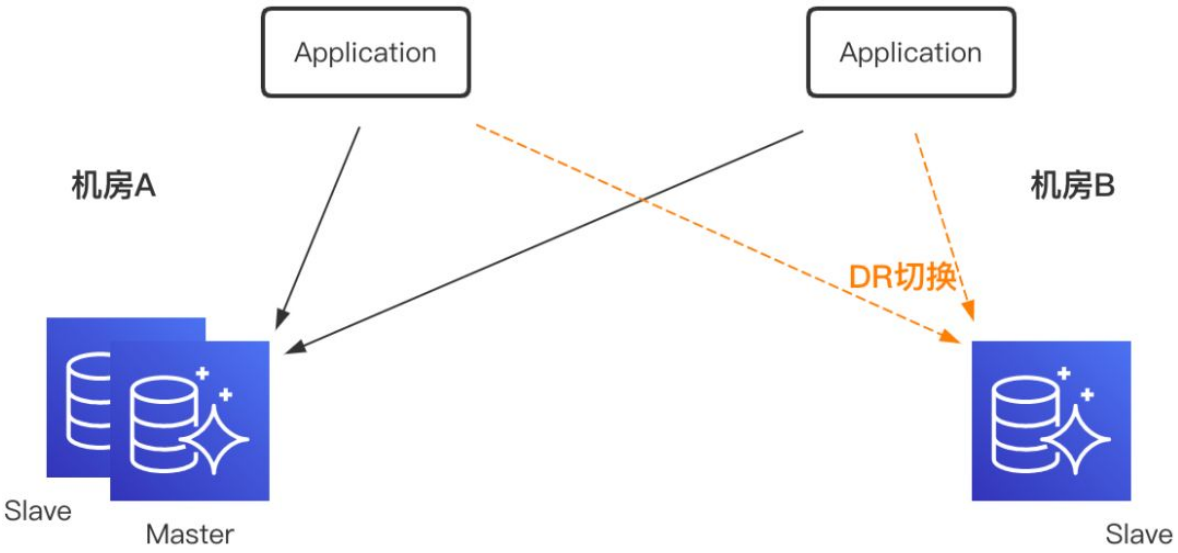
携程异地多活-MySQL实时双向（多向）复制实践

一、前言

携程内部MySQL部署采用多机房部署，机房A部署一主一从，机房B部署一从，作为DR（Disaster Recovery）切换使用。当前部署下，机房B部署的应用需要跨机房进行写操作；当机房A出现故障时，DBA需要手动对数据库进行DR切换。

为了做到真正的数据异地多活，实现MySQL同机房就近读写，机房故障时无需进行数据库DR操作，只进行流量切换，就需要引入数据实时双向（多向）复制组件。

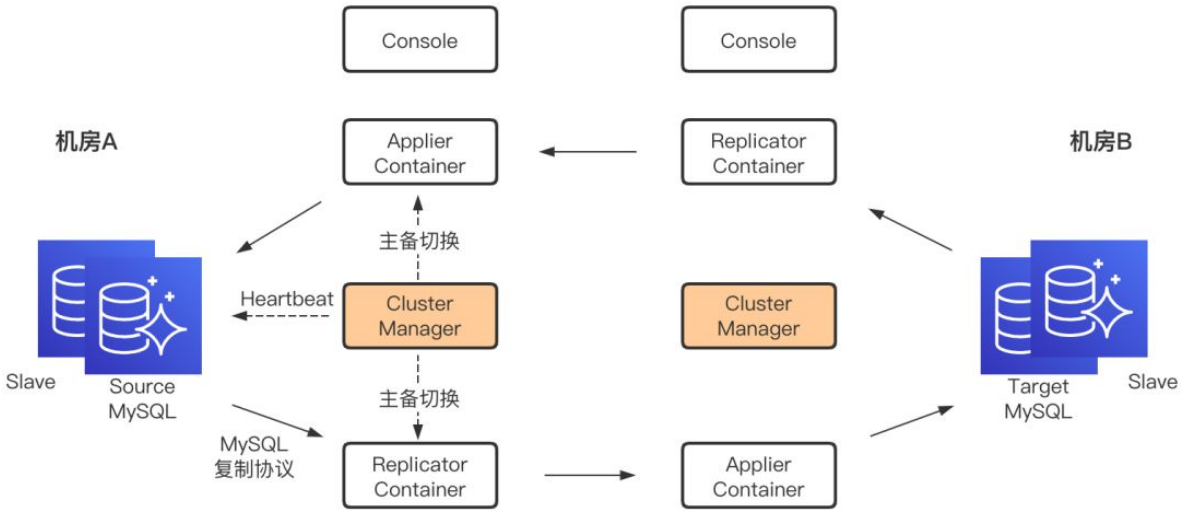


二、DRC 介绍

DRC（Data Replicate Center）是携程框架架构研发部推出的用于数据双向或多向复制的数据库中间件，在公司G2（高品质Great Service、全球化Globalization）战略的背景下，服务于异地多活项目，赋予了业务全球化的部署能力。

三、DRC 架构设计

DRC采用服务端集中化设计，配合另一数据库访问中间件DAL（Data Access Layer）的本地读写功能，实现数据就近访问。



模块介绍

- Replicator Container

Replicator Container 实现对 Replicator 实例的管理，一个 Replicator 实例表示对一个MySQL集群的复制单元，Instance将自己伪装为MySQL的Slave，实现Binlog的拉取和本地存储。

- Applier Container

Applier Container实现对Applier 实例的管理，一个Applier 实例连接到一个Replicator 实例，实现对Replicator 实例本地存储Binlog的拉取，进而解析出SQL语句并应用到目标MySQL，从而实现数据的复制。

- Cluster Manager

Cluster Manager负责集群高可用切换，包括由于MySQL主从切换导致的Replicator 实例和Applier 实例重启，以及Replicator 实例与Applier 实例自身主从切换引起的新实例启动通知。

- Console

Console提供UI操作、外部系统交互API以及监控告警。

四、DRC 详细设计

4.1 接入DB规范

DRC的核心指标包括复制延迟和数据一致性。

为了实现数据复制的低延迟，Applier能够快速应用SQL，就需要每个表至少包含主键或者唯一键，加速执行效率；同时在保证数据准确的前提下，SQL应该尽量并行复制，需要MySQL开启从5.7.22版本引入的Writeset功能。

为了保证数据复制的准确性，在主备切换时Replicator仍能准确定位Binlog位点，需要MySQL开启GTID；当数据复制发生冲突时，为了具备自动解决冲突的能力，需要表包含时间戳列，并精确到毫秒。

这就需要接入DRC的MySQL数据库满足：

1）5.7.22及以上版本；

2）Master上开启Writeset并行复制；

3）MySQL开启GTID；

4）每个表包含时间戳列，精确到毫秒；

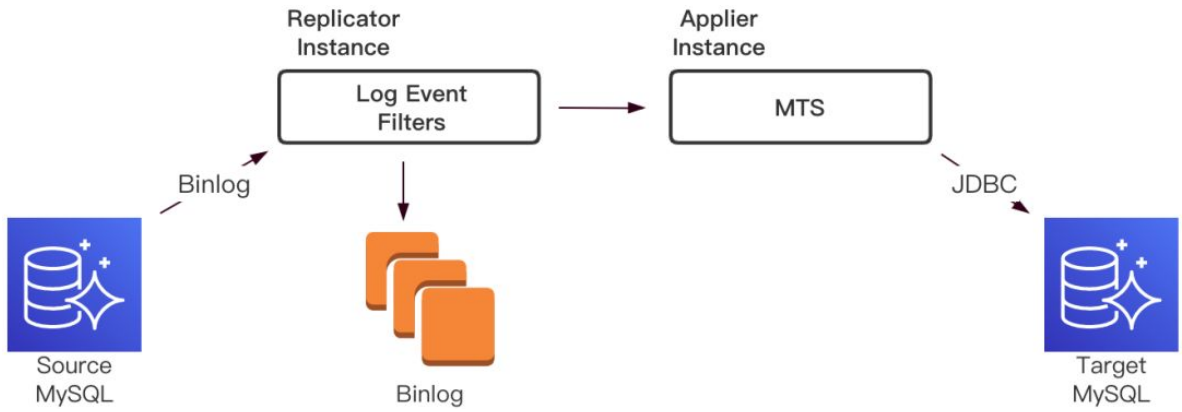
5）每个表至少包含主键或者唯一键。

DRC的复制依赖GTID（Global Transaction ID），这里先简单介绍一下GTID的概念。MySQL 5.6.5版本新增了一种基于GTID的复制方式，强化了数据库的主备一致性，故障恢复以及容错能力，取代传统的基于file和position主从复制，使得在MySQL主备切换时，仍能准确定位到Binlog位点。

GTID的格式形如：source_id:transaction_id，其中source_id表示MySQL服务器的uuid，transaction_id是在事务提交的时候系统顺序分配的一个序列号。

4.2 Binlog 复制

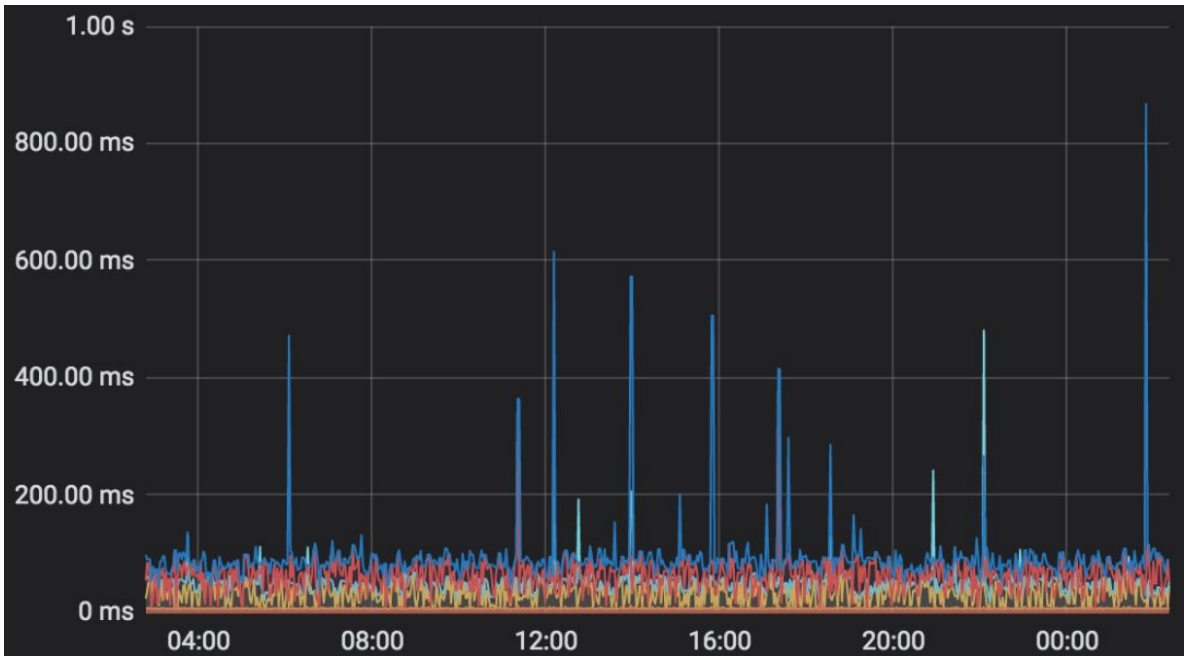
单向复制链路包含拉取Binlog并持久化到本地磁盘的Replicator，和请求Binlog且并行应用到目标MySQL的Applier。整个链路涉及的I/O操作包括网络传输和磁盘读写。



4.2.1 低复制延迟

为了降低复制延迟，就要求复制链路中每一环都尽可能高效。网络层通信模型使用异步I/O；系统层尽可能使用操作系统提供的**Zero Copy**和**Page Cache**；应用层提高数据处理并行度以及降低系统不可用时间。

监控显示生产环境业务双向复制延迟999线 < 1s。下面就介绍一下DRC在降低复制延迟方面所做的性能优化工作。



1）网络层

Replicator采用GTID复制方式，实现了MySQL复制协议，伪装成源MySQL的Slave拉取Binlog。网络层通信组件采用携程开源组件XPipe(https://github.com/ctripcorp/x-pipe)，实现网络交互异步化。

2）系统层

接收Binlog时，从数据流中解析出不同类型的Event，直接保存在**堆外内存**。

每个Event需要经过一组过滤器，进而决定是否需要落盘持久化。

> 对于Heartbeat类型的Event需要过滤丢弃；针对某些不需要进行数据同步的库和表，需要丢弃相应Event，减少存储量和传输量；

高性能IO的要点

> 对于需要持久化的Event，直接将堆外内存中的数据写入文件Page Cache并定时刷入磁盘，减少数据复制和IO操作，降低处理耗时，提升Replicator拉取效率。

发送Binlog时，当Applier进度落后Replicator，需要从磁盘读取，这时只解析gtid_event事件，其他需要发送的事件直接从磁盘读取到堆外内存进行发送，减少数据复制。

3）应用层

Applier借鉴原生MySQL基于Writeset的并行复制，内嵌了基于水位的并行算法，高效的将SQL应用到目标数据库。

除去正常复制之外，为了降低系统的不可用时间，就需要系统在异常情况下，尽快恢复正常功能。比如断网恢复时，为了避免一端使用老连接，就需要对连接进行空闲检测；为了应对断网导致数据堆积出现流量突增，就需要对流量进行控制。

4）空闲检测

Replicator 与MySQL、Applier和Replicator通过Netty进行数据传输，当网络出现故障，可能一端仍然使用老连接进行通信，会导致数据复制出现中断。

针对网络故障，Replicator对MySQL添加了读空闲检测，启动时设置MySQL空闲时间隔10s发送一次heartbeat_event，如果30s没有收到MySQL任何事件，则认为MySQL出现问题，发起重连。

Replicator对Applier设置了写空闲检测，当没有Event需要发送给Applier时，间隔10s发送一次heartbeat_event，如果发送失败，则认为Applier出现问题，断开连接。

Applier对Replicator设置了读空闲检测，如果30s没有收到Replicator任何事件，则认为Replicator出现问题，发起重连。

5）流量控制

设计上Replicator Container使用物理机，其中会运行若干Replicator实例，Applier Container使用虚拟机，这样会造成发送和消费的速率不匹配。

尤其当Applier由于某种原因出现故障后，在Replicator端堆积大量未消费的Event，重启后如果堆积的Event全部发送过来，可能会直接打垮Applier，这样就需要在Replicator实例上对Applier进行限流。

Replicator发送端使用Netty提供的**WRITE_BUFFER_WATER_MARK高低水位的变化来控制流控的开关**，进而动态调整发送速率，整形平滑流量。

4.2.2 数据一致性

为了保证数据的一致，就需要满足：

1）数据拉取时保证时序；

2）数据拉取不能遗漏，SQL应用时不重，或者即使重复，要保证幂等操作，保证At Least Once；

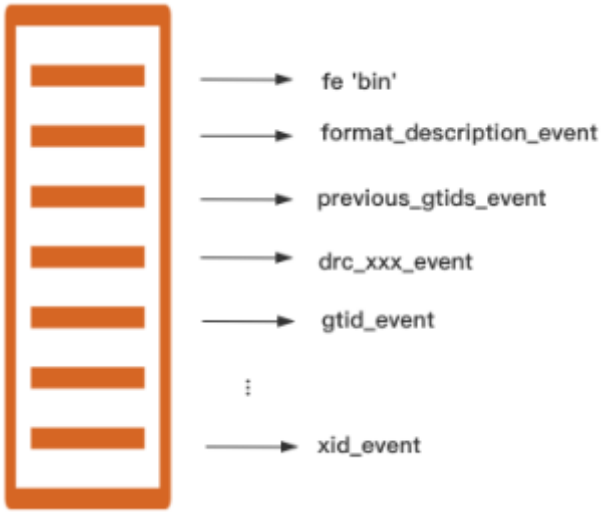
3）数据冲突时，能正确处理，保证数据最终一致。

下面就看下DRC是如何保证以上3个要求。

1）时序保证

本地磁盘保存Binlog采用原生的存储协议，Replicator顺序处理接收到每一个Event事件。

存储协议兼容MySQL原生的mysqlbinlog命令，其中根据DRC自身的需要，保存了自定义的一些辅助事件，比如DDL事件，表结构事件。消费时顺序发送Binlog文件中的事件给Applier。



2）At Least Once

为了实现At Least Once，需要解决3个子问题：

1）Replicator或者Applier重启时，如何保证请求的GTID set准确体现目前的消费偏移？

2）双向（多向）复制如何解决循环复制？

3）Applier由于异常重复拉取时，如何保证幂等？

下面逐一介绍每个子问题的解决方案。

断点重续

当Replicator重启时，会从本地磁盘中恢复已经拉取过的GTID set：

1）定位重启前使用的最后一个Binlog文件；

2）解析出previous_gtids_event；

3）遍历该文件的所有gtid_event，与previous_gtids_event解析出的GTID set取并集。

恢复过程中，会校验文件的正确性，对于没有以xid_event结束的事务，Replicator会对文件进行截断，对应的gtid事务会重新请求。

当Applier重启时，Cluster Manager会从目标数据库中查询出当前已经执行过的GTID set发送给Applier，Applier带着该参数向Replicator发送Binlog拉取请求。Replicator收到请求中的GTID set，从本地磁盘中定位出第一个需要发送的Event所在的Binlog文件，依次遍历该文件中的每一个Event，针对gtid_event事件取出其中的gtid，判断该gtid对应的事务是否包含在GTID set中，如果包含其中，则表示Applier已经消费过，无需发送，否则通过堆外内存直接将Event发送给Applier。

循环复制

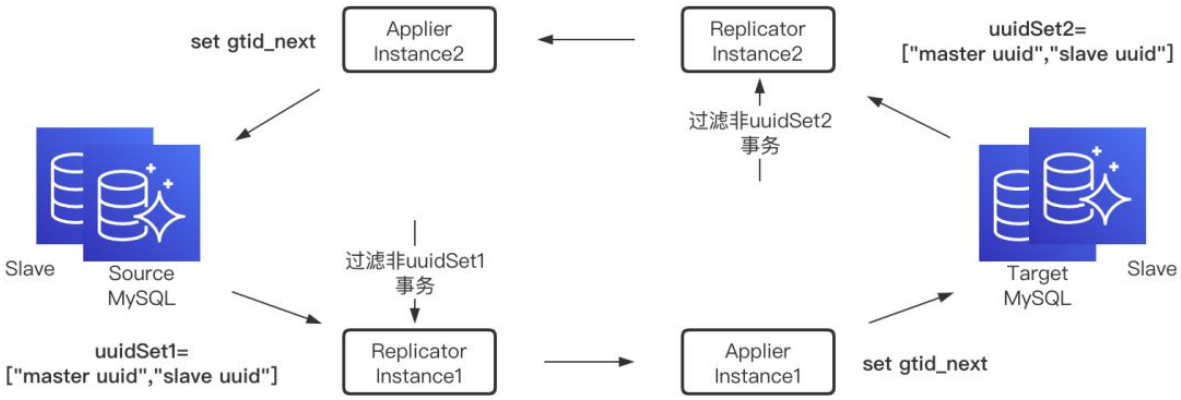
单向复制时，经过DRC复制到对端的SQL在执行后，同样会落到MySQL的Binlog中，这样在双向(多向)复制结构中，对端的Replicator Instance在拉取到该条Binlog后如果继续复制，就会出现循环复制的问题。

针对循环复制，业内可选的解决方案是在Binlog事务开头插入一条写操作，标识出该条事务是DRC复制过来，而不是真实业务写入，这样对端Replicator发现一个事务开头包含DRC特殊标记时，就不会继续复制该事务。

分析MySQL自身主从复制，Slave在收到Master同步过来的Binlog时，通过set gtid_next将该事务的GTID设置为同步过来的gtid_event中的GTID，这样就实现了主从GTID set的一致性。

如果将Replicator拉取Binlog类比为Slave的I/O线程，磁盘文件类比为Relay log，Applier类比为Slave的SQL线程，那么Applier是可以采用同样的方式，使用set gtid_next设置经过DRC复制到对端事务的GTID，这样源和目标数据库的GTID set会保持一致，更重要的是可以标识出该事务是经DRC复制过来的。这也是DRC最终采用的破解循环复制的方案。

如下双向复制结构，Replicator Instance1只会同步源MySQL集群uuidSet1中的服务器产生事务，Replicator Instance2只会同步目标MySQL集群uuidSet2中的服务器产生事务。如果业务在源MySQL集群写入一条数据，Replicator Instance1从gtid_event中的GTID解析出uuid属于uuidSet1，那么会持久化到磁盘并发送给Applier Instance1，Applier Instance1接收到事务中包含的所有Event后，执行set gtid_next=GTID，然后通过JDBC将SQL写入目标MySQL，完成单向复制；Replicator Instance2接收到gtid_event后，同样解析出GTID，但是uuid并不属于uuidSet2，这样该条事务就会被过滤，从而避免的循环复制。



幂等

Applier如果重复接收到相同GTID的事务，由于MySQL会记录已经执行的GTID set，如果该GTID已经被执行，则会自动忽略，这样即使Applier重复应用同一条事务，也不会对业务产生影响。

小结

从上面可以看到，在保证数据一致性时，GTID不论是在Replicator和Applier重启后Binlog位点定位，标识Binlog来源避免循环复制，还是Applier重复应用时幂等实现，都起到了至关重要的作用。

3）冲突解决

设计上，首先要避免冲突的出现：

1）接入Set化的业务在流量入口处就会根据uid进行分流，同一个用户的流量进入同一个机房；数据接入层中间件DAL同样会采用local-2-local的路由策略。这样同一条记录在2个机房同时被修改的情况很少发生；

2）对于使用自增ID的业务，通过不同机房设置不同的自增ID规则，或者采用分布式全局ID生成方案，避免双向复制后数据冲突。

如果数据确实出现了冲突，2个机房对同一条数据进行的修改，这时需要根据冲突处理策略进行处理：

1）Applier根据默认的冲突处理策略进行处理，接入DRC的表都有一个精确到毫秒自动更新的时间戳，冲突时时间戳靠后的会被采用，进而实现数据的一致；

2）冲突的SQL会被监控记录，连同数据库中的原始数据同时提供给用户，进而自助决定是否需要进行覆盖。

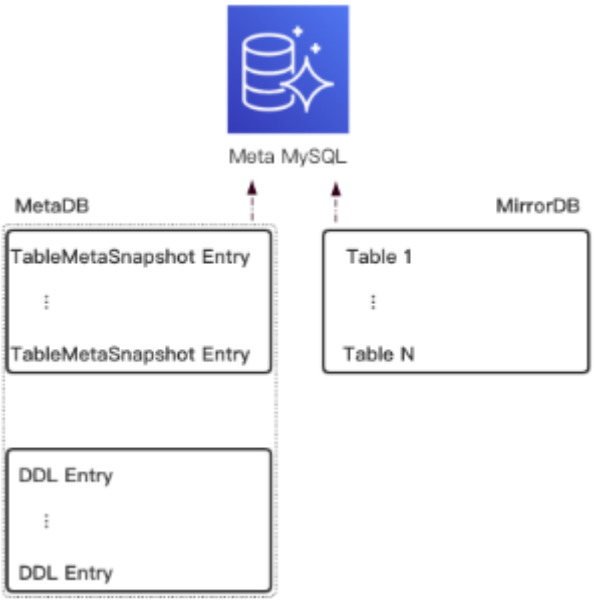
4.3 DDL 支持

DDL操作会引起表结构的变更，在复制链路中Applier需要表结构信息解析对应时刻的Binlog Event，当Applier消费速率落后Replicator的发送速率时，就需要历史版本的表结构信息才能够正确解析Binlog Event。

这就引入了表结构设计第一个问题：历史版本如何存储？

为了存储表结构，势必首先要获得表结构，如果从源MySQL直接抓取表结构，由于Binlog是异步发送，就导致抓取到DDL的Binlog时刻，与MySQL上表结构未必能够一一对应，从而引起Applier解析出现问题，进而导致数据不一致。这就引入表结构设计第二个问题：表结构从何处抓取？

业界通用的解决方案是基于独立的第3方数据库进行表结构单独存储管理。数据库本身就是存储工具，Snapshot表和DDL表分别保存表结构快照和DDL变更记录，这样任意时刻的表结构等于Snapshot及其后DDL变更集合，则第一个表结构存储问题顺其自然得以解决；独立数据库镜像一份源数据库的库表结构，每次从Binlog接收到DDL Event后，将解析出的DDL语句直接应用到镜像数据库，随即抓取相应表结构即可，这样就解决了第二个表结构从何处抓取的问题。



独立数据库解决方案的缺点是引入外部依赖，降低了系统的可用性，提高了运维成本。

4.3.1 表结构存储和计算

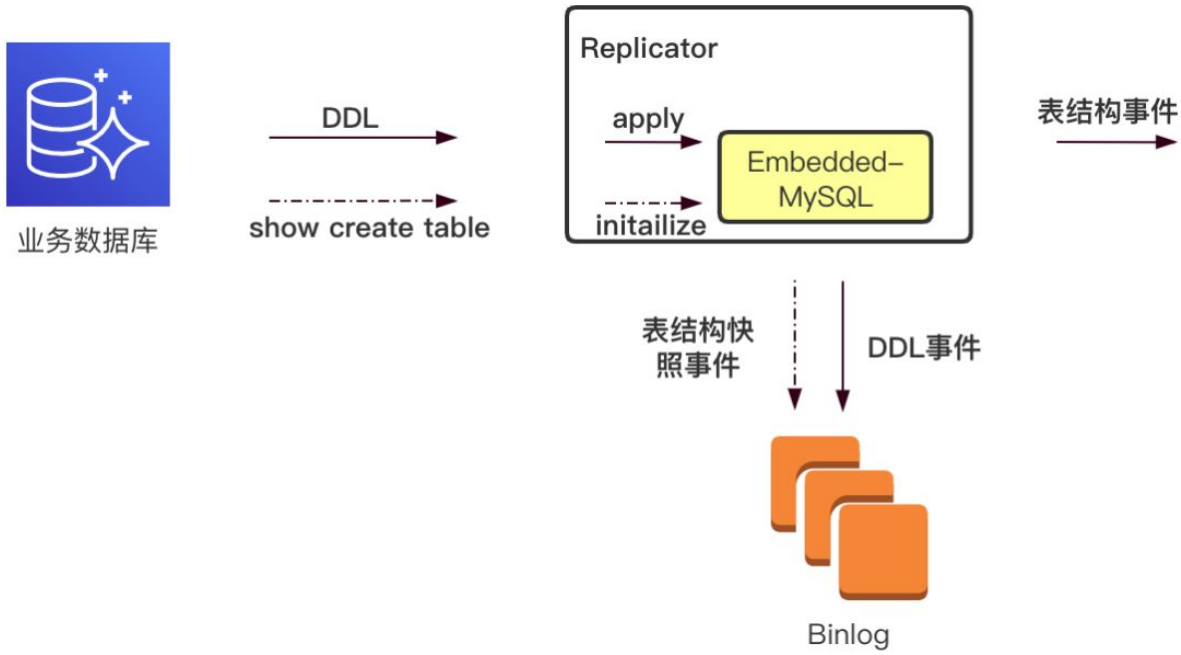
针对DDL功能中问题一：

从数据库中查询Snapshot和DDL记录的好处是时间顺序容易确定，能够简单准确的恢复表结构。那么是否有其他存储介质，在保存表结构快照和DDL操作的同时，能够保证时序呢？有，保存Binlog的文件就具有这种特性，DRC采用了这种基于Binlog的表结构文件存储方案。

针对DDL功能中问题二：

镜像数据库是为了实时计算出DDL变更后最新的表结构信息，在存储不使用独立部署的数据库后，DRC引入嵌入式轻量数据库，降低外部依赖和系统运维成本。

这样整体的设计方案如下图所示：



Binlog文件头会保存自定义表结构快照事件，当从接收的Event事件检测到DDL后，保存为自定义的DDL事件。这样当Applier连接上Replicator后，总是会根据GTID set定位到需要的第一个历史版本表结构所在的文件，从而实时恢复表结构历史，用于后续Binlog Event的解析。

我们将数据库最小依赖打成独立的Jar包服务，每个Replicator实例启动时，会一并启动一个独立的嵌入式数据库，在恢复GTID set的同时，根据表结构快照事件和DDL事件重建嵌入式数据库中表结构。

4.3.2 DDL 入口

携程内部发布DDL是通过gh-ost进行变更，gh-ost会在影子表中执行DDL操作，等影子表中数据同步完成后，业务低峰期进行原表和影子表的切换。

针对gh-ost，需要追踪gh-ost变更过程中内部形如_xxx_gho的表的DDL所有操作，最终执行切换时检测出rename操作，保存对应表结构最新信息发送给Applier即可。

同时针对数据库直接进行的DDL操作，直接检测出DDL类型的Event即可。

4.3.3 DDL 异常处理

对于接入DRC的数据库，当在进行DDL变更时，可能会出现两边数据库变更不同步，单侧进行了DDL变更，另一侧未进行变更。针对新增列这种场景，Applier在保证数据一致的前提下，对新增列的值进行比较，如果Binlog中解析出的值和该列的默认值一致，则会剔除该列，继续数据复制。这样在另一侧补上DDL变更后，两侧的数据最终仍然一致。

4.4 监控告警

DRC核心指标包括复制延迟和数据一致性。除此之外我们还提供BU、应用和IDC维度的监控：

1）流量和TPS监控告警；

2）BU、应用和IDC维度的监控告警；

3）DDL变更监控；

4）表结构一致性监控告警；

5）数据冲突监控；

6）GTID set GAP监控。

五、总结

本次分享围绕DRC的核心指标复制延迟和数据一致性，介绍了复制过程中对性能的优化以及各种场景如何保证数据的一致性。针对DDL，分别支持gh-ost和直接DDL操作，实现在线表结构变更不影响数据复制。

后续DRC的工作会集中在高可用、海外支持上以及外围设施的建设上，为携程的国际化战略提供数据层面的支撑。

【作者简介】Roy，携程软件技术专家，负责MySQL双向同步DRC和数据库访问中间件DAL的开发演进，对分布式系统高可用设计、数据一致性领域感兴趣。