

# 消息最终一致性的架构革命

分享人：叶东富



# 内容大纲

- 1 新架构二阶段消息
- 2 vs 其他方案
- 3 应用
- 4 总结

01

## 新架构二阶段消息




# DTM 简介

- 二阶段消息是DTM中的一种事务模式
- DTM 是一个跨语言分布式事务框架，主repo采用Go开发
- 6月开源至今，4.7K star，已经被多家大厂使用
- 全新的思路，全新的架构，很多的新理论与实践



## 二阶段消息定位

- 完美替代事务消息或本地消息表
- 消息最终一致性方案的革命性架构
- 架构复杂度
- 性能
- 接入难度



## 问题场景--跨行转账

- 跨行转账作为示例
- 跨数据库，跨服务无法用本地事务。
- 转入转出的操作中间可能宕机，如何保证同时成功或失败
- 场景假设：假如只有转出可能失败，转入不会失败



## 跨行转账--方案

- 没有回滚，适合消息最终一致性方案
- 其他备选方案：
  - 本地消息表，或者叫发件箱模式
  - 事务消息，RocketMQ的特殊消息
- 新架构方案：二阶段消息

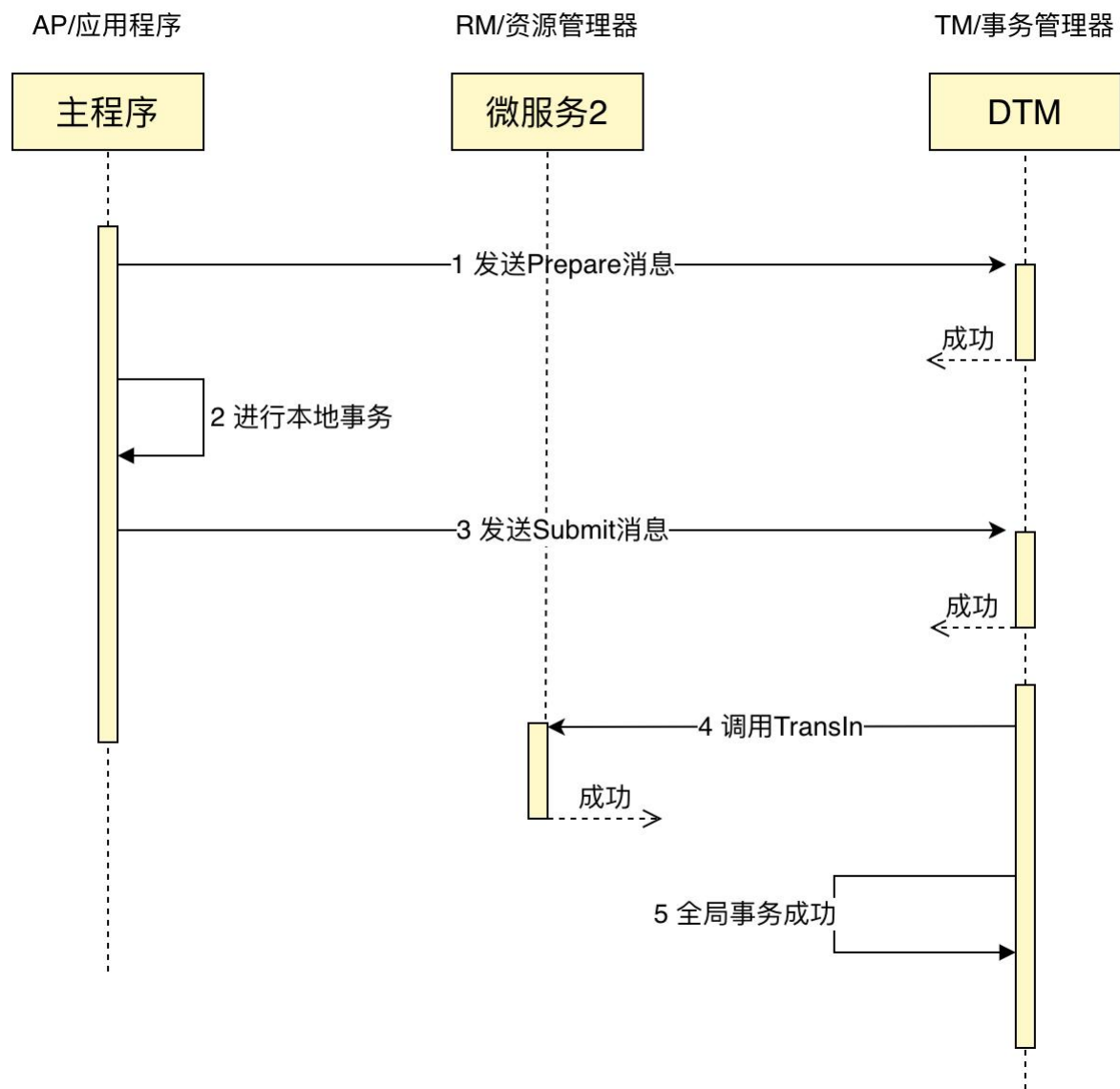


## DTM二阶段消息方案

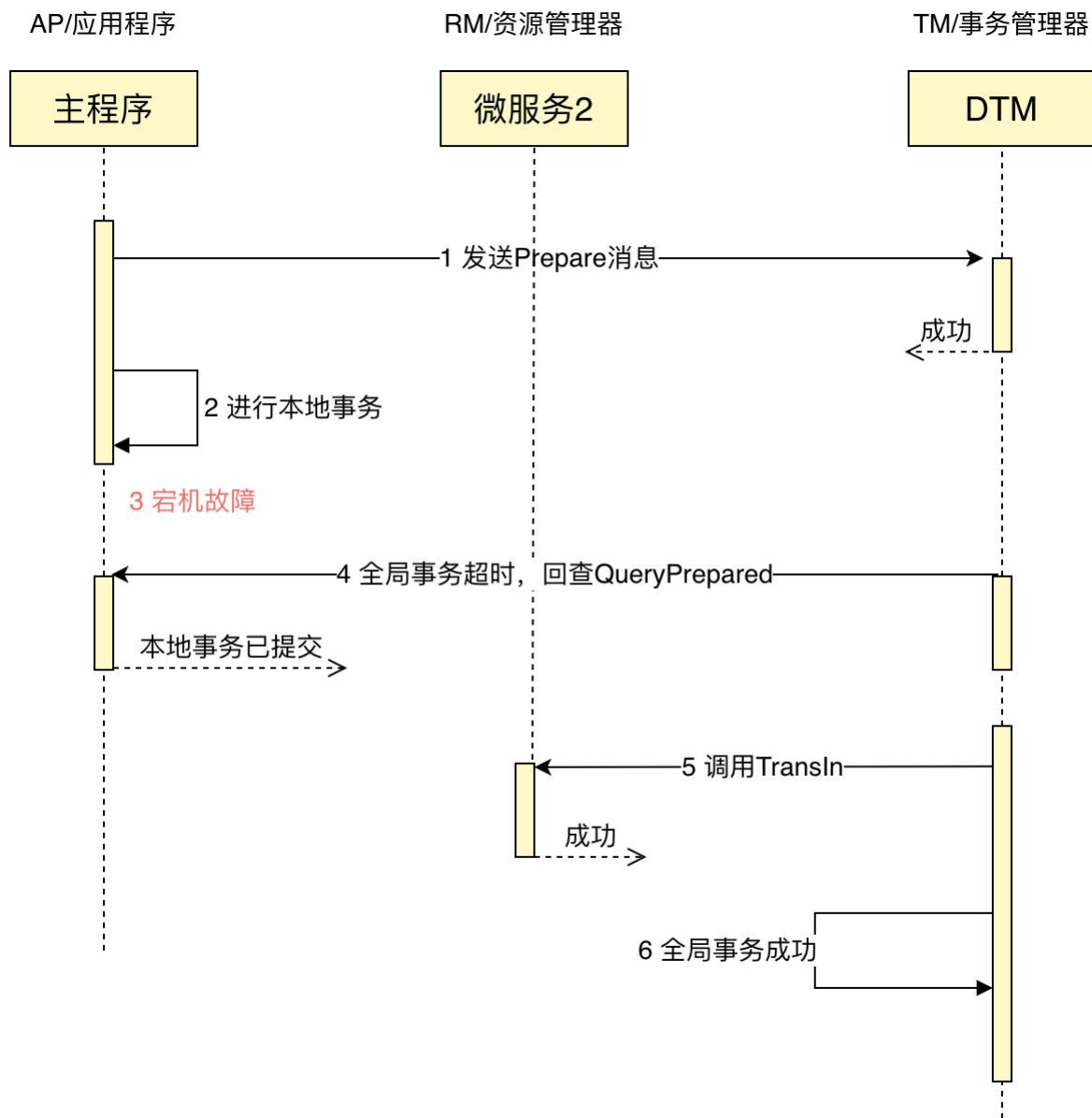
```
msg := dtmcli.NewMsg(DtmServer, gid).
    Add(busi.Busi+"/TransIn", &TransReq{Amount: 30})
err := msg.DoAndSubmitDB(busi.Busi+"/QueryPreparedB", db, func(tx *sql.Tx) error {
    return busi.SagaAdjustBalance(tx, busi.TransOutUID, -req.Amount, "SUCCESS")
})
```



# 成功时序图



# 提交后宕机



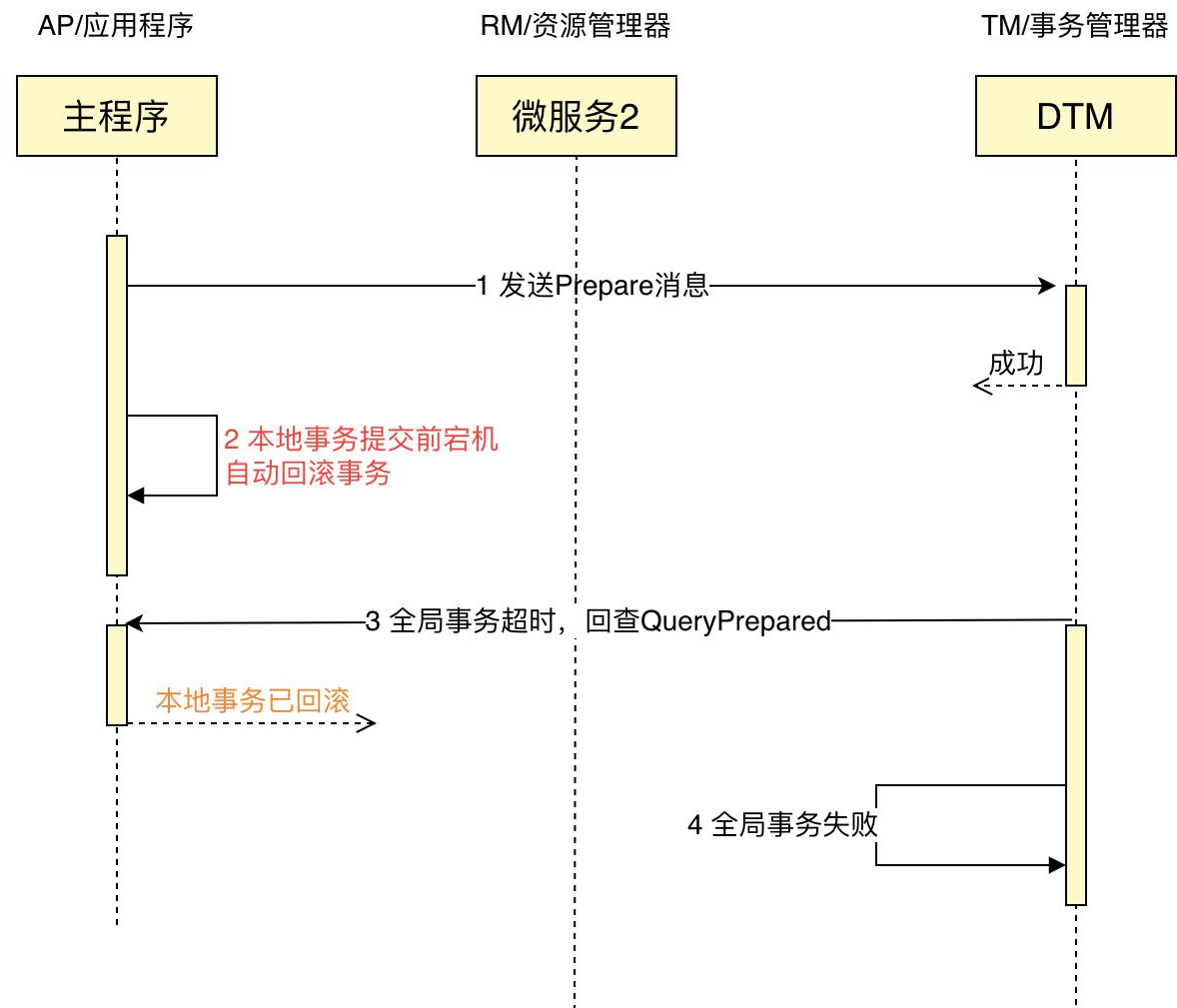


## 回查逻辑

- 回查由框架处理，只需要粘贴图中代码就行
- 自动回查属于首创，申请了专利 :)

```
app.GET(BusiAPI+"/QueryPreparedB", dtmutil.WrapHandler2(func(c *gin.Context) interfa
    return MustBarrierFromGin(c).QueryPrepared(dbGet())
}))
```

# 提交前宕机



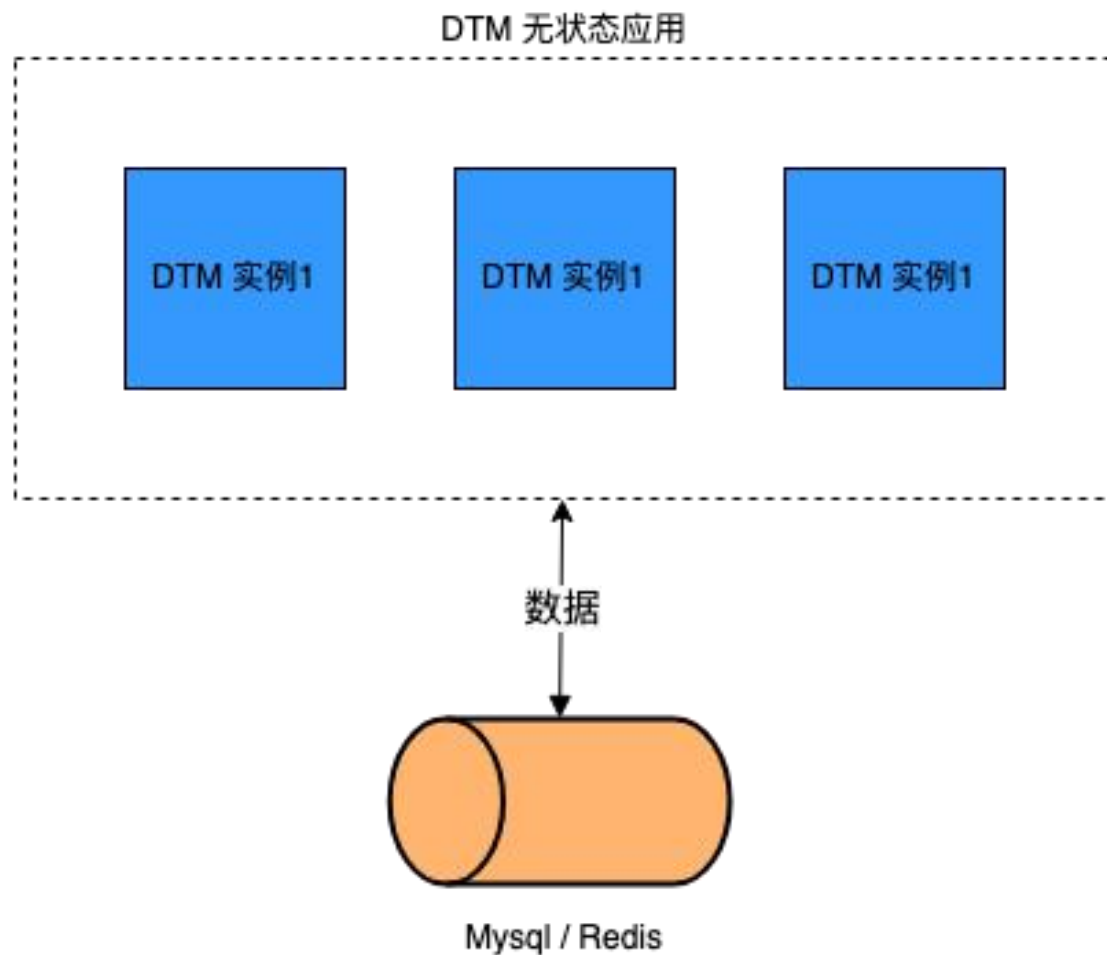
## DTM二阶段消息方案

```
msg := dtmcli.NewMsg(DtmServer, gid).
    Add(busi.Busi+"/TransIn", &TransReq{Amount: 30})
err := msg.DoAndSubmitDB(busi.Busi+"/QueryPreparedB", db, func(tx *sql.Tx) error {
    return busi.SagaAdjustBalance(tx, busi.TransOutUID, -req.Amount, "SUCCESS")
})
```

- QueryPreparedB 是回查url
- db是数据库连接

# 架构简单易用

- 简单优雅!!!
- 仅依赖dtm分布式事务管理器





## 支持情况

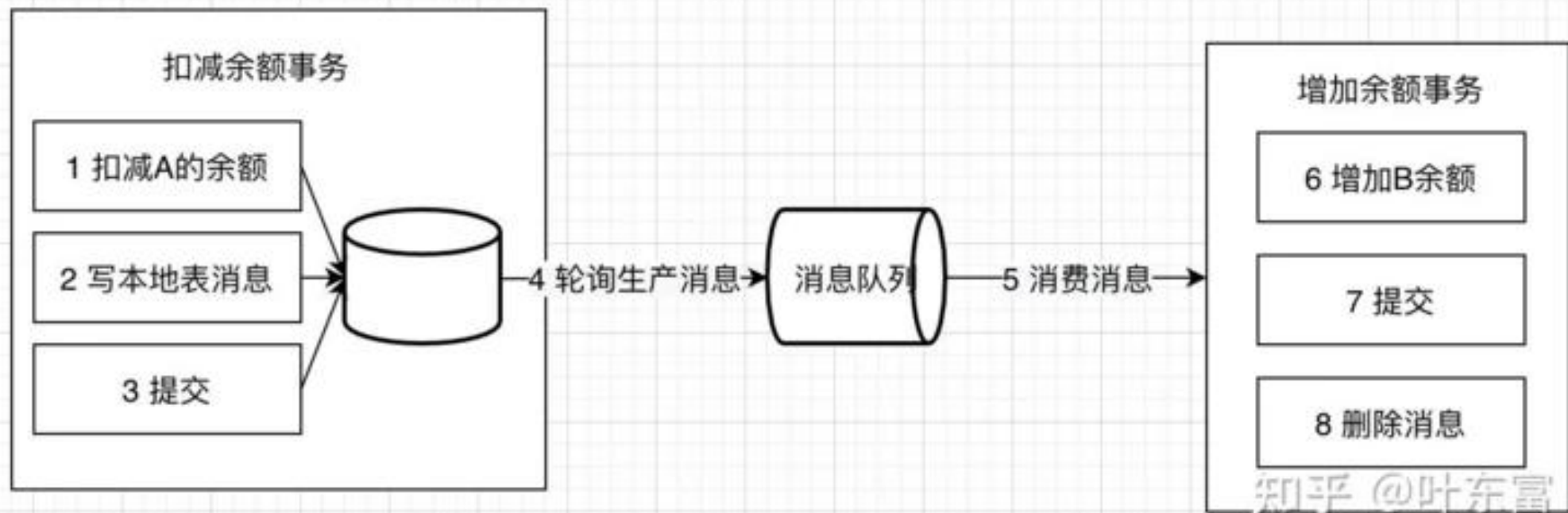
- 不仅支持数据库，还支持Redis，Mongo
- 已支持Go，C#
- PHP SDK已开发完毕，下周发布
- PHP SDK作者是PHP大佬，Hyperf创始人 黄朝晖

02

vs 其他方案



# 本地消息表

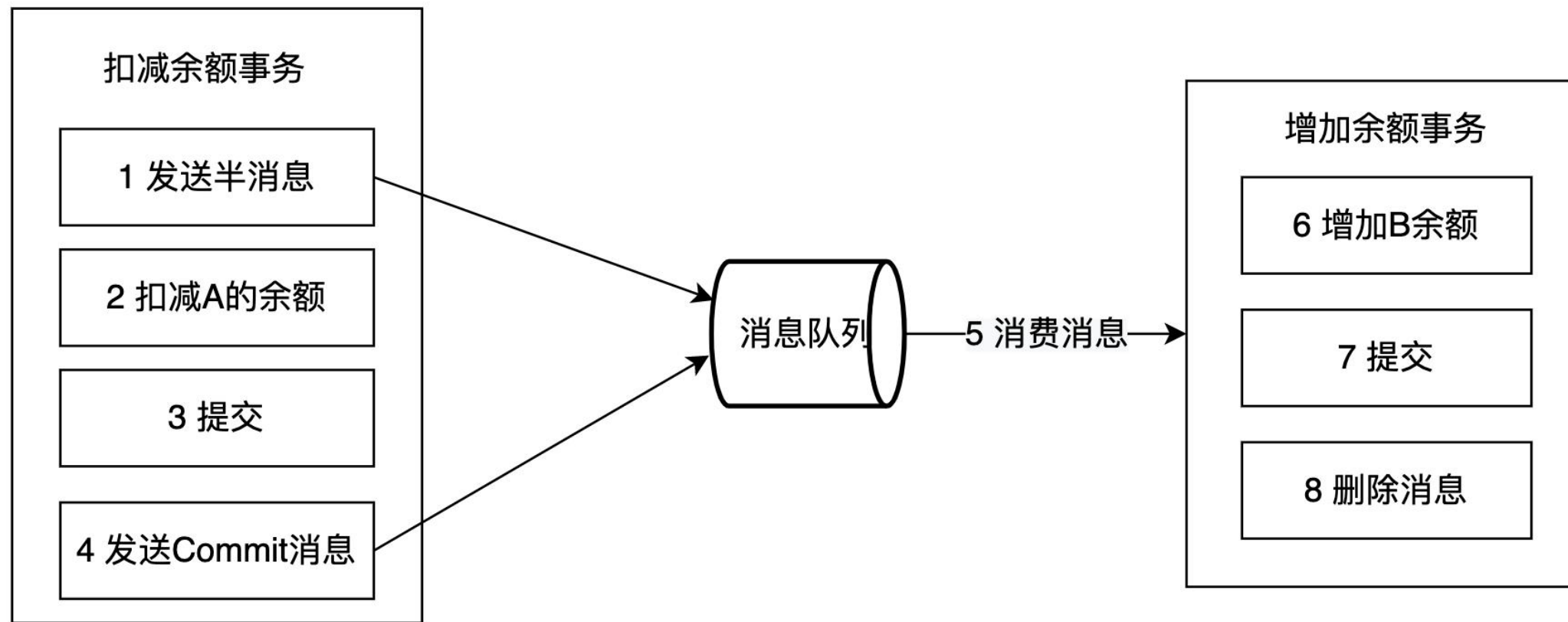




## 难点

- 轮询生产消息难实现：
  - 轮询逻辑：延时大，难以高效
  - binlog方式的Canal：重，难维护，只适用于Mysql
- 难维护：每个数据库实例，都需要维护生产消息的任务

# 事务消息--RocketMQ





## 难点

- 如果没有收到Commit消息，则会进行回查
- 回查方案为手动，并且有大问题：
  - 如果查到本地事务未提交，则每10s不断回查，直到2分钟超时
  - 未提交的情况导致大量无用回查
  - 极端情况还会出现数据错误



## 二阶段消息的回查

- 回查由框架处理，健壮性非常好
- 回查技术基于DTM框架，以及dtm首创的子事务屏障
- 高效、及时获得结果，极端情况也无数据错误



## 回查步骤

- 在处理本地事务时，会将(gid, 'committed')插入到dtm\_barrier.barrier表中。该表有一个唯一索引，字段为gid。
- 当进行回查时，再insert ignore一条(gid, 'rollbacked')的数据。
- 然后再查gid的记录，如果记录为committed，那么说明本地事务已提交；如果为rollbacked，那么说明本地事务已回滚或未来会回滚。



## 二阶段消息的对比优势

- 不依赖消息队列，全部是API式的接口，更符合开发者习惯
- 架构简单，逻辑模块化更好，不需要把逻辑放到队列消费者
- 二阶段消息是面向开发者设计，而队列方案则是开发者迁就架构
- 完美替代本地消息表和事务消息
- 推荐给熟悉的朋友，十个人中有八个，直接说会用到近期的项目中

03

## 二阶段消息应用





## 秒杀的问题

- 为了扛高并发，在Redis中扣库存
- 存在数据不一致问题，需要手动修复数据



## 在秒杀中的应用

- 核心代码如下：扣库存成功后，再创建订单
- <https://dtm.pub/app/flash.html>

```
gid := "{a}flash-sale-" + activityID + "-" + uid
msg := dtmcli.NewMsg(DtmServer, gid).
    Add(busi.Busi+"/createOrder", gin.H{activity_id: activityID, UID: uid})
err := msg.DoAndSubmit(busi.Busi+"/QueryPreparedRedis", func(bb *BranchBarrier) er
    return bb.RedisCheckAdjustAmount(rds, "{a}stock-"+stockID, -1, 86400)
})
```



## 秒杀--在Redis中精准扣库存

- 单Redis单机可以支持1.2w op/s，该架构可以轻松抗住几乎所有的秒杀场景，20年双十一峰值订单量为58.3万笔/秒
- 会精准创建等同于库存数量的订单，毫厘不差
- 组合了Redis的操作和数据库操作，保证两者的一致



## 在缓存管理中的应用

- 保证DB更新与缓存一致
- 大大优于Canal与队列方案
- 详情见: <https://dtm.pub/app/cache.html>



## 适用场景

- 除去第一个分支可能失败，其他分支都不会“最终失败”的场景
  - 异步调用
  - 消息解耦（计划中）
  - 延迟消息（计划中）

04

小结



## 二阶段消息

- 二阶段消息是DTM里的一种事务模式
- 该技术基于子事务屏障演变而来
- 完美解决不需要回滚的场景



# DTM 新一代分布式事务框架

- 分布式事务现状是，大家认为很难，能不用就不用
- 手动重试，补偿，为什么？？？
- 大部分一致性的本质需求就是重试和补偿。DTM 把它们做到了极致！
- DTM 提供了傻瓜式的接入方式，非常简单易懂，大幅降低分布式事务的使用门槛，初级工程师也能掌握





# DTM 使用情况

- 腾讯多个事业部在使用，很多PR来自腾讯的同学，已支持Polaris
  - 字节在用
  - 还有很多其他小公司在用
- 
- 稳定性很高，代码测试覆盖率95+%，用户报过来的bug，基本是环境配置相关，很少分布式事务本身的bug



# DTM 与 go-zero的深度合作

- go-zero 是go领域非常热门的微服务框架，社区非常活跃
- 与go-zero深度合作，可以在go-zero中原生的使用dtm
- 非单体的订单系统可以通过dtm大幅简化架构， go-zero社区有非常详尽的文章讲解，很容易上手
- 大家在这方面遇见任何问题，都可以快速获得社区的支持，快速得到解决

# DTM 功能

 订单			
 优惠券服务	 支付服务	 账户服务	 库存服务
 mongoDB	 MySQL	 PostgreSQL	 redis



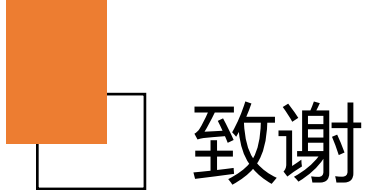
# DTM 事务模式

- 二阶段消息：不需要回滚的事务
- SAGA：需要回滚的事务
- TCC：对一致性要求较高的事务
- XA：并发较低，不会争抢行锁的事务



# DTM 适用场景

- 非单体的订单系统，基本都需要
- 微服务架构，基本离不开
- DTM的适用场景非常广非常广，需要大家改变旧认知
- 尝试一下DTM，就会发现它能帮你大幅简化与一致性相关的工作



致谢

## 新一代分布式事务架构

<https://github.com/dtm-labs/dtm>

欢迎大家使用我们的项目，并Star支持我们

好友进社区

