MySQL的binlog解析与应用

MySQL的binlog

日志是MySQL数据库的重要组成部分，记录着数据库运行期间各种状态信息。MySQL日志主要包括错误日志、查询日志、慢查询日志、事务日志、二进制日志几大类。其中MySQL的binlog用于记录数据库执行的写入性操作（不包括查询）信息，以二进制的形式保存在磁盘中。binlog是MySQL的逻辑日志，并且由Server层进行记录，使用任何存储引擎的MySQL数据库都会记录binlog日志。

* 逻辑日志：可以简单理解为记录的就是SQL语句。
* 物理日志：因为MySQL数据最终是保存在数据页中的，物理日志记录的就是数据页变更。

BinLog事件

Mysql已经经历了多个版本的发布，最新已经到8.x，然而目前企业中主流使用的还是Mysql 5.6或5.7。不同版本的Mysql中，binlog的格式和事件类型可能会有些细微的变化，不过暂时我们并不讨论这些细节。

总的来说，**binlog中存储的内容称之为二进制事件(binary log event)，每一个数据库更新操作(Insert、Update、Delete，不包括Select)等都对应一个事件，这是本文讲述的重点。**，binlog主要分为2种格式：

* **Statement模式** **：** binlog中记录的就是我们执行的SQL；
* **Row模式：** binlog记录的是每一行记录的每个字段变化前后得到值。

熟悉主从复制的同学，应该知道，还有第三种模式**Mixed**(即混合模式)，从严格意义上来说，这并不是一种新的binlog格式，只是将Statement和Row两种模式进行结合而已。

当我们选择不同的binlog模式时，在binlog文件包含的事件类型也不相同，如:

* **1)** 在Statement模式下，我们就看不到Row模式下独有的事件类型。
* **2)** 有一些类型的event，必须在我们开启某些特定配置的情况下，才会出现；
* **3)** 当然也会有一些公共的event类型，在任何模式下都会出现。

多文件存储
---

mysql 将数据库更新操作对应的event记录到本地的binlog文件中，显然在一个文件中记录所有的event是不可能的，过大的文件会给我们的运维带来麻烦，如删除一个大文件，在I/O调度方面会给我们带来不可忽视的资源开销。

因此，目前基本上所有支持本地文件存储的组件，如MQ、Mysql等，都会控制一个文件的大量。在数据量较多的情况下，就分配到多个文件进行存储。

在mysql中，我们可以通过”show binary logs”语句，来查看当前有多少个binlog文件，以及每个binlog文件的大小，如下：

```
mysql> show binary logs;
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| mysql-bin.000001 |       829 |
| mysql-bin.000002 |       124 |
+------------------+-----------+
```

另外，mysql提供了：

* **max_binlog_size**配置项，用于控制一个binlog文件的大小，默认是1G
* **expire_logs_days**配置项，可以控制binlog文件保留天数，默认是

0，也就是永久保留。

在实际生产环境中，一般无法保留所有的历史binlog。因为一条记录可能会变更多次，记录依然是一条，但是对应的binlog事件就会有多个。在数据变更比较频繁的情况下，就会产生大量的binlog文件。此时，则无法保留所有的binlog文件。

在mysql的percona分支上，还提供了**max_binlog_files**配置项，用于设置可以保留的binlog文件数量，以便我们更精确的控制binlog文件占用的磁盘空间。这是一个非常有用的配置，本人曾经遇到一个库，大约10分钟就会产生一个binlog文件，也就是1G，按照这种增长速度，1天下来产生的binlog文件，就会占用大概144G左右的空间，磁盘空间可能很快就会被使用完。因此，我们可以显示的控制binlog文件的数量，例如指定50，binlog文件最多只会占用50G左右的内存。

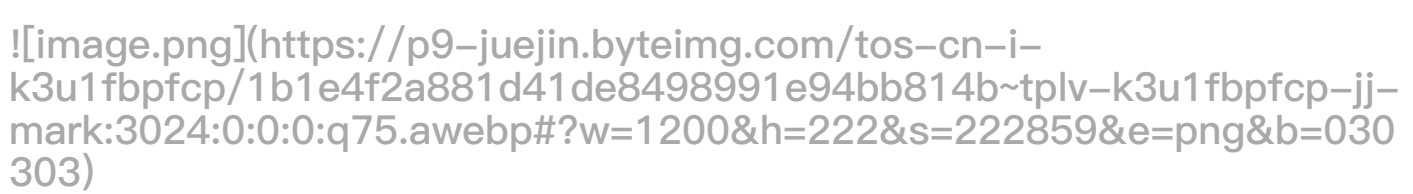
在更高版本的mysql中，支持按照秒级精度，来控制binlog文件的保留时间。

管理事件

所谓binlog管理事件，你可以认为是一些在任何模式下都有可能会出现的事件，不管你的配置binlog_format是Row、Statement还是Mixed。

在当前binlog v4版本中，每个binlog文件总是以**Format Description Event**作为开始，以**Rotate Event**结束作为结束。如果你使用的是很古老的Mysql版本中，开始事件也有可能是**START EVENT V3，** 而结束事件是**Stop Event。** 在开始和结束之间，穿插着其他各种事件。

以下通过 **”show binlog events”** 语法进行查看一个空的binlog文件，也就是只包含管理事件，没有其他数据更新操作对应的事件。如下：



在Event_Type列中，我们看到了三个事件类型：

* **Format_desc：** 也就是我们所说的Format Description Event，是binlog文件的第一个事件。在Info列，我们可以看到，其标明了Mysql Server的版本是8.0.21，Binlog版本是4。
* **Previous_gtids：** 该事件完整名称为，PREVIOUS_GTIDS_LOG_EVENT。熟悉Mysql 基于GTID复制的同学应该知道，这是表示之前的binlog文件中，已经执行过的GTID。需要我们开启GTID选项，这个事件才会有值，在后文中，将会详细的进行介绍。
* **Rotate：** Rotate Event是每个binlog文件的结束事件。在Info列中，我们看到了其指定了下一个binlog文件的名称是mysql-bin.000004。

关于 **"show binlog events"** 语法显示的每一列的含义如下：

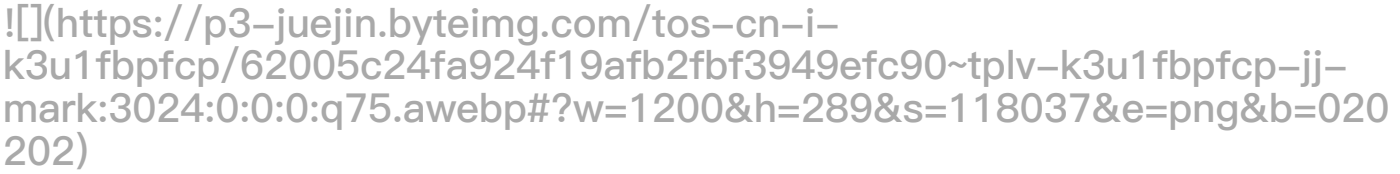
* Log_name：当前事件所在的binlog文件名称
* Pos：当前事件的开始位置，每个事件都占用固定的字节大小，结束位置(End_log_position)减去Pos，就是这个事件占用的字节数。细心的读者可以看到了，第一个事件位置并不是从0开始，而是从4。Mysql通过文件中的前4个字节，来判断这是不是一个binlog文件。这种方式很常见，很多格式的文件，如pdf、doc、jpg等，都会通常前几个特定字符判断是否是合法文件。
* Event_type：表示事件的类型
* Server_id：表示产生这个事件的mysql server_id，通过设置my.cnf中的**server-id**选项进行配置。
* End_log_position：下一个事件的开始位置

Statement模式事件
---

mysql5.0及之前的版本只支持基于语句的复制，也称之为逻辑复制，也就是binary log文件中，直接记录的就是数据更新对应的sql。

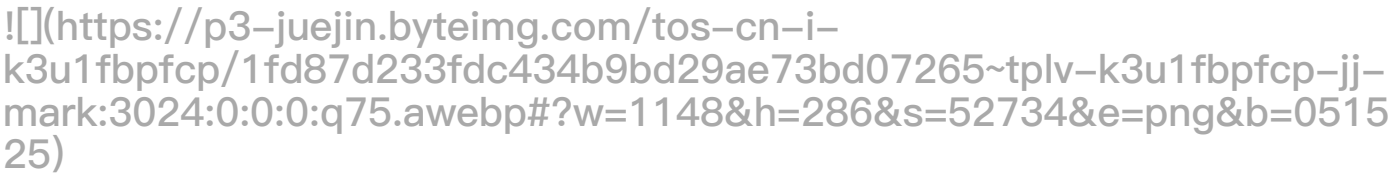
假设有名为test库中有一张user表，如下：

```
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

```

现在，我们往user表中插入一条数据

```
insert into user(name) values(”user1”);
```

之后，可以使用”**show binlog events**” 语法查看binary log中的内容，如下：



红色框架中Event，是我们执行上面Insert语句产生的4个Event。下面进行详细的说明：

**首先，需要说明的是每个事务都是以Query Event作为开始，其INFO列内容为”BEGIN”，以Xid Event表示结束，其INFO列内容为COMMIT。即使对于单条更新SQL我们没有开启事务，Mysql也会默认的帮我们开启事务**。因此在上面的红色框中，尽管我们只是执行了一个INSERT语句，没有开启事务，但是Mysql 默认帮我们开启了事务，所以第一个Event是Query Event，最后一个是Xid Event。

接着，是一个**Intvar Event**，因为我们的Insert语句插入的表中，主键是自增的(AUTO\_INCREMENT)列，Mysql首先会自增一个值，这就是Intvar Event的作用，这里我们看到INFO列的值为INSERT\_ID=1，也就是说，这次的自增主键id为1。需要注意的是，这个事件，只会在Statement模式下出现。

然后，还是一个Query Event，这里记录的就是我们插入的SQL。这也体现了Statement模式的作用，就是记录我们执行的SQL。

Statement模式下还有一些不常用的Event，如**USER\_VAR\_EVENT**，这是用于记录用户设置的变量，仅仅在Statement模式起作用。
```

Row模式事件
----

mysql5.1开始支持基于行的复制，这种方式记录的某条sql影响的所有行记录**变更前**和**变更后**的值。Row模式下主要有以下10个事件：



很直观的，我们看到了INSERT、DELETE、UPDATE操作都有3个版本(v0、v1、v2)，v0和v1已经过时，我们只需要V2版本。

此外，还有一个**TABLE_MAP_EVENT**，这个event我们需要特别，可以理解其作用就是记录了INSERT、DELETE、UPDATE操作的表结构。

下面，我们通过案例演示，ROW模式是如何记录变更前后记录的值，而不是记录SQL。这里只演示UPDATE，INSERT和DELETE也是类似。

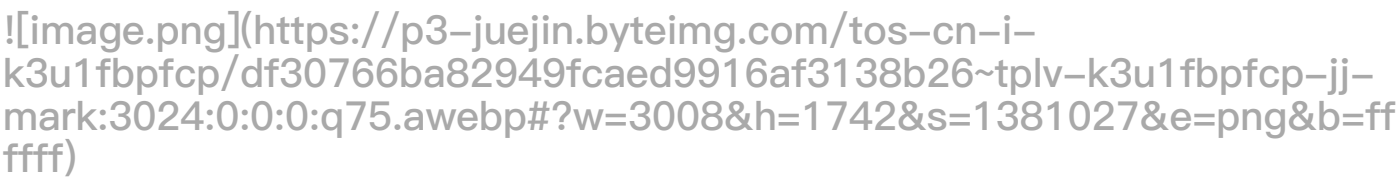
在前面的操作步骤中，我们已经插入了2条记录，如下：

现在需要从Statement模式切换到Row模式，重启Mysql之后，执行以下SQL更新这两条记录：

```
update user set name='user3';
```

在binary log中，会把这2条记录变更前后的值都记录下来，在默认情况下，受到影响的记录行，每个字段变更前的和变更后的值，都会被记录下来，即使这个字段的值没有发生变化 **。**

接着，我们还是通过”show binlog events”语法来验证：



首先我们可以看到的是，在Row模式下，单条SQL依然会默认开启事务，通过Query Event(值为BEGIN)开始，以Xid Event结束。

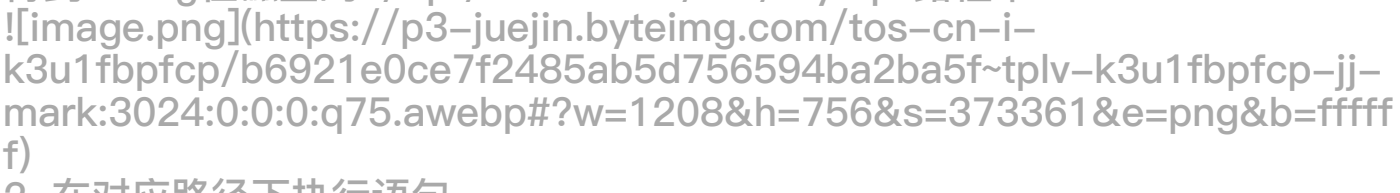
接着，我们看到了一个Table_map 事件，就是前面提到的TABLE_MAP_EVENT，在INFO列，我们可以看到其记录table_id为111，操作的是mysql_binlogs库中user表。

最后，是一个Update_rows事件，然而其INFO，并没有像Statement模式那样，显示一条SQL，我们无法直接看到其变更前后的值是什么。此时，由于存储的都是二进制内容，直接vim无法查看，我们需要借助另外一个工具**mysqlbinlog**来查看其内容。

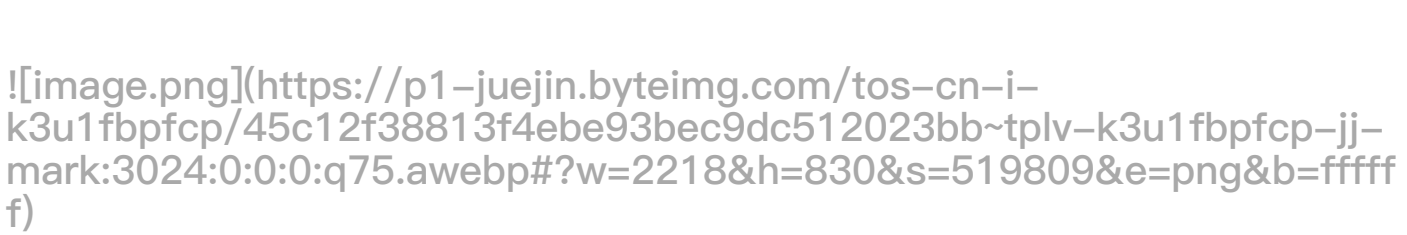
在查看内容前我们需要定位到binlog文件在磁盘的位置（以mac为例）：

1. 在mysql命令行执行语句：

```
show variables like ’log_%’;
```

得到binlog在磁盘的“/opt/homebrew/var/mysql”路径下

2. 在对应路径下执行语句

```
mysqlbinlog --base64-output=”decode-rows” -v binlog.000001
```

```

结果如下：




截图中显示了2个event，第一个红色框就是Table\_map事件，第二个是Update\_rows事件。

在第二个红色框架中，显示了两个Update sql，这是只是mysqlbinlog工具为了方便我们查看，反解成SQL而已。我们看到了WHERE以及SET子句中，并没有直接列出字段名，而是以 **@1**、 **@2**这样的表示字段位于数据库表中的顺序。**事实上，这里显示的内容，WHERE部分就是每个字段修改前的值，而SET部分，则是每个字段修改后的值，** 也就是变更前后的值都会记录。

这里我们思考一下mysqlbinlog工具的工作原理，其可以将二进制数据反解成SQL进行展示。那么，**如果我们可以自己解析binlog，就可以做数据恢复，这并非是什么难事**。例如用户误删除的数据，执行的是DETELE语句，由于Row模式下会记录变更之前的字段的值，我们可以将其反解成一个INSERT语句，重新插入，从而实现数据恢复。

### **binlog\_row\_image参数**

我们经常会看到一些Row模式和Statement模式的比较。ROW模式下，即使我们只更新了一条记录的其中某个字段，也会记录每个字段变更前后的值，binlog日志就会变大，带来磁盘IO上的开销，以及网络开销。

事实上，这个行为可以通过**binlog\_row\_image**，其有3个值，默认为**FULL**：

* FULL：记录列的所有修改，即使字段没有发生变更也会记录。
* MINIMAL： 只记录修改的列。
* NOBLOB：如果是text类型或clob字段，不记录这些日志。


```

k3u1fbpfcp/39bd14d0b59542b3a906e43f642aaf2a~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=730&h=222&s=70854&e=png&b=ffffff)

我们可以将其修改为MINIMAL，则可以只记录修改的列的值。

binlog_rows_query_log_events参数

除此之外，部分同学认为Statement模式下，直接记录SQL比较直观，事实上，在Row模式下，也可以记录。mysql提供了一个**binlog_rows_query_log_events**参数，默认为值为FALSE，如果为true的情况下，会通过**Rows Query Event**来记录SQL。

![image.png](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/138c0b625cde4088a6c61453fd8bede8~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=870&h=208&s=79199&e=png&b=fefefe)

可以在my.cnf中添加以下配置，来开启row模式下的原始sql记录(需要重启)：

```
binlog-rows-query-log_events=1
```

之后，再插入数据数据时，在binlog文件中，我们将看到**Rows Query Event**

GTID Event

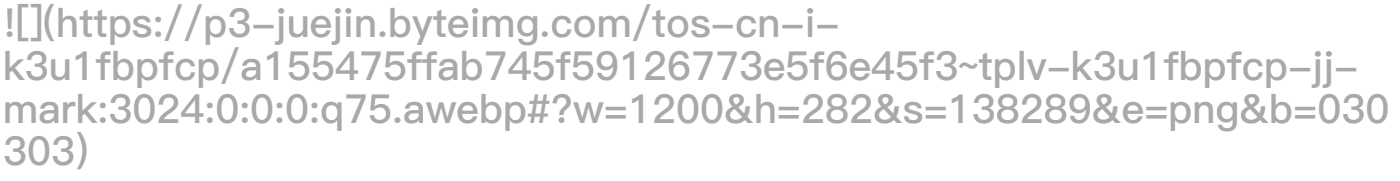
从MySQL 5.6开始支持GTID复制。要开启GTID，修改my.cnf文件，添加以下配置

```
gtid-mode=on

enforce-gtid-consistency=true
```

在这种情况下，每当我们执行一个事务之前，都会记录一个GTID Event

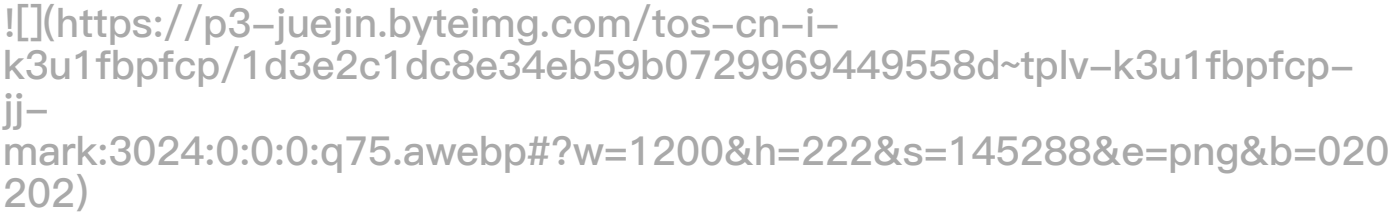
```
insert into user(”name”) values(”zhuyihan”);
```

此时binlog内容如下：



而当我们切换到下一个binlog文件时，会记录之前的已经执行过的GTID。这里我们通过执行以下sql手工创建一个新的binlog文件。

```
mysql> flush logs;

Query OK, 0 rows affected (0.00 sec)
```



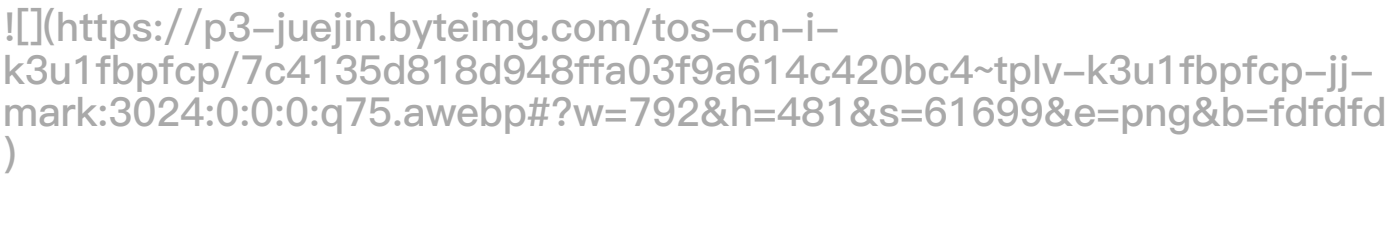
基于binlog的主从复制
==========

Mysql从5.0引入binary log(二进制日志)以支持主从复制。通过复制(Replication)允许将来自一个MySQL数据库服务器（master) 的数据复制到一个或多个其他MySQL数据库服务器（slave)，以实现灾难恢复、水平扩展、统计分析、远程数据分发等功能。

binary log中存储的内容称之为事件(event)，每一个数据库更新操作(Insert、Update、Delete，不包括Select)等都 **对应一个事件** **，这是本文讲述的重点。**

下面以mysql主从复制为例，讲解一个从库是如何从主库拉取binlog，并回放其中的event的完整流程。**注意：本文不是讲解mysql主从复制，而是讲解binlog中包含哪些类型的event，这些event的作用是什么，而讲解主从复制有利于我们理解binlog是如何工作。**

mysql主从复制的流程如下图所示：



主要分为3个步骤：

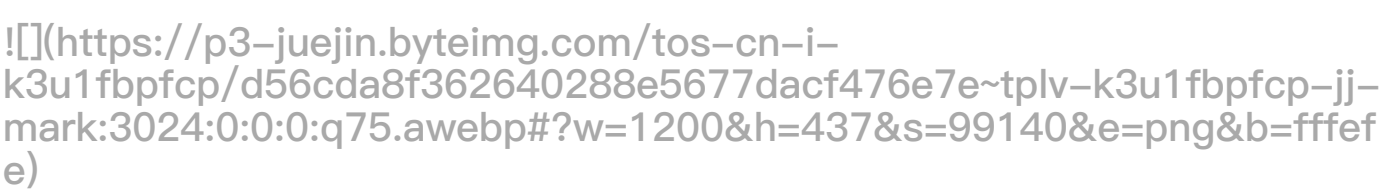
* **第一步：** master在每次准备提交事务完成数据更新前，将改变记录到二进制日志(binary log)中（这些记录叫做二进制日志事件，binary log event，简称event)
* **第二步：** slave启动一个I/O线程来读取主库上binary log中的事件，并记录到slave自己的中继日志(relay log)中。
* **第三步：** slave还会起动一个SQL线程，该线程从relay log中读取事件并在备库执行，从而实现备库数据的更新。

binlog应用场景
==========

读写分离

最典型的场景就是通过Mysql主从之间通过binlog复制来实现横向扩展，来实现

读写分离。如下图所示：



在这种场景下：

* 有一个主库Master，所有的更新操作都在master上进行
* 同时会有多个Slave，每个Slave都连接到Master上，获取binlog在本地回放，实现数据复制。
* 在应用层面，需要对执行的sql进行判断。所有的更新操作都通过Master(Insert、Update、Delete等)，而查询操作(Select等)都在Slave上进行。由于存在多个slave，所以我们可以在slave之间做负载均衡。

数据恢复

一些同学可能有误删除数据库记录的经历，或者因为误操作导致数据库存在大量脏数据的情况。例如我，曾经因为误操作污染了业务方几十万数据记录。

如何将脏数据恢复成原来的样子？如果恢复已经被删除的记录？

这些都可以通过反解binlog来完成，也是通过这个手段，来恢复业务方的记录。

最终一致性

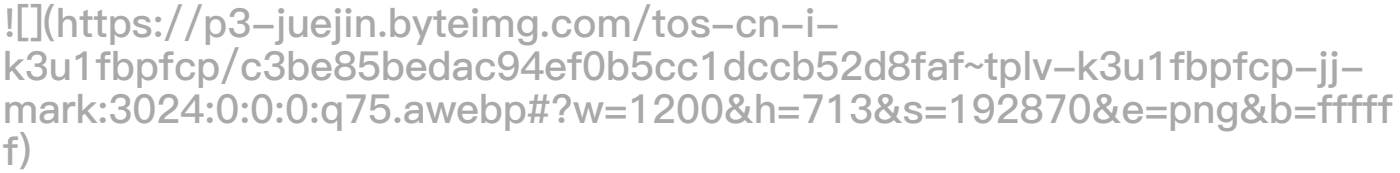
在实际开发中，我们经常会遇到一些需求，在数据库操作成功后，需要进行一些其他操作，如：发送一条消息到MQ中、更新缓存或者更新搜索引擎中的索引等。

如何保证数据库操作与这些行为的一致性，就成为一个难题。以数据库与redis缓存的一致性为例：操作数据库成功了，可能会更新redis失败；反之亦然。很难保证二者的完全一致。

遇到这种看似无解的问题，最好的办法是换一种思路去解决它： 不要同时去更新数据库和其他组件，只是简单的更新数据库即可。

如果数据库操作成功，必然会产生binlog。之后，我们通过一个组件，来模拟的mysql的slave，拉取并解析binlog中的信息。通过解析binlog的信息，去异步的更新缓存、索引或者发送MQ消息，**保证数据库与其他组件中数据的最终一致**。

在这里，我们将模拟slave的组件，统一称之为**binlog同步组件**。你并不需要自己编写这样的一个组件，已经有很多开源的实现，例如linkedin的databus，阿里巴巴的canal，美团点评的puma等。当我们通过binlog同步组件完成数据一致性时，此时架构可能如下图所示：



增量索引

通常索引分为全量索引和增量索引。对于增量索引的部分，可以通过监听binlog变化，根据binlog中包含的信息，转换成对应存储的语法，进行实时索引更新。

可靠消息

可靠消息是指的是：保证本地事务与发送消息到MQ行为的一致性。一些业务使用**本地事务表**或者**独立消息服务**，来保证二者的最终一致。Apache RocketMQ在4.3版本开源了**事务消息**，也是用于完成此功能。事实上，这两种方案，都有一定侵入性，对业务不透明。通过订阅binlog来发送可靠消息，则是一种解耦、无侵入的方案，因此有不少公司通过RMQ实现存储binlog的同步和在线到离线数据的增量拉取

缓存一致性

业务经常遇到的一个问题是，如何保证数据库中记录和缓存中数据的一致性。不妨换一种思路，只更新数据库，数据库更新成功后，通过拉取binlog来异步的更新缓存(通常是删除，让业务回源到数据库)。如果数据库更新失败，没有对

应binlog，那么也不会去更新缓存，从而实现最终一致性。

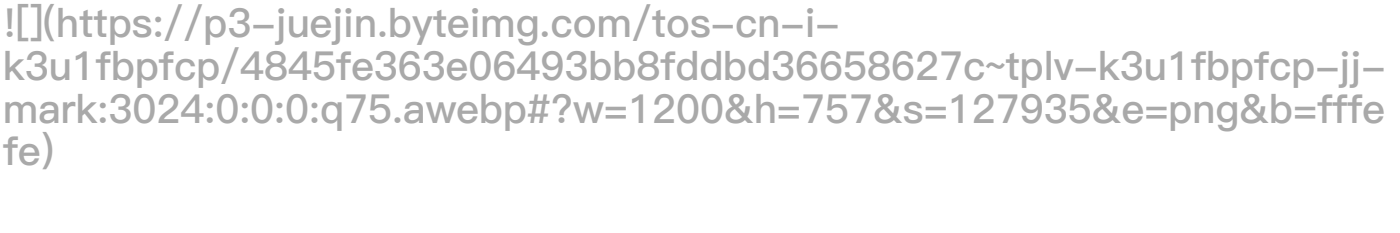
可以看到，binlog是一把利器，可以保证数据库与与其他任何组件(**es** **、** **mq** **、** **redis** **等)的最终一致。这是一种优雅的、通用的、无业务入侵的、彻底的解决方案** **。** **我们没有必要再单独的研究某一种其他组件如何与数据库保持最终一致，可以通过binlog来实现统一的解决方案** **。**

在实际开发中，你可以简单的像上图那样，每个应用场景都模拟一个slave，各自连接到Mysql上去拉取binlog，master会给每个连接上来的slave一份完整的binlog拷贝，业务拿到各自的binlog之后进行消费，彼此之间互不影响。但是这样，有一些弊端，多个slave会给master带来一些额外管理上的开销，网卡流量也将翻倍的增长。

我们可以做的更好，不同场景模拟多个slave来连接master获取同一份binlog，本质上要满足的是：一份binlog数据，同时提供给多个不同业务场景使用，彼此之间互不影响。

显然，消息中间件是一个很好的解决方案。现在很多主流的消息中间件，都支持consumer group的概念，如kafka、rocketmq等。**同一个topic中的数据，可以由多个不同consumer group来消费，且不同的consumer group之间是相互隔离的**，例如：当前消费到的位置(offset)。

因此，我们完全可以将binlog，统一都发送到MQ中，不同的应用场景使用不同的consumer group来消费，彼此之间互不影响。此时架构如下图所示：



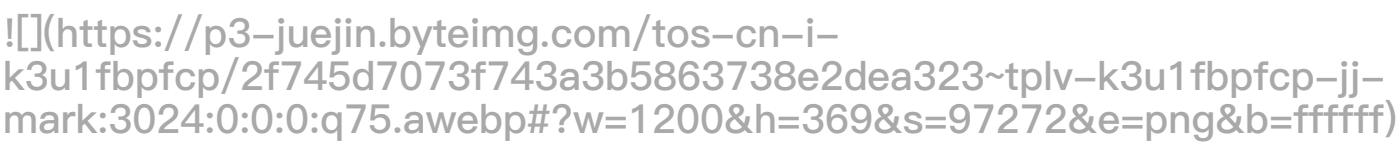
通过这样方式，我们巧妙的达到了一份数据多个应用场景来使用。通常情况下，一个Mysql实例中可能会创建多个库(Database)，通常我们会将一个库的binlog放到一个对应的MQ中的Topic中。

当将binlog发送到MQ中后，我们就可以利用MQ的一些高级特性了。例如binlog发送到MQ过快，消费方来不及消费，可以利用MQ的消息堆积能力进行流量削峰。还可以利用MQ的消息回溯功能，例如一个业务需要消费历史的binlog，此时MQ中如果还有保存，那么就可以直接进行回溯。

当然，有一些binlog同步组件可能实现了类似于MQ的功能，此时你就无序再单独的使用MQ。

异地多活

一个更大的应用场景，异地多活场景下，跨数据中心之间的数据同步。这种场景的下，多个数据中心都需要写入数据，并且往对方同步。以下是一个简化的示意图：



这里有一些特殊的问题需要处理。典型的包括：

* 数据冲突：双方同时插入了一个相同主键的值，那么往对方同步时，就会出现主键冲突的错误。
* 数据回环：一个库A中插入的数据，通过binlog同步到另外一个库B中，依然会产生binlog。此时库B的数据再次同步回库A，如此反复，就形成了一个死循环。

如何解决数据冲突、数据回环，就变成了binlog同步组件要解决的问题。同样，业界也有了成熟的实现，比较知名的有阿里开源的otter，以及摩拜(已经属于美团)的DRC等。

在线到离线同步

当在线DB并发足够高，此时离线需求小时/天级别去拉取DB的数据，就会出现性能问题，如果在拉取的时候度全表数据，可能会产生巨大事物，会极大影响在线业务，因此DB的在线到离线同步（增量+全量）都采用binlog同步的方式

以上图为例，用户对DB进行操作，DB产生binlog，然后由特定的节点（Worker）拉取数据写入RMQ（或者其他消息队列），再由dump组件从消息队列中解析消息并写入hdfs或者hive用以离线使用

原文链接: https://juejin.cn/post/7385457987908288564