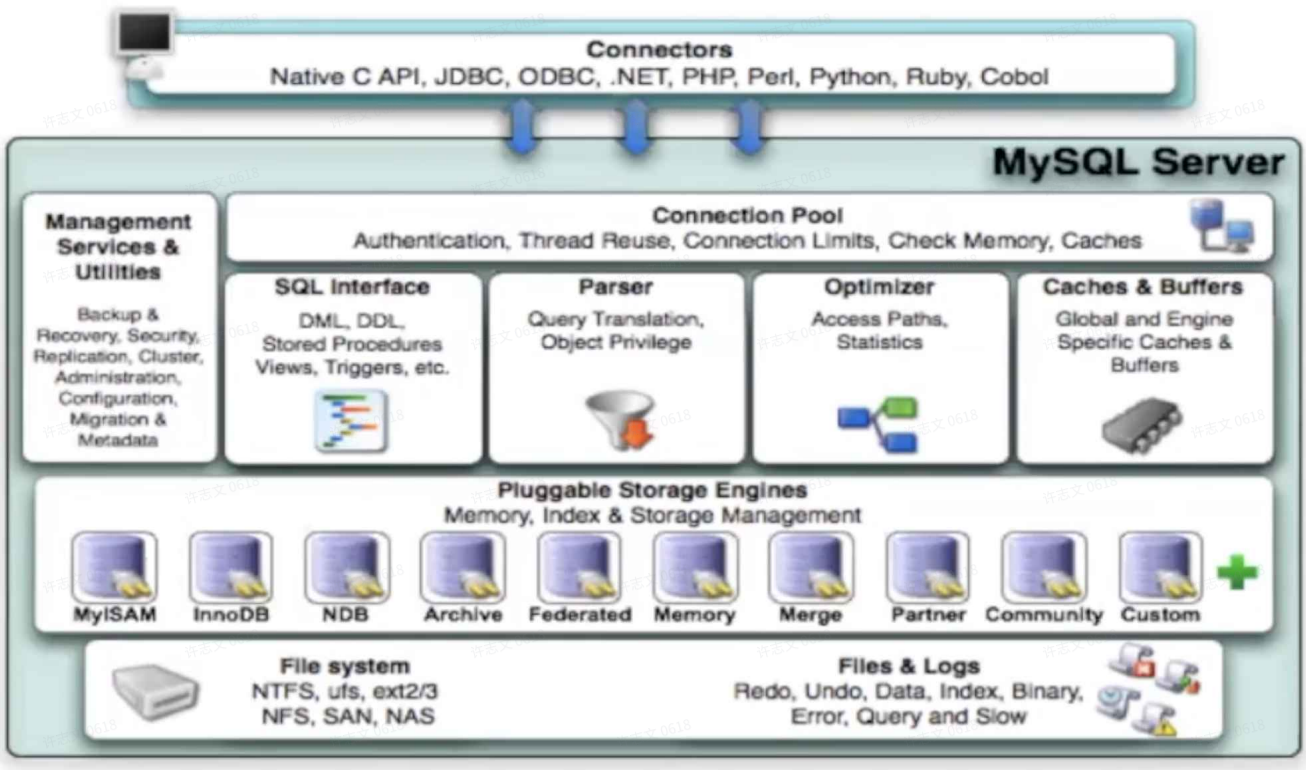
MySql介绍

目录

1 MySql的介绍及常用规范

1.1 MySQL的体系架构是什么样的



1.2 什么场景下应该使用MySQL

1. 事务支持
2. 高并发需求
3. 数据一致性要求比较高
4. 响应时延有一定要求
5. 稳定性要求
6. 数据结构化存储、查询

1.3 使用MySQL的时候我应该遵循哪些规范

基础规范

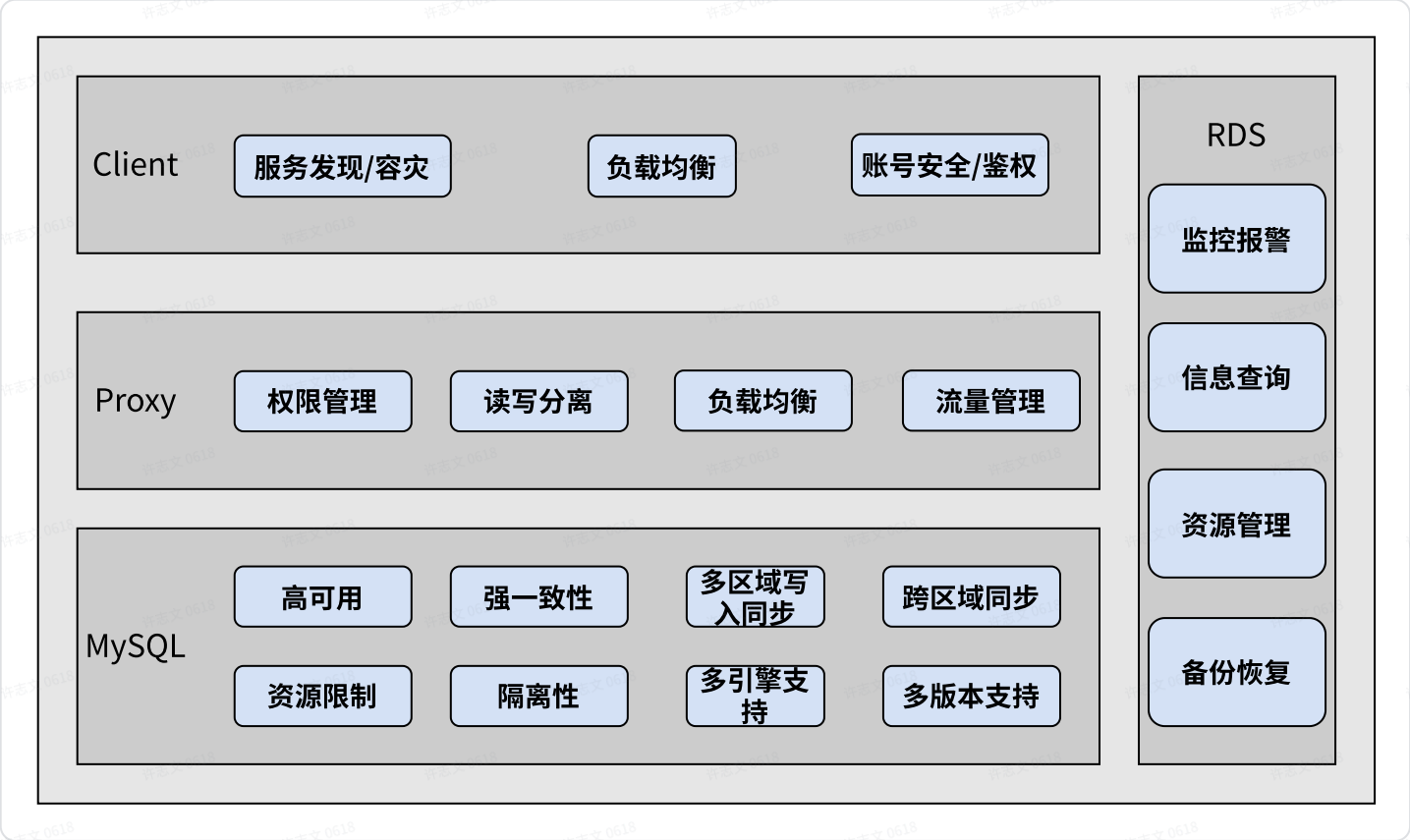
1. 一般场景下尽量使用InnoDB存储引擎。InnoDB是MySQL最成熟存储引擎，适用于大部分场景。
2. 新库字符集选择上请使用utf8mb4字符集。utf8存一个字符要用3个字节，utf8mb4一个字符用4个字节存储。现在数据库存储Emoji表情比较多，一个Emoji表情占4个字节，utf8mb4比起utf8多了emoji编码支持。
3. 线上数据库禁止使用存储过程、视图、触发器、Event。高并发互联网场景下需要解放DB的cpu，把所有计算上移，减少不必要的cpu消耗。
4. 线上数据表必须添加中文注释。方便维护。
5. 图片视频不存表，表命名等规范。

使用规范

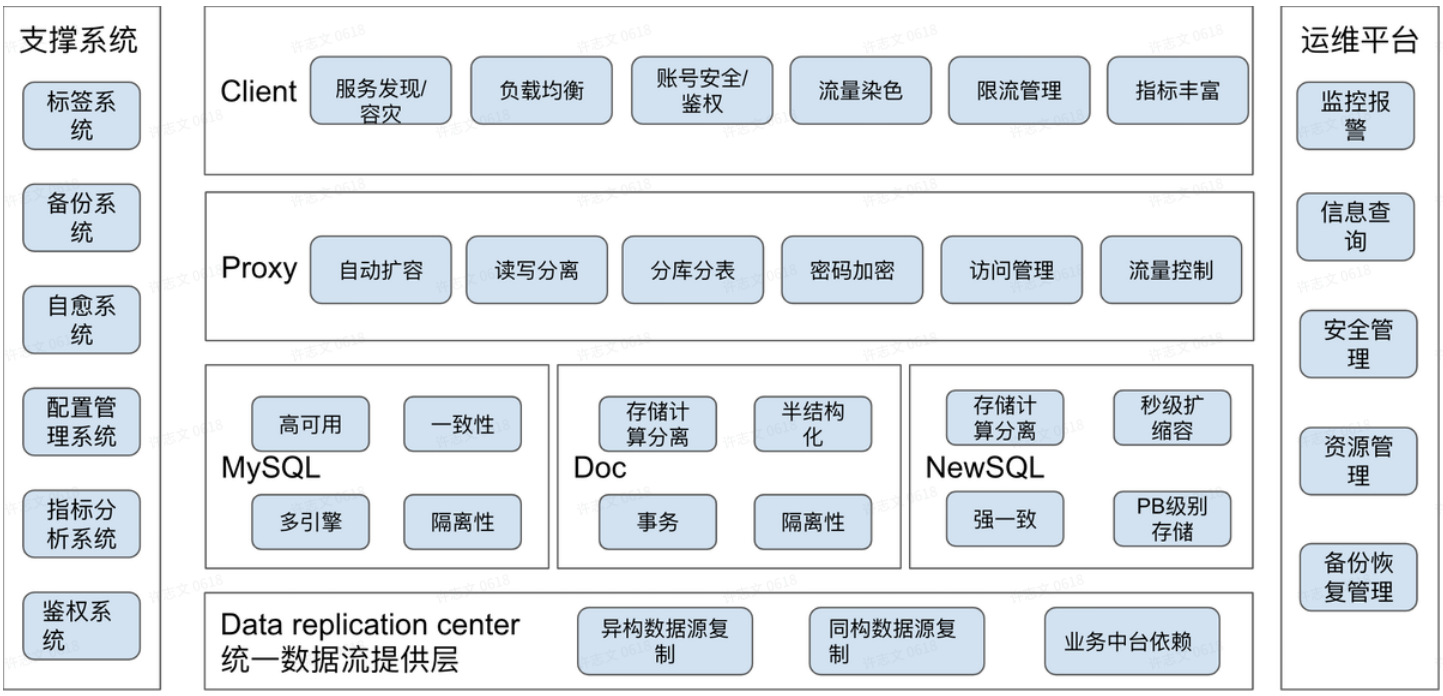
1. 不允许负向查询以及%开头的模糊查询。负向查询即 `!=` 或 `not in` 。负向查询和%开头的模糊查询会导致查询无法使用索引，全表扫描。
2. 不允许在WHERE条件的字段上直接使用函数或者表达式。例如 `where from_unixtime(date) > '2023-6-6'` ，该写法会导致全表扫描。正确写法应该是 `where date > unix_timestamp('2023-6-6')`
3. 不允许WHERE条件的字段使用隐式转换。例如phone字段是string类型，但是写了 `where phone = 123456` ，此时会发生隐式转换，导致全表扫描。
4. 最好不使用SELECT *，只获取必要的字段，需要显示说明列属性。SELECT * 会增加cpu、io、net消耗，增加回表次数（覆盖索引失效）。
5. 分片库使用场景下，要求在线查询必须带分片键。不带分片键会导致查询放大问题，会导致所有分片做无效查询。
6. 不允许超过3表join，不允许大表使用join、子查询等

2 字节跳动MySQL服务

2.1 字节跳动MySQL服务的架构体系是什么样的



后续架构：



Client

- 服务发现/容灾。 服务发现，即提供一个consul，consul会解析到proxy，consul会有多个ip，所以需要解决多个ip的问题。解决ip问题的同时也可以解决负载均衡的问题。容灾即，会有多个proxy服务，即使大部分proxy down了，只要还有一个proxy，也可以连上数据库。
- 负载均衡。如上。

- 账号安全/鉴权。连接数据库不需要账号密码

Proxy

- 权限管理。黑白名单，和上面Client的鉴权一起使用。
- 读写分离。
- 负载均衡。
- 流量管理。使用RDS的人员需要填写服务最大峰值，如果超过某个峰值，就拒绝引入流量。例如缓存雪崩场景，当cache down后，所有流量都会打到DB上，这样可能会直接打死DB，不能提供服务。当有流量管理介入后，比如希望流量峰值有1000qps，那么可以保证这1000qps是正常的，超过1000的直接拒绝。

MySQL

- 高可用。高可用架构。
- 强一致性。半同步支持。
- 多区域写入同步。使用drc组件，可以支持A和B对同一条数据做写入。
- 跨区域同步。场景比如国内和国外用同一业务，但是配置在国内做的，此时修改配置，希望在国外也能生效。
- 资源限制。
- 隔离性。
- 多引擎支持。
- 多版本控制。

2.2 字节跳动MySQL服务如何接入

数据库服务接入。目前罗盘产品上，数据库配置是写在TCC上的，通过读取TCC配置连接数据库。

```
1
2  func InitCaller() {
3     DbCompassInsight = NewDB("db")
4     TccClient = NewTcc("ecom.compass.strategy_xxxxx")
5     AddListener(context.Background())
6     SQLClient = InitSQLClient()
7     ......
8  }
9
10 // TCC配置读取
11 func NewTcc(serviceName string) *tccclient.ClientV2 {
12    config := tccclient.NewConfigV2()
13    config.Confspace = "default" // ConfSpace is optional, default value is "defa
14    var err error
```

```
    client, err := tccclient.NewClientV2(serviceName, config)
    if err != nil {
        logu.CtxFatal(context.Background(), errcodes.FatalError, "tccclient.NewCli
        return nil
    }
    return client
}

// NewDB 构造一个新的DB对象
func NewDB(configName string) *DB {
    confMap, err := readDBConfig(configName)
    if err != nil {
        panic(err)
    }

    var conf DBConf
    var ok bool
    if env.IsPPE() || os.Getenv("cron_env") == "ppe" {
        conf, ok = confMap["Ppe"]
        if env.Env() == "ppe_release_brand" {
            conf, ok = confMap["Product"]
        }
    } else if env.IsProduct() {
        conf, ok = confMap["Product"]
    } else {
        conf, ok = confMap["Boe"]
    }

    ......

    dbObject, err := initDB(conf)
    if err != nil {
        panic(fmt.Errorf("db init failed, err:%v", err))
    }
    return &DB{read: dbObject, write: dbObject}
}

// initDB 使用 bytedgorm 创建 数据库连接
func initDB(c DBConf) (*gorm.DB, error) {
    // 参考: https://bytedance.feishu.cn/wiki/wikcnStoK2G1dNlRYr6owcFHEod#ZOlpnf
    // 加上`parseTime`和`loc`来启用mysql的`timestamp`和golang的`time.Time`自动转换,
    dsnf := `%s:%s@sd(%s)/%s?collation=%s&parseTime=True&loc=Local&timeout=%s&rea
    instRead := c.Read
    instWrite := c.Write
    dsnWrite := fmt.Sprintf(dsnf, instWrite.Username, instWrite.Password, instWri
    dsnRead := fmt.Sprintf(dsnf, instRead.Username, instRead.Password, instRead.E
    psmStr := strings.ReplaceAll(instWrite.Endpoint, "_write", "") // PSM 地址
```

```
62     logu.CtxInfo(context.Background(), "rds psm %v dsn = %v %v", psmStr, dsnWrite
63
64     // 使用 bytegorm
65     dbObject, err := gorm.Open(
66        bytedgorm.MySQL(psmStr, c.DBName).With(func(conf *bytedgorm.DBConfig) {
67           // 使用自定义的 DSN
68           conf.DSN = dsnWrite    // 写节点
69           conf.ReadDSN = dsnRead // 读节点
70        }).WithReadReplicas(), // 读写分离
71        bytedgorm.WithDefaults(), // 默认参数
72        bytedgorm.Logger{
73           LogLevel: logLevel, // 日志级别
74           //IgnoreRecordNotFoundError: true,
75        },
76     )
77     // 检查错误
78     if err != nil {
79        return nil, err
80     }
81     query_log.AddCallback(dbObject)
82     return dbObject, err
83  }
```

2.3 字节跳动MySQL服务关键功能点的架构实现

核心功能

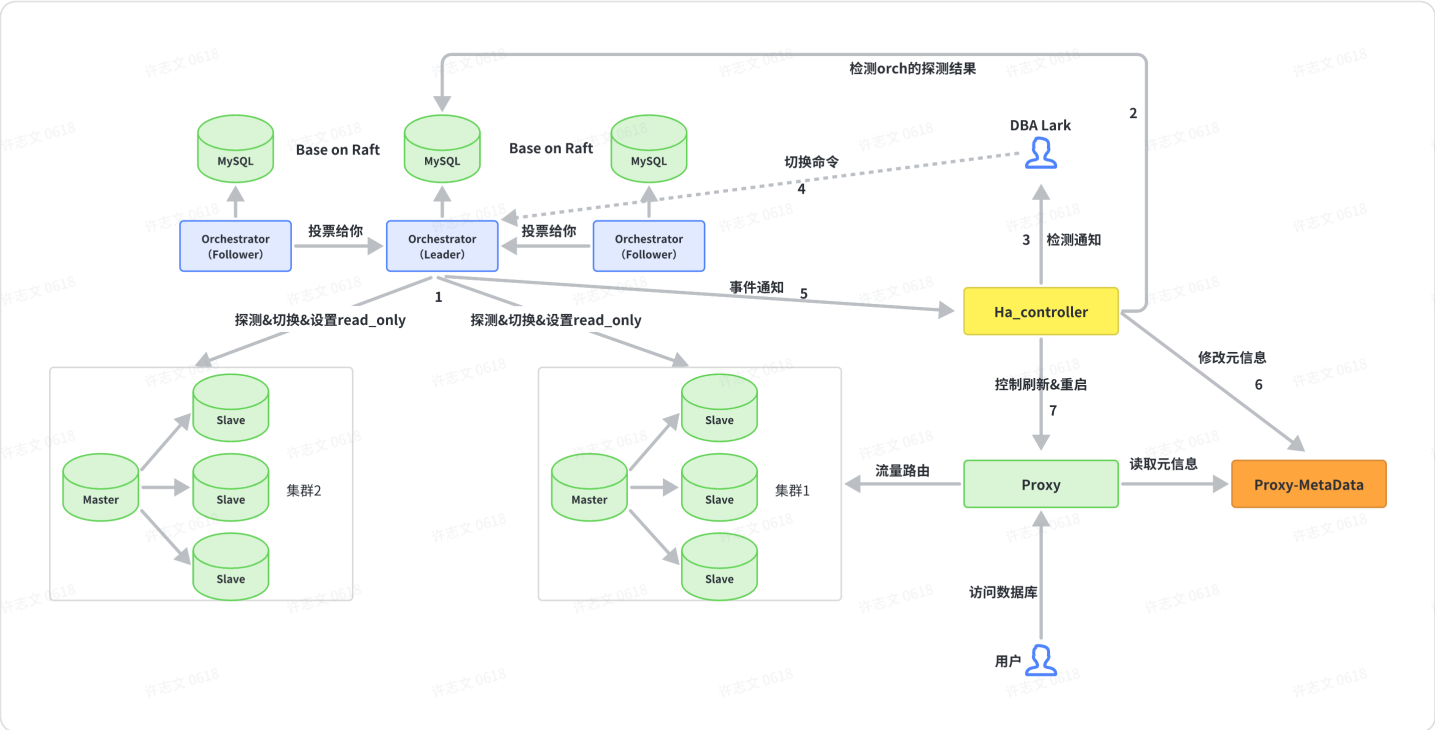
1. 收集mysql实例的实时状态
2. 当mysql实例异常时，发送异常监控报警
3. 实例的切换（异常切换、主动切换）
4. 多机房的容灾切换等功能
5. 业务场景定制化切换（半同步场景）

目前公司使用现状

- 实例5w+，集群5k+
- 日常宕机数量较多（主库10+，从库40+）

高可用实现架构

架构图



- **Orchestrator：**Orchestrator是一个开源的HA组件，负责mysql 实例信息的实时进行采集。每3个Orchestrator结点组成一个Orchestrator集群，每个集群的每个Orchestrator结点通过raft协议进行数据通信和同步，Orchestrator集群的高可用也是通过raft实现。因为Orchestrator是一个开源的组件，只包含一些通用功能，对于切换前后的业务逻辑处理，还需要自己实现，所以有了下面我们的HAControl组件。
- **HAControl：**HAControl负载整个HA的切换逻辑，它周期性的检查Orchestrator的实例采集结果，如果发现实例超时没有采集则进行相应的报警（有lark提醒及电话提醒两种），DBA接到报警后如果认为需要切换（有自动切换功能，可配置），则发送对应的切换命令给HAControl进行切换前后的业务逻辑处理。
- **Proxy：**主要负责流量重定向。

高可用流程

1. Orchestrator会探活集群，一旦发现集群或者集群的节点有问题了，会将状态记录到元信息库中。
2. HAControl会通过元信息库获取到orch的探测的结果。
3. 如果探测结果出问题，就会发消息给DBA。
4. DBA收到消息，就会去手动切换。在切换前，首先要做的就是强制将出问题的节点（还未break的）设置为read_only状态，去保证数据一致性；然后开始做选Master的操作，优先以一个机房（机房有lf，hl等）的节点做切换，当然也可以手动切换其他机房；在切换后，即选出新Master后，然后将消息通知给HAControl。

5. HAControl收到消息后，就会去修改元信息，修改元信息后，就会强制的让Proxy去取Proxy-MetaData(新Master信息)。
6. 最后重启Proxy，此时流量就可以正常访问。

总体切换流程大概1分钟左右，一分钟内，如果主库down了就不可写只可读。1分钟的原因在于，Proxy很多，在元信息做更新重启以及一些前置任务等过程大概需要1分钟左右。

使用GTID

这里主从复制字节采用的是GTID的方式去做的，没有采用传统的偏移量方法去做。

GTID(Global Transaction ID)是对于一个已提交事务的编号，并且是一个全局唯一的编号。GTID实际上是由UUID+TID组成的。其中UUID是一个MySQL实例的唯一标识，保存在mysql数据目录下的auto.cnf文件里。TID代表了该实例上已经提交的事务数量，并且随着事务提交单调递增。

传统主从复制（binlog + position文件偏移量）中，尤其是半同步复制中， 由于Master 的dump进程一边要发送binlog给Slave，一边要等待Slave的ACK消息，这个过程是串行的，即前一个事物的ACK没有收到消息，那么后一个事物只能排队候着，这样将会极大地影响性能；此外在主从故障切换中，如果一台MASTER down，需要提取拥有最新日志的Slave做MASTER，这需要Slave一直保存binlog和position。

MySQL从5.6开始增加GTID这个特性，用来强化数据库的主从一致性，故障恢复以及容错能力。相比于传统的主从复制的方式：

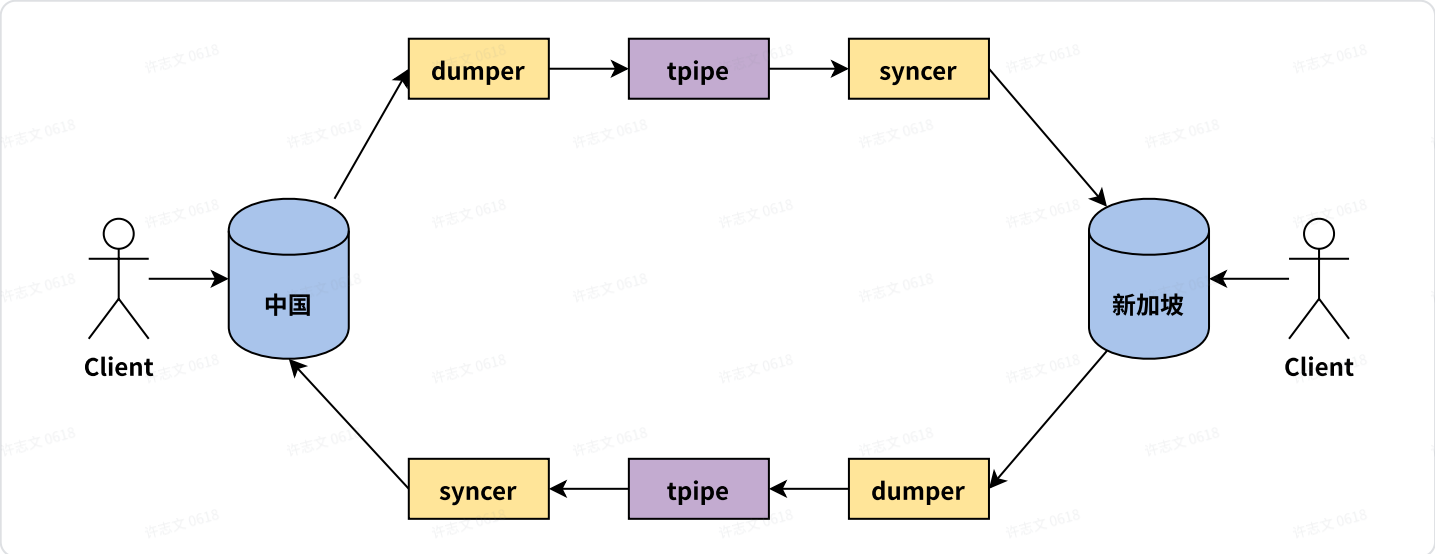
1. 主从复制过程，SLAVE直接可以通过数据流获得GTID信息。
2. 主从切换过程，Slave可以直接通过binlog获得GTID信息，而后执行的事务采用该GTID。

GTID流程如下：

- master节点在更新数据的时候，会在事务前产生GTID信息，一同记录到binlog日志中。
- slave节点的io线程将binlog写入到本地relay log中。
- 然后SQL线程从relay log中读取GTID，设置gtid_next的值为该gtid，然后对比slave端的binlog是否有记录。
- 如果有记录的话，说明该GTID的事务已经运行，slave会忽略。
- 如果没有记录的话，slave就会执行该GTID对应的事务，并记录到binlog中。

多区域写入架构

架构图



该架构主要解决弱网环境下数据同步问题。重点在于引入的drc组件。

drc主要分为dumper和syncer两个组件。同步原理也是基于binlog。

- dumper：负责从环境里解析出binlog，然后通过tpipe（类似queue，解决弱网环境问题）将数据传出去。
- syncer：负责从tpipe里取出数据消费，消费后将数据同步到另外一个环境。

两个问题

1. 数据回环问题，即数据从中国经过drc传到新加坡，再从新加坡drc到中国这个过程。

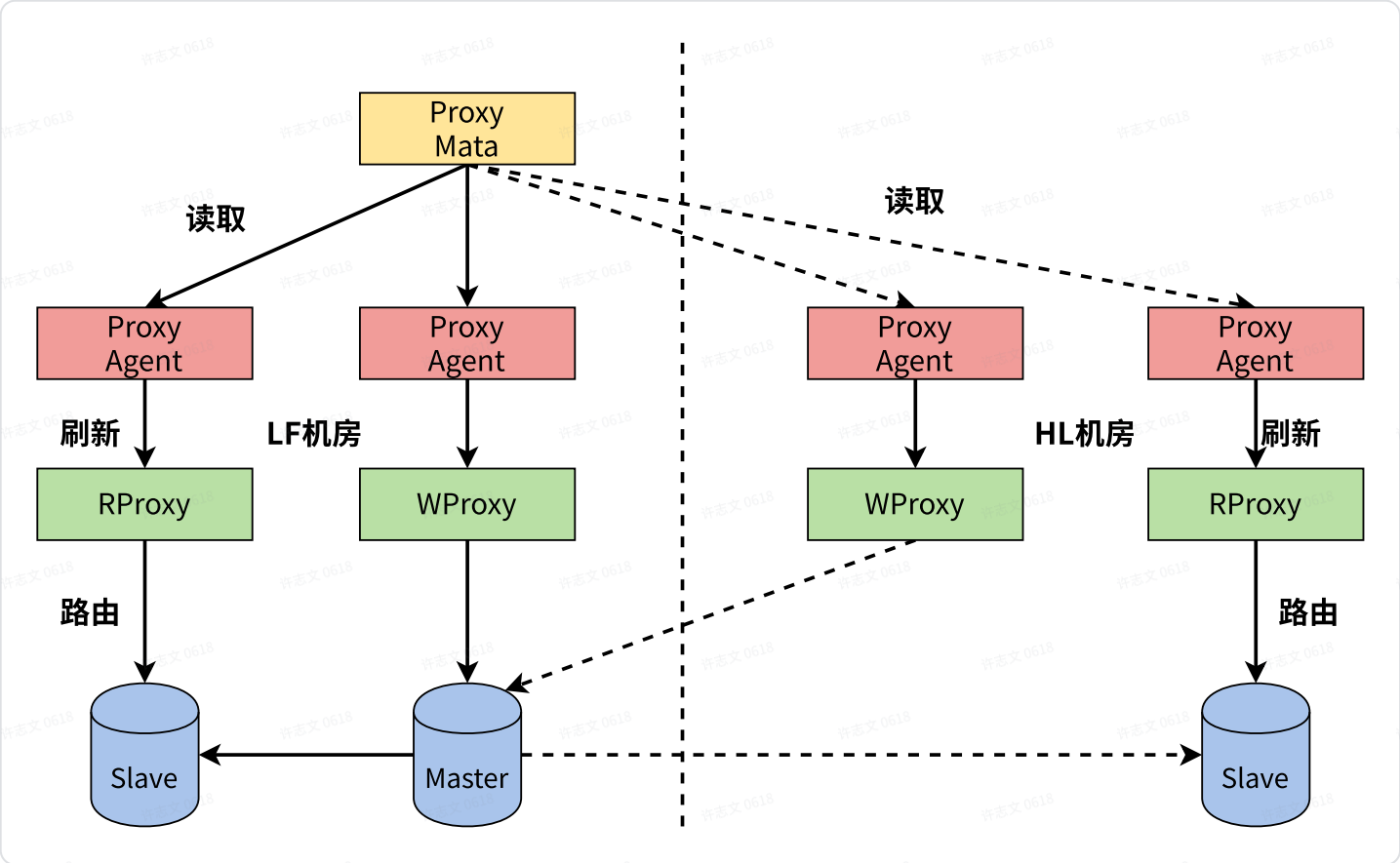
解决方式：会在数据中加入类似标签的东西，来标注数据来源，比如发现数据是从中国来的，那么再次drc时，就不会再传到中国。

2. 数据一致性问题，即假设数据同时从中国和新加坡开始写，数据冲突问题。

解决方式：依赖UTC时间戳（记录在binlog中作为动作发生时间）来判断谁先写。当然，也做了策略定制，比如就想以新加坡为先写，那么可以设置。

同城双机房

架构图



由于保证写强一致性问题，这里并没有使用drc的方式来同步两个机房的主写问题，而是使用同一个主库，其他机房写入也会直接路由到一个主库中。

Proxy-Agent会从Proxy-Mata中获取流量路由地址，这个地址通过instance属性标识具体哪个机房，然后路由到具体的机房读库下，这样，读请求就不会存在跨机房延迟问题。