binlogbroker

背景

1、希望找到一个更舒服的缓存更新方案

```
a、散落在业务代码；
b、加一层数据库代理
```

2、商品数据和订单数据同步非实时(mysql -> elasticsearch)

```
  a、商品同步:每天定时同步
  b、订单同步:业务代码 -> mom-woker -> 同步服务
```

Mysql Replication

主从复制基础: binlog

- binary log: 对所有涉及写的操作进行记录，每一条log都有对应的事件，以二进制的形式保存在磁盘中
- 用途：查看历史操作，数据备份和恢复，主从复制

binlog 格式

- 基于SQL语句的复制(statement-based replication, SBR): 记录的是sql语句
    - 优点:
      1、仅记录sql语句, binlog文件体积小,节约I/O, 复制过程就是再执行一次sql;
      2、主从表的定义只需要兼容但可以不同可(比如可以先改从的表结构再提升为主)
    - 缺点
      1、包含不确定操作时不能正确复制(调用不确定函数, 正在使用触发器或者存储过程);
      2、必须保持额外的信息才能保证在主从执行是有相同的结果;
      3、相比RBR而言，同样的耗时sql会再执行一次
- 基于行的复制(row-based replication, RBR): 记录的是每一条记录的变化情况(修改前和修改后)
    - 优点
      1、所有的操作都能复制，mysql推荐的复制方式
      2、只记录更改，不用再次执行高消耗sql，减少锁的使用, 在从执行写操作是性能更好
    - 缺点
      1、binlog文件体积变大，比如 update正张表，会记录所有变更
- 混合模式复制(mixed-based replication, MBR)：通常情况下使用SBR，有一些sql语句(uuid)无法使用sbr时，自动切换从RBR
    - 综合SBR和RBR和优缺点, 看着挺好，为啥不用?

RBR格式 控制binlog文件大小参数: binlog_row_image

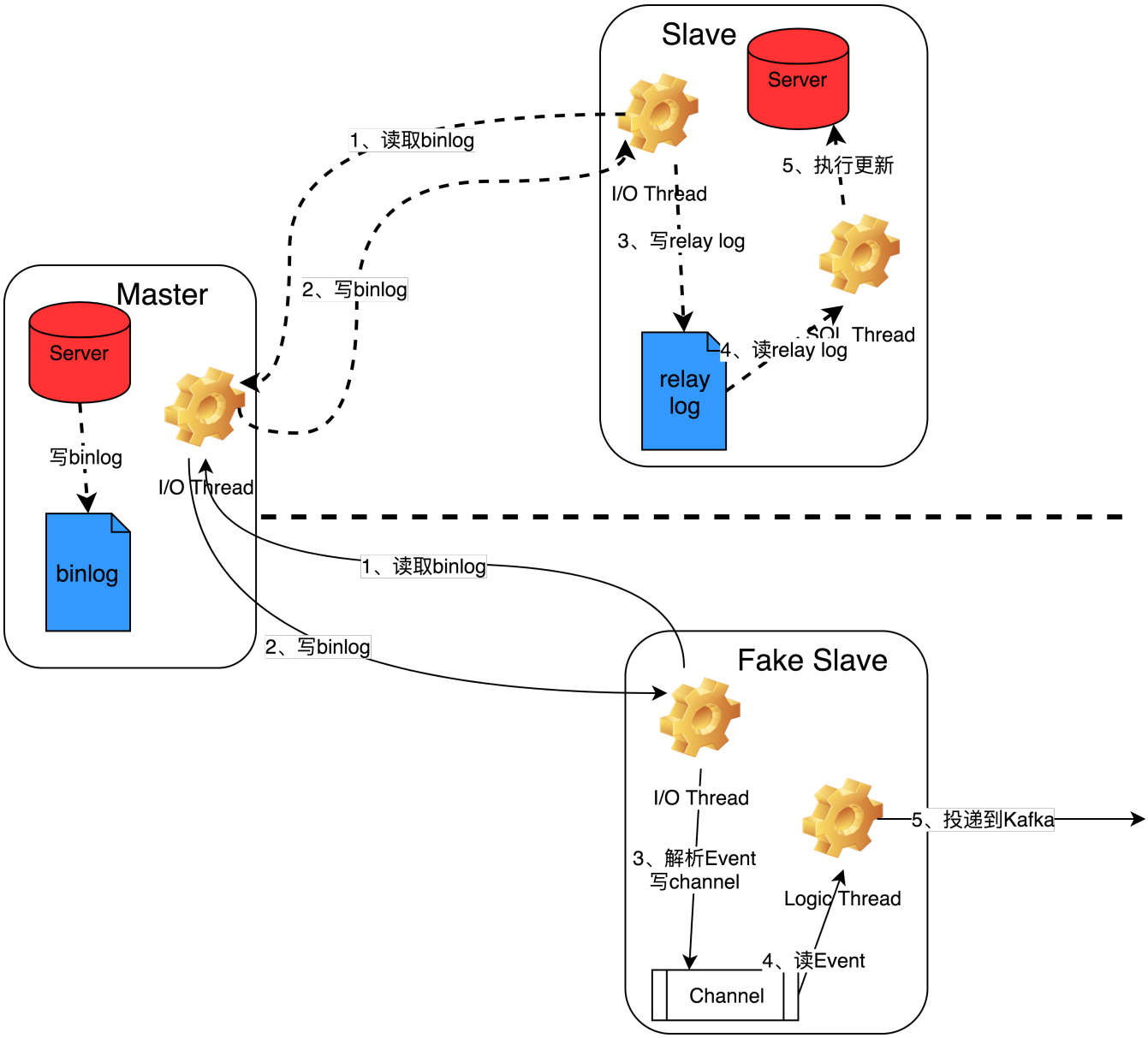
- FULL: 所有字段
- MINIMAL: 只记录了修改了的字段

- NOBLOB: 除去BLOB&TEXT的全纪录

> 我们线上是 Row, FULL方式, 这个是数据最全的，也是建议的方式

binlogbroker工作流程

模拟slave server，通过网络同步binlog



binlogbroker实现

> 我们线上都是基于传统复制, 所以不考虑GTID复制

mysql复制命令

```
change master to
master_host='192.168.15.178',master_user='backup',master_password='123456',master_log_file='
mysql-bin.000056',master_log_pos=151744220
```

- master_log_file: 指定开始同步的文件
- master_log_pos: 从文件pos位置开始
  如果我们要自己实现一个fake slave, 也需要这样这两个参数。

关注的binlog事件

mysql的事件很多, 只需要关注我们需要的时间。
通过复制命令，需要关注binlog文件发生变化的事件; 需要解析数据，所以需要关心DML相关操作的事件; 如果修改表结构，需要关注DML操作的时间

- FORMAT_DESCRIPTION_EVENT: binlog文件中第一个事件, 根据该事件的定义来解析其他事件
- TABLE_MAP_EVENT: 记录表的元数据信息(名称, 数据库，表接口等)
- ROWS_EVENT: 所有的DML操作都是记录在ROWS_EVENT中(将数据解析成完整的记录就是根据TABLE_MAP_EVENT的信息来解析)
- QUERY_EVENT: RBR格式下, 对应的DDL操作记录在QUERY_EVENT
- XID_EVENT: 在事务提交时，会在末尾添加一个XID_EVENT事件代表事务的结束
- ROTATE_EVENT: binlog文件发生切换的时候，会在当前日志添加改事件

Fake Slave 开始工作

发送COM_REGISTER_SLAVE -> 发送 COM_BINLOG_DUMP -> 获取Blog Event —> 根据mysql replicaiotn 协议解析log -> 投递到kafka

Kafka数据发布策略

取决于如何使用kafka

Kafka Topic策略

- 实例维度
- DB维度
- Table维度

默认topic是schema name

Kafka Partition策略

- 以Table名称分区
  - 优点：同一张表数据在同一partition, 基本有序
  - 缺点：
    a、按照表的维度，每张表的更新频率不一样，partition数据可能分布不均，扩展性不好
- 指定字段名分区
  - 优点：数据根据具体字段分区，分布更均匀，能保证有序
    默认是以table name做partion key

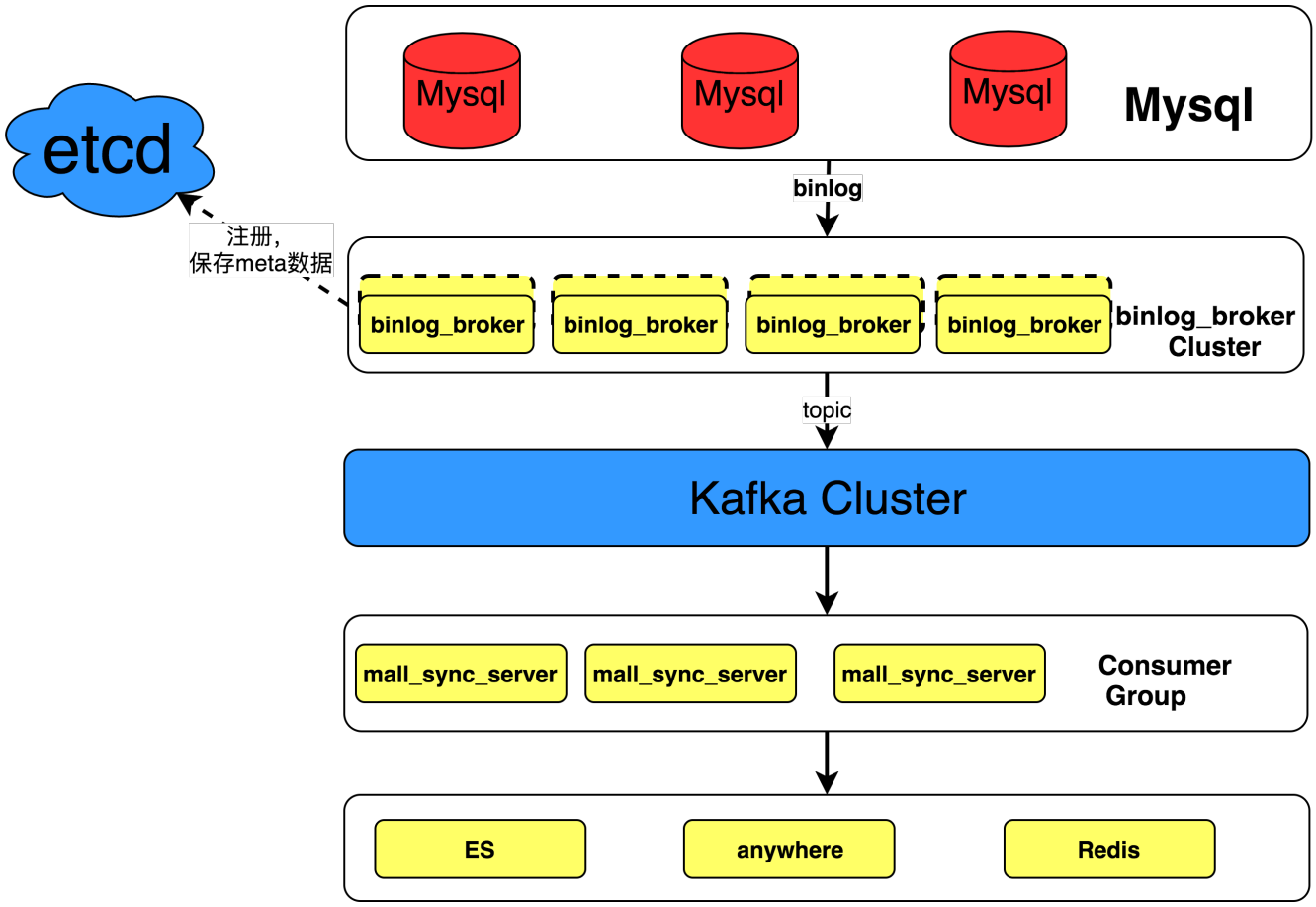
HA实现(强依赖etcd)

- 唯一的状态数据: binlog文件名、position保存在etcd
- 基于etcd实现选主逻辑
- 向etcd发送心跳，检查网络连接
- binlog同步异常，kafka消息投递异常，会主动弃主
- 一个mysql实例配置一组binlogbroker(>=1个)，同一时刻只会有一个broker工作

数据一致性

- 取决于kafka的使用方式
- broker确保消息不丢以及有序投递

broker在应用中的位置



基于binlogbroker能方便的实现哪些功能?

binlog解析完成后消息投递到kafka，相当于实现了一条数据总线, 通过订阅kafka消息进行各种处理:

- 数据同步
- 缓存更新
- 基于数据变动事件通知
- 实时统计

业务接入(kafka消费)

订阅kafka消息, 封装了一个简单的client方便处理数据:

```
//kafka消息格式
type EventData struct {
  //增删改
  Action  EventAction
  Schema  string
  Table   string
  //字段列表
  Columns []string
  //数据列表如果是update,  Rows[i]是更新前的数据, Rows[i+1]是更新后的数据
  Rows  [][]interface{}
  Owner TopicInfo `msg:"-"`
}
```

```
//以json格式处理
func (p *EventData) ToJsonArrary() (string, error) {
}
//以结构体方式处理
//container := make([]Goods, 0)
func (p *EventData) ToObjArrary(container interface{}) error {
}

//---------demo---------
//实现这样一个handle即可
func Handle(msg *protocol.EventData) error {

}

func run() {
    cc := &queue.ConsumerCfg{
        TopicList: []string{"mall_db", "social_db", "order"}, GrouponId: servername, Addr: addr,
    }
    queue.SetConsumer(&queue.SimpleConsumer{
        Cfg: cc, Work: Handle,
    })
    queue.StartConsume()
}
```