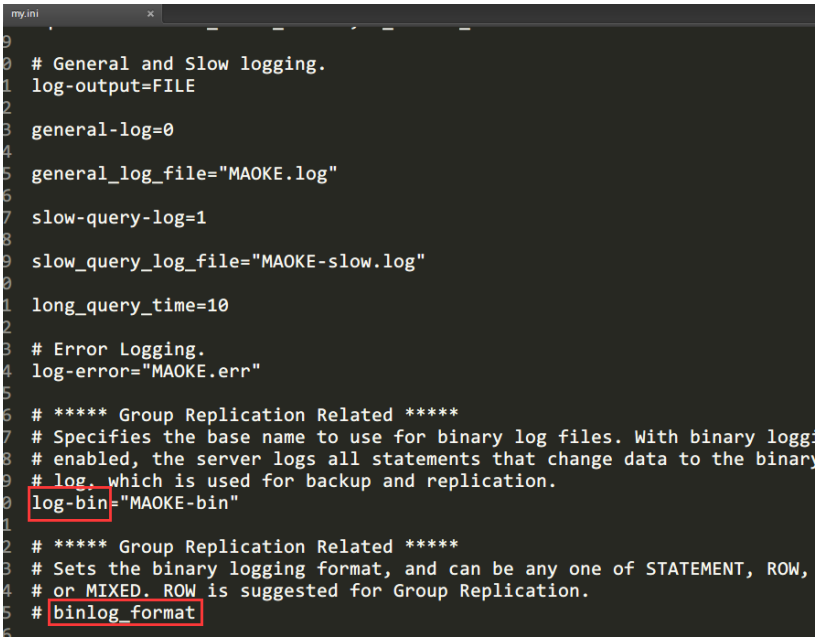
```
2  show VARIABLES like "%server_id%"
```

信息 | 结果 1 | 剖析 | 状态

Variable_name	Value
server_id	1

如果本机的 MySQL 的配置与上述不符合，可以修改 MySQL 的配置文件并重启 MySQL，一般来说是 my.ini，大多情况下这个文件放在 MySQL 的安装目录或者 C:\ProgramData 的对应 MySQL 的目录下





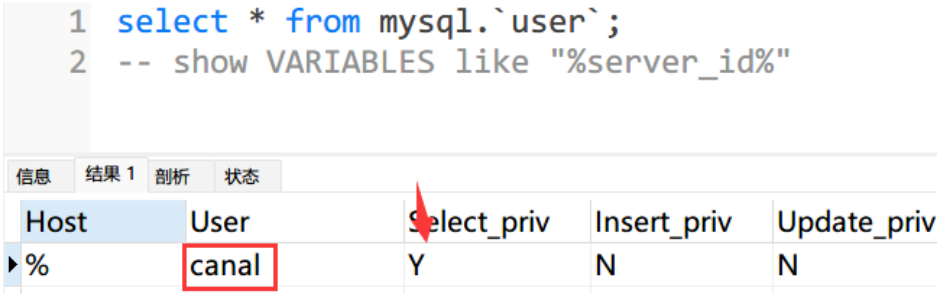
既然是将 Canal 模拟 MySQL 从节点，自然还要给 Canal 设置一个用来复制数据的 MySQL 账号，我们这里设定这个用户名为 Canal，密码 Canal

CREATE USER canal IDENTIFIED BY 'canal';

GRANT SELECT, REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO 'canal'@'%';

FLUSH PRIVILEGES;

此时在 MySQL 的表中就应该存在 canal 用户，拥有 select、repl 权限



当然要完成主从复制，还有一步，找到当前 binlog 的进度让从 MySQL 使用

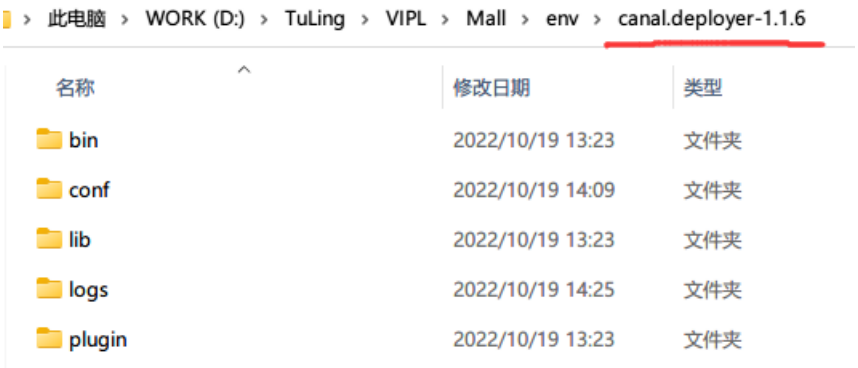
```
1  show master STATUS
```

信息 | 结果 1 | 剖析 | 状态

File	Position	Bir
MAOKE-bin.000049	155	

Canal 服务端

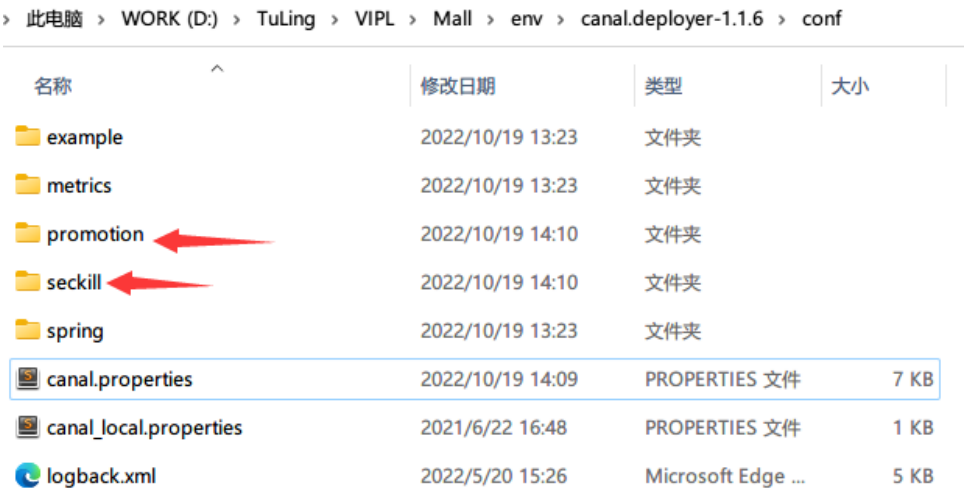
把得到的 Canal 服务端程序解压缩



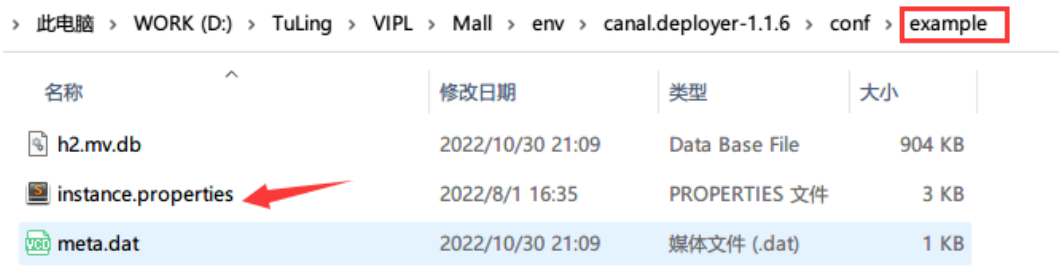
进入 conf 目录，修改 canal.properties 文件，比较关键的是 canal.destinations

```
#################################################
#########       destinations        #############
#################################################
canal.destinations = promotion,seckill
# conf root dir
canal.conf.dir = ../conf
# auto scan instance dir add/remove and start/stop
```

这里表示，我们需要监控与促销 promotion、秒杀 seckill 相关的数据变动，而相关的数据库、表配置会分别放在 promotion、seckill 目录下，如果没有这两个目录需要新建这两个目录



进入 promotion 目录，修改 instance.properties，如果目录中没有这个文件，可以从 example 目录拷贝一个过来



对于 instance.properties 的修改比较关键的就是几处，一是 MySQL 主服务的连接配置

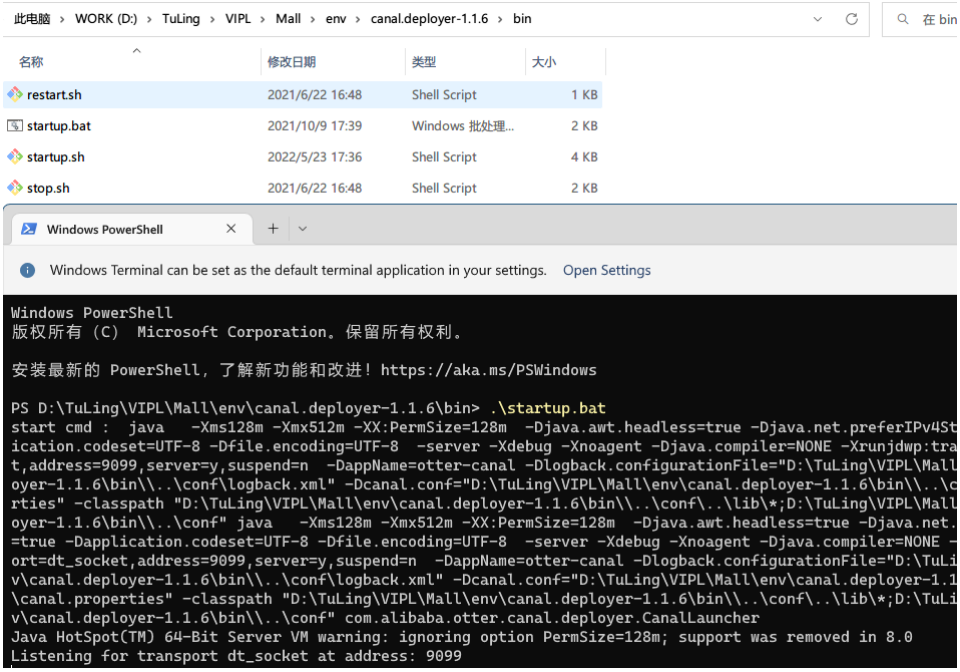
```
# position info
canal.instance.master.address=127.0.0.1:3306
canal.instance.master.journal.name=MAOKE-bin.000049
canal.instance.master.position=155
canal.instance.master.timestamp=
canal.instance.master.gtid=
```

```
# username/password
canal.instance.dbUsername=canal
canal.instance.dbPassword=canal
```

二则是要对哪些相关的业务表进行监视，比如我们这里是 promotion 促销信息，数据放在 tl_mall_promotion 库中：

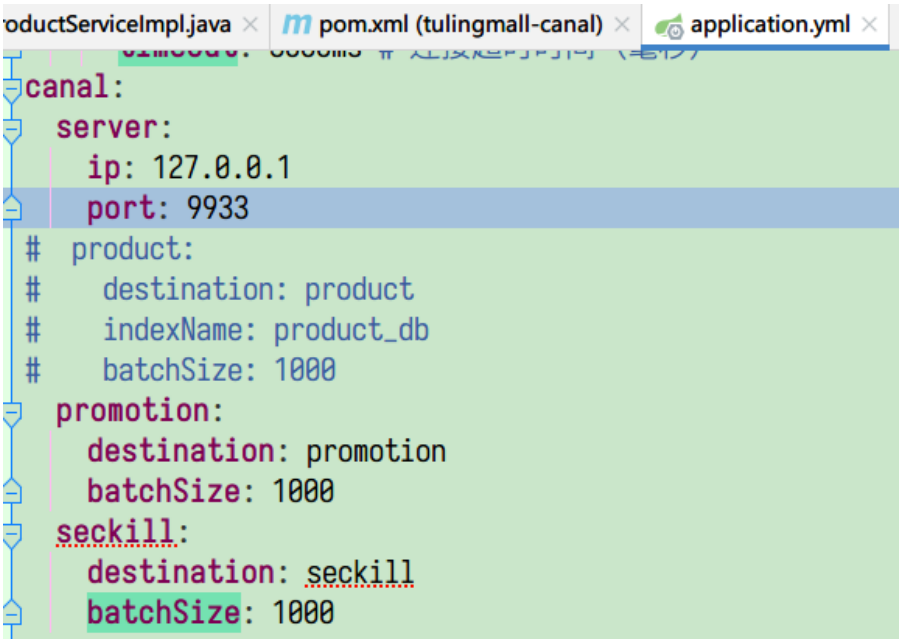
```
# table regex
canal.instance.filter.regex=tl_mall_promotion.sms_home_advertise,tl_ma
```

配置完成后，进入 bin 目录，执行 startup.bat 即可



tulingmall-canal 的实现

完成了 Canal 服务端配置后，接下来我们就可以实现包含 Canal Client 的服务了。当然，首先要把 Canal 服务端的相关信息配置到我们的代码中



和 Maven 配置

```
<!-- 引入canal -->
<dependency>
    <groupId>com.alibaba.otter</groupId>
    <artifactId>canal.client</artifactId>
    <version>1.1.4</version>
    <exclusions>
        <exclusion>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-client</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

就可以进入实际的开发了。

在我们的 PromotionData 类中，主要的业务就是发现促销信息相关的表发生变动后删除促销信息在 Redis 中保存的键值对，以方便缓存的更新。具体解释请参考代码注释或者视频的讲解。

基于 Binlog 实现跨系统实时数据同步

前面说过当数据量太大的时候，如果单个存储节点存不下，就需要分片存储数据。

数据分片之后,数据的查询操作就会受到诸多限制。比如如果将用户 ID 作为分片键对订单表进行分片，那就只能根据用户 ID 维度来查询。这样,商家就会无法查询自家店铺的订单。当然强行查询也不是不行，只是要在所有分片上都查询一遍，再把结果聚合起来，整个过程又慢又麻烦，实际意义不大。

对于这样的需求，目前普遍采取的解决方案是用空间换时间、毕竟如今存储设备越来越便宜。再存一份订单数据到商家订单库,然后以店铺 ID 作为分片键进行分片,专门供商家查询订单之用。

另外对于同一份商品数据，如果是按照关键字搜索，放在 ES 中会比放在 MySQL 中更合适，毕竟 ES 就是做搜索的，所以在我们的 tulingmall-canal 中的 ProductESData 就是负责将商品数据的变化从 MySQL 同步到 ElasticSearch。

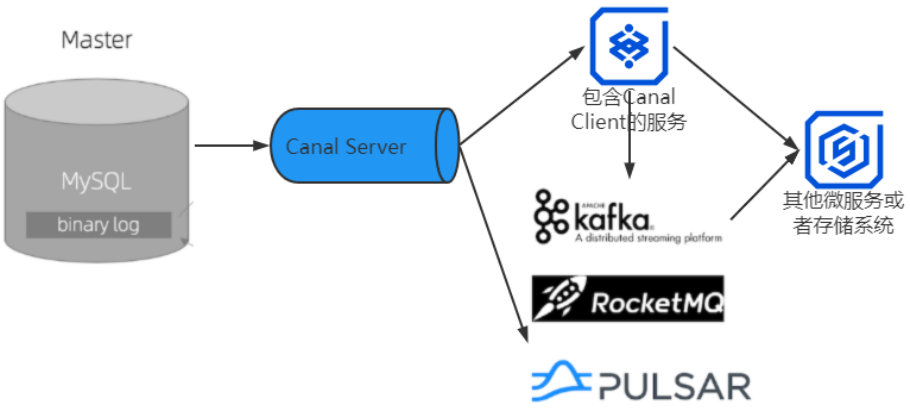
所以在大规模系统中，对于海量数据的处理原则都是根据业务对数据查询的需求反过来确定选择什么数据库、如何组织数据结构、如何分片数据等之类的问题，这样才能获得最优的查询性能。

在大型互联网企业中、其核心业务数据，以不同的数据结构和存储方式，保存几十甚至上百份，都是非常正常的。

那么如何才能做到让这么多份数据实时地保持同步呢？分布式事务解决不了大规模数据的实时同步问题。

前面我们已经看到如何利用 Canal 把自己伪装成一个 MySQL 的从库，从 MySQL 数据库中实时接收 Binlog，然后修改 Redis 缓存。所以实现异构数据库的同步也可以采用这个方法。

当然为了能够支撑下游的众多数据库，从 Canal 出来的 Binlog 数据肯定不能直接写入下游的众多数据库中。原因也很明显：一是写不过来；二是下游的每个数据库，在写入之前可能还要处理一些数据转换和过滤的工作。所以一般我们会增加一个消息队列来解耦上下游。



更换数据库

随着系统规模的逐渐增大，我们迟早会面临需要更换数据库的问题，比如下面这几种常见的情况。

对 MySQL 做了分库分表之后，需要从原来的单实例数据库迁移到新的数据库集群上。

系统从传统部署方式向云上迁移的时候,也需要从自建的数据库迁移到云数据库上。

当 MySQL 的性能不够用的时候，一些在线分析类的系统需要更换成一些专门的分析类数据库,比如 HBase。

更换数据库需要面临非常大的技术挑战,因为需要保证在整个迁移过程中，既不能长时间停止服务，也不能丢失数据。

如何在不停机的情况下，安全地迁移数据、更换数据库呢？

实现不停机更换数据库

墨菲定律:“如果事情有变坏的可能,不管这种可能性有多小,它总会发生。”

对应到更换数据库这件事情上，就是在更换数据库的过程中，只要有一点可能会出问题的地方，哪怕出现问题的概率非常小，它都会出问题。

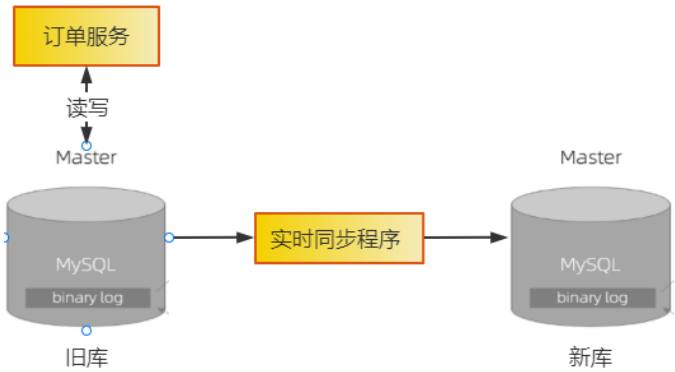
实际上无论是新版本的程序还是新的数据库，即使我们做了严格的验证测试，实现了高可用方案，对于刚刚上线的系统，它的稳定性也是不够好的。需要有一个磨合的过程，才能逐步达到一个稳定的状态，这是客观规律。这个过程中一旦出现故障，如果不能及时恢复，那么其所造成的损失往往是我们难以承担的。

所以我们在设计迁移方案的时候，**一定要保证每一步都是可逆的。也就是必须保证，每执行完一个步骤，一旦出现任何问题，都能快速回滚到上一个步骤。**这是设计这种升级类技术方案的时候比较容易忽略的问题。

我们还是以订单库为例来说明这个迁移方案应该如何设计。

1、首先要做的一点是，把旧库的数据全部复制到新库中。因为旧库还在服务线上业务，所以不断会有订单数据写入旧库，我们不仅要向新库复制数据，还要保证新旧两个库的数据是实时同步的。所以，需要用一个同步程序来实现新旧两个数据库的实时同步。

可以使用 Binlog 实现两个异构数据库之间数据的实时同步。这一步不需要回滚，因为这里只增加了一个新库和一个同步程序，对系统的旧库和程序没有任何改变。即使新上线的同步程序影响到了旧库，停掉同步程序也就可以了。

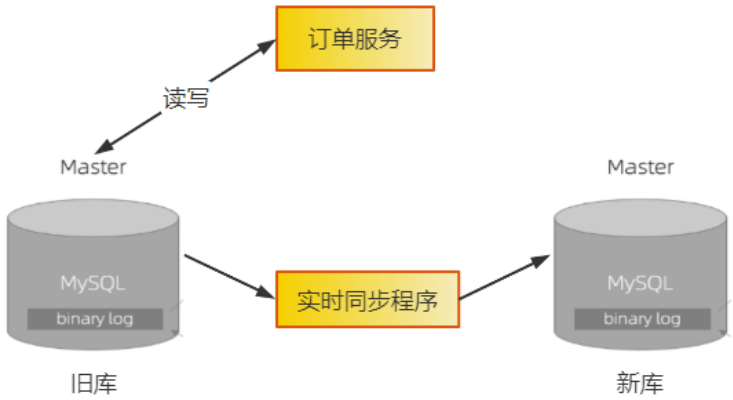


2、然后需要改造一下订单服务，业务逻辑部分不需要变动，数据访问的 DAO 层需要进行如下改造：

1）支持双写新旧两个库，并且预留热切换开关，能通过开关控制三种写状态:只写旧库、只写新库和同步双写。

2）支持读取新旧两个库，同样预留热切换开关，控制读取旧库还是新库。

3、然后上线新版的订单服务，这个时候订单服务仍然是只读写旧库，不读写新库。让这个新版的订单服务稳定运行至少一到两周的时间，其间我们不仅要验证新版订单服务的稳定性，还要验证新旧两个订单库中的数据是否保持一致。这个过程中，如果新版订单服务出现任何问题，都要立即下线新版订单服务，回滚到旧版本的订单服务。

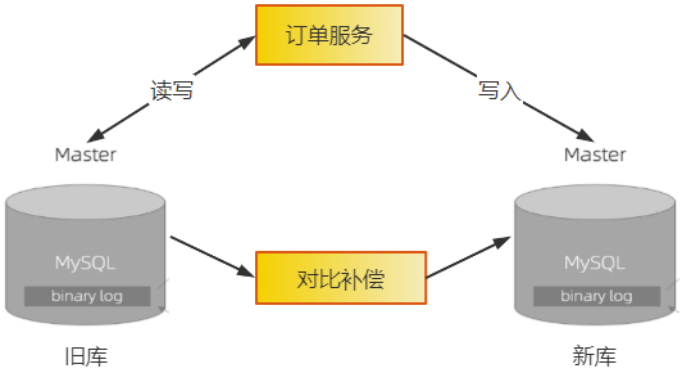


3、稳定一段时间之后，就可以开启订单服务的双写开关了。开启双写开关的同时，需要停掉同步程序。这里有一个需要特别注意的问题是，这里双写的业务逻辑，一定是先写旧库，再写新库，并且以旧库的结果为准。

如果旧库写成功，新库写失败，则返回成功，但这个时候要记录日志，后续我们会根据这个日志来验证新库是否还有问题。如果旧库写失败，则直接返回失败，同时也不再写新库了。这么做的原因是不能让新库影响到现有业务的可用性和数据准确性。上面这个过程如果出现任何问题都要关闭双写,回滚到只读写旧库的状态。

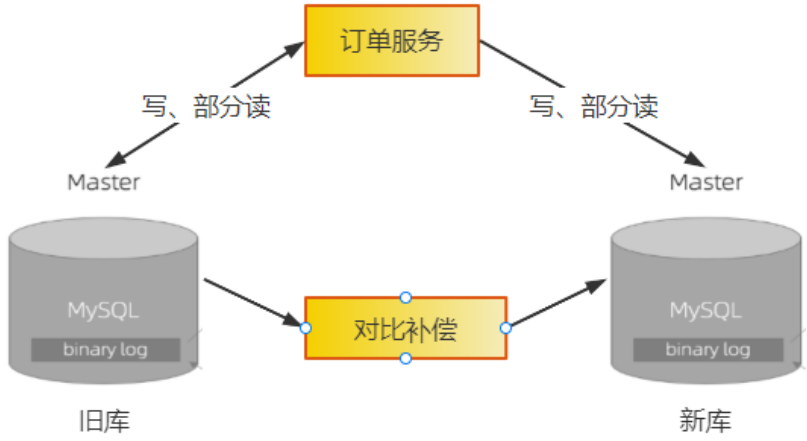
切换到双写之后,新库与旧库的数据可能会出现不一致的问题。原因有两点：一是停止同步程序和开启双写，这两个过程很难做到无缝衔接；

二是双写的策略也不能保证新旧库的强一致性。对于这个问题,我们需要上线一个比对和补偿的程序，用于比对旧库最近的数据变更，然后检查新库中的数据是否一致，如果不一致，则需要进行补偿。



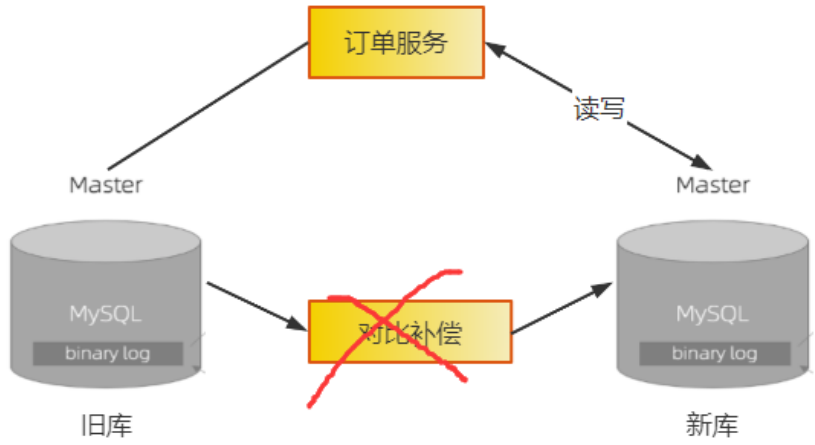
开启双写之后，还需要稳定运行至少几周的时间，并且在这期间我们需要不断地检查，以确保不能有旧库写成功、新库写失败的问题。如果在几周之后比对程序发现新旧两个库的数据没有不一致的情况，那就可以认为新旧两个库的数据一直都是保持同步的。

4、接下来就可以用类似灰度发布的方式把读请求逐步切换到新库上。同样，运行期间如果出现任何问题，都要再切回到旧库。



5、将全部读请求都切换到新库上之后，其实读写请求已经全部切换到新库上了，虽然实际的切换已经完成,但后续还有需要收尾的步骤。

再稳定一段时间之后，就可以停掉比对程序，把订单服务的写状态改为只写新库。至此，旧库就可以下线了。注意，在整个迁移过程中,只有这个步骤是不可逆的。由于这一步的主要操作就是摘掉已经不再使用的旧库，因此对于正在使用的新库并不会有什么影响，实际出问题的可能性已经非常小了。



ps：如果这一步也需要可逆怎么办？

双写切换为新库单写这一步不可逆的主要原因是，一旦切换为新库单写，旧库的数据与新库的就不一致了，这种情况是无法再切换回旧库的。所以问题的关键是，切换为新库单写后，需要保证旧库的数据能与新库保持同步。这时双写需要增加一种过渡状态：从双写以旧库为准过渡到双写以新库为准。然后把比对和补偿程序反过来，用新库的数据补偿旧库的数据。这样就可以做到一旦出现问题，就直接切回到旧库上。但是这样做一般成本比较高。

至此们完成了在线更换数据库的全部流程。双写版本的订单服务也完成了它的历史使命，可以在下一次升级订单服务版本的时候下线双写功能。

数据表的变更，如果只是新增表，这个很简单，一般直接回退到旧版本程序即可；但如果牵涉到表字段的变化就麻烦些，但是也可以采用类似的思路，双写新旧表并设计热切换开关。

实现比对和补偿程序

在上面的数据库切换过程中，如何实现比对和补偿程序是个切换设计方案中的一个难点。这个比对和补偿程序的实现难点在于，我们要比对的是两个随时都在变化的数据库中的数据。在这种情况下，我们没有类似复制状态机这样理论上严谨、实际操作还很简单的方法来实现比对和补偿。但我们还是可以根据业务数据的实际情况,有针对性地实现比对和补偿，经过一段时间之后，把新旧两个数据库的差异逐渐收敛到一致。

像订单这类时效性比较强的数据，是比较容易进行比对和补偿的。因为订单一旦完成之后，几乎就不会再改变了，比对和补偿程序就可以根据订单完成时间，每次只比对这个时间窗口内完成的订单。补偿的逻辑也很简单，发现不一致的情况后，直接用旧库的订单数据覆盖新库的订单数据就可以了。

这样，切换双写期间，对于少量不一致的订单数据，等到订单完成之后，补偿程序会将其修正。后续在双写的时候只要新库不是频繁写入失败，就可以保证两个库的数据完全一致。

比较麻烦的是更一般的情况，比如像商品信息之类的数据，随时都有可能会发生变化。如果数据上带有更新时间,那么比对程序就可以利用这个更新时间，每次从旧库中读取一个更新时间窗口内的数据，到新库中查找具有相同主键的数据进行比对，如果发现数据不一致，则还要比对一下更新时间。如果新库数据的更新时间晚于旧库数据，那么很可能是比对期间数据发生了变化，这种情况暂时不要补偿，放到下个时间窗口继续进行比对即可。另外，时间窗口的结束时间不要选取当前时间，而是要比当前时间早一点，比如 1 分钟之前，这样就可以避免比对正在写入的数据了。

如果数据没带时间戳信息，那就只能从旧库中读取 Binlog，获取数据变化信息后到新库中查找对应的数据进行比对和补偿。

安全地实现数据备份和恢复

对于任何一个企业来说，数据安全的重要性不言而喻。能够影响数据安全的事件，都是极小概率的事件(比如数据库宕机、磁盘损坏甚至机房着火，还有大家喜欢调侃的“程序员不满老板删库跑路”)，但这些事件一旦发生,我们的业务就会遭受惨重损失。

一般来说，由存储系统导致的比较严重的损失主要有两种情况。第一种情况是数据丢失造成的直接财产损失。比如订单数据丢失造成了大量的坏账。为了避免这种损失，系统需要保证数据的高可靠性。第二种情况是，由于存储系统的损坏，造成整个业务系统停止服务而带来的损失。比如，电商系统停服期间造成的收入损失。为了避免这种损失，系统需要保证存储服务的高可用性。

所谓防患于未然，一个系统从设计的第一天起,就需要考虑今后在出现各种问题的时候，如何保证该系统的数据安全性。

保证数据安全，最简单且有效的方法就是定期备份数据，这样无论因为出现何种问题而导致的数据损失，都可以通过备份来恢复数据。但是如何备份才能最大程度地保证数据安全还是需要仔细考虑的。

2018 年曾出现过一次重大故障，某著名云服务商因为硬盘损坏，导致多个客户数据全部丢失。通常来说，一个大的云服务商，数据通常都会有多个备份，即使硬盘损坏，也不会导致数据丢失的重大事故，但是因为各种各样的原因，最终的结果是数据的三个副本都被删除，数据丢失无法找回。

所以并不是简单地定期备份数据就可以高枕无忧了。我们最常用的 MySQL 如何更安全地实现数据的备份和恢复呢？

最简单的备份方式就是全量备份。备份的时候把所有的数据复制一份，存放到文件中，恢复的时候再把文件中的数据复制回去，这样就可以保证恢复之后，数据库中的数据与备份时的数据是完全一样的。在 MySQL 中，我们可以使用 mysqldump 命令执行全量备份。

比如全量备份数据库 test 的命令

```
$ mysqldump -uroot -p test > test.sql
```

备份出来的文件是一个 SQL 文件，文件的内容就是创建数据库、表，写入数据等之类的 SOL 语句，如果要恢复数据，则直接执行这个备份的 SQL 文件就可以了

不过全量备份的代价非常高，为什么这么说呢?

首先备份文件包含了数据库中的所有数据，占用的磁盘空间非常大；其次，每次备份操作都要拷贝大量的数据，备份过程中会占用数据库服务器大量的 CPU 和磁盘 IO 资源、同时为了保证数据一致性，备份过程中很有可能会锁表。这此都会导致在备份期间数据库本身的性能严重下降。所以我们不能频繁地对数据库执行全量备份操作。

一般来说，在生产系统中每天执行一次全量备份就已经是非常频繁的了。这就意味着，如果数据库中的数据丢失了就只能恢复到最近一次全量备份的那个时间点，这个时间点之后的数据是无法找回的。也就是说，因为全量备份的代价比较高不能频繁地执行备份操作,所以全量备份不能做到完全无损的恢复。

既然全量备份代价太高不能频繁执行，那么有没有代价较低的备份方法,能让我们的数据少丢失甚至不丢失呢？增量备份可以达到这个目的。相比于全量备份，增量备份每次只用备份相对于上一次备份发生了变化的那部分数据，所以增量备份的速度更快。

MySQL 自带的 Binlog 就是一种实时的增量备份工具。Binlog 所记录的就是 MySQL 数据变更的操作日志。开启 Binlog 之后，MySQL 中数据的每次更新操作，都会记录到 Binlog 中。Binlog 是可以回放的，回放 Binlog，就相当于是把之前对数据库中所有数据的更新操作，都按顺序重新执行一遍，回放完成之后数据自然就恢复了。这就是 Binlog 增量备份的基本原理。很多数据库都有类似于 MySQL Binlog 的日志工具，原理也与 Binlog 相同，备份和恢复的方法也与之类似。

通过定期的全量备份配合 Binlog，我们可以把数据恢复到任意一个时间点，再也不怕“删库跑路”了。详细的命令，可以参考 MySQL 官方文档中的“备份和恢复”相关章节。

在执行备份和恢复的时候，大家需要特别注意如下两个要点。

第一，也是最重要的“不要把所有的鸡蛋放在同一个篮子中”，无论是全量备份还是 Binlog，都不要与数据库存放在同一个服务器上。最好能存放到不同的

机房，甚至不同城市离得越远越好。这样即使出现机房着火、光缆被挖断甚至地震也不怕数据丢失。

第二，在回放 Binlog 的时候，指定的起始时间可以比全量备份的时间稍微提前一点儿，这样可以确保全量备份之后的所有操作都在恢复的 Binlog 范围内，从而保证数据恢复的完整性。

注意：为了确保回放的幂等性，需要将 Binlog 的格式设置为 ROW 格式。

本文档分享地址：

http://note.youdao.com/noteshare?id=3b6d5fbcbf4753c88ef940b82359f93f&sub=970C4B89C18848DEA21D3B6C7CD426D8