

Architect

SACC

2022 中国系统架构师大会

SYSTEM ARCHITECT CONFERENCE CHINA 2022

· 激发架构性能 点亮业务活力

云上会议 网络直播 | 2022年10月27-29日

IT168.com

ChinaUnix

ITPUB

# 微服务数据一致性常见问题及解决方案

常青藤爸爸 前CTO 叶东富

# 内容大纲

- 一致性问题
- 缓存一致性
- 消息一致性
- 异构存储的一致性
- 小结

# 一致性问题

# 一致性问题概述

- 一致性含义很多
- 在这里将它划分为两大类方案
  - 系统层：由系统软件提供一致性，开发者只关心原理
  - 应用层：由应用提供一致性方案，开发者关心较多



# 系统层一致性方案

- CAP中的C：多副本一致性
- CAP理论，CAP三者不可兼得
  - 让大家接受了最终一致性方案，如Dynamo， Riak等Nosql
- Paxos/Raft 协议进化到了CP+HA
  - 谷歌Chubby达到了99.99958%的平均可用性，一年130s不可用

# 系统层一致性方案

- ACID中的C：数据完整性
- 单机数据库：单机事务保证A-100 -> B+100 作为一个原子操作
- 分布式数据库：poculator提交保证原子操作
  - 多处修改，一处提交，复制提交结果

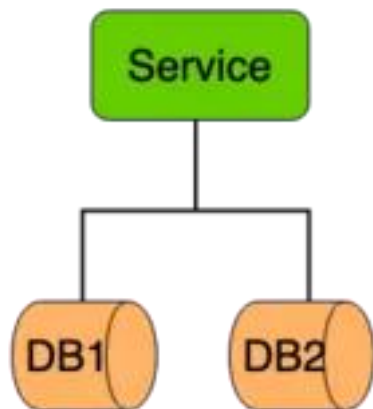
# 应用层一致性例子

- redis缓存与数据库的一致性问题
- 分布式系统中订单
  - 优惠券，库存但是单独服务，保证同时成功或同时回滚
- 微服务领域下，场景非常多 ...

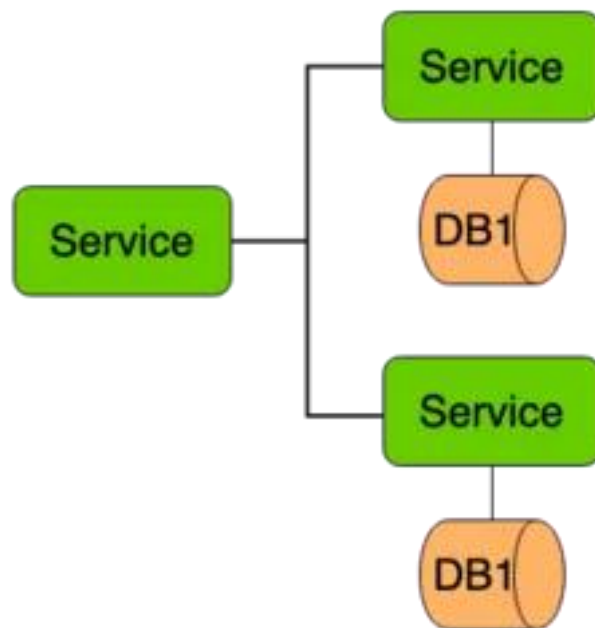


# 应用层一致性

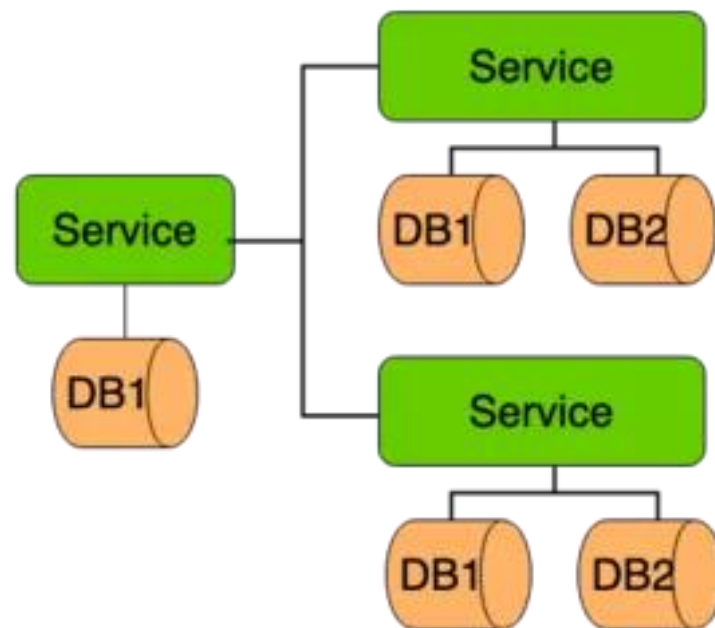
跨数据库



跨服务



混合



# 各种模式简介

- 消息一致性：一个事务里修改业务和消息表，消费消息做后续修改
- XA：数据库支持的分布式事务
- Saga：直接改数据，出错则反向补偿
- TCC：Try-Confirm-Cancel，一种较复杂的事务模式

# 没有完美的方案

事务模式	宽松一致性	开发量	性能	回滚支持
MSG	无中间态	低	高	×
XA	无中间态	低	低	✓
TCC	内部可见中间态	高	较低	✓
SAGA	外部可见中间态	较高	高	✓

# 各种模式适用场景

- 消息最终一致性：适合不需要回滚的场景
- XA：适合并发度不高的业务，或没有锁争抢的业务
- Saga：适合需要回滚，兼顾性能的场景
- TCC：适合一致性强，且性能高的场景

# DTM：一站式的数据一致性解决方案

- 支持以上各种模式
- 支持Go、C#、PHP、Java、Python、Nodejs等多种语言
- 支持多种模式的混合使用
  - 本质上各种模式都是针对一个子事务



# DTM：成熟稳定的开源方案

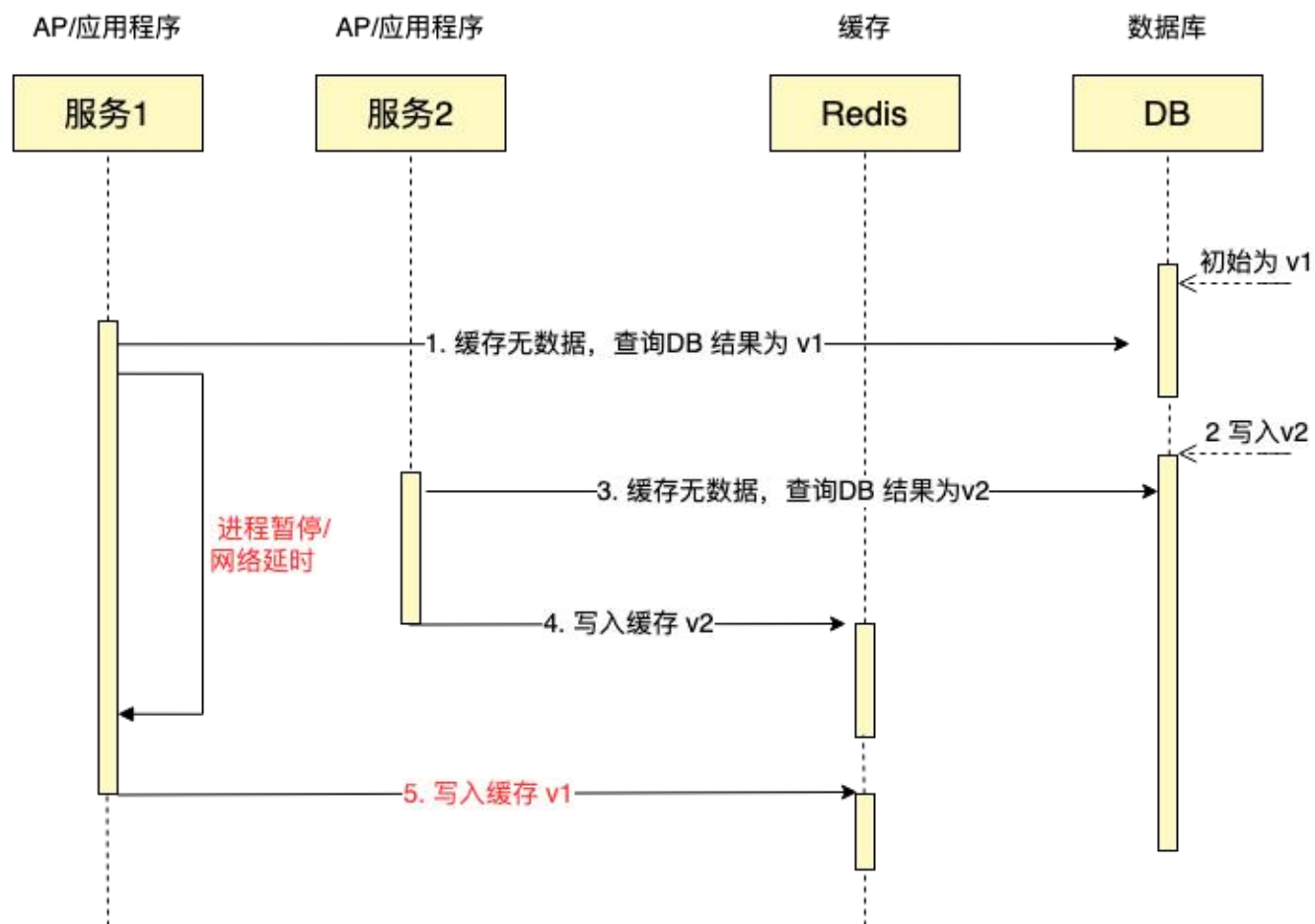
- 一年多的时间7.7K start
- 四十多家公司线上使用
- 腾讯多个事业部在用
- 字节有两个部门在用

# 缓存一致性

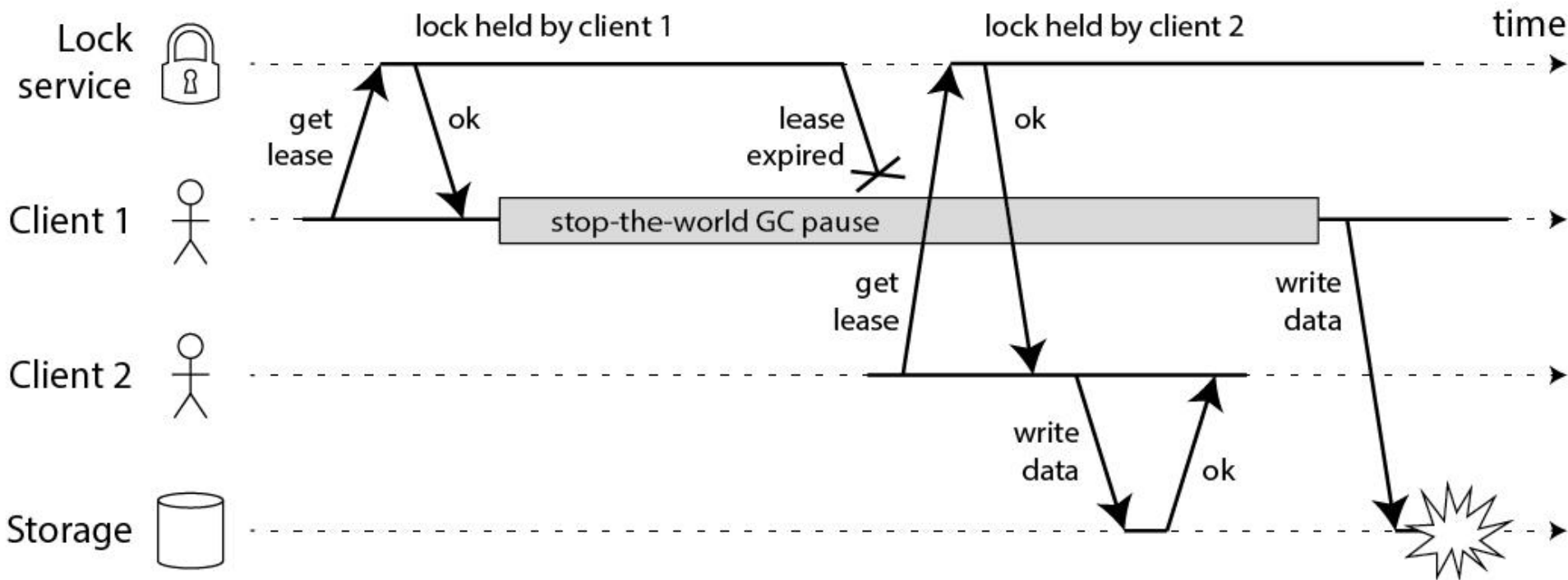
# 现有的典型方案

- 修改数据
  - 修改数据库
  - 监听binlog或轮询消息表
  - 删除缓存
- 查数据
  - 缓存未命中
  - 读取数据库
  - 写入缓存

# 现有问题



# 分布式锁无法解决问题



此图来自DDIA作者关于Redis锁的讨论



# 对现有问题的妥协

- 方法一：允许这种不一致
  - 通过设定较短的过期时间，减少不一致的时长
  - 通过延迟删除等方式，降低不一致的概率
- 方法二：应用层引入 版本
  - 对于关键数据，在应用层维护数据的版本，避免旧版覆盖新版
  - 因为涉及应用层修改，导致这种方案的维护成本很高

# 新方案

- 基于DTM衍生出来，同时也支持独立于DTM之外使用
- 开源项目rockscache：确保缓存一致
- 使用简单，两个API：
  - 查数据：Fetch
  - 更新数据库后删除缓存：TagAsDeleted

# 新方案原理

- 自动维护版本信息，应用层无需关心
- 通过版本比较，避免旧版本覆盖新版本
- 性能与常见方案差别非常小

# 新方案原理细节

- 缓存多出版本字段，采用uuid作为版本
- 并不直接删除数据，而是标记删除
- 查缓存未命中时，写入uuid
- 最后写入缓存时检查uuid，确保未变更

# 新方案优势

- 确保旧版本不会覆盖新版本，保证了一致性
- 应用层不用维护版本信息，适用绝大部分数据缓存场景
- 也支持缓存和数据库保持强一致
- 支持防击穿、防穿透、防雪崩



# 如何做到强一致

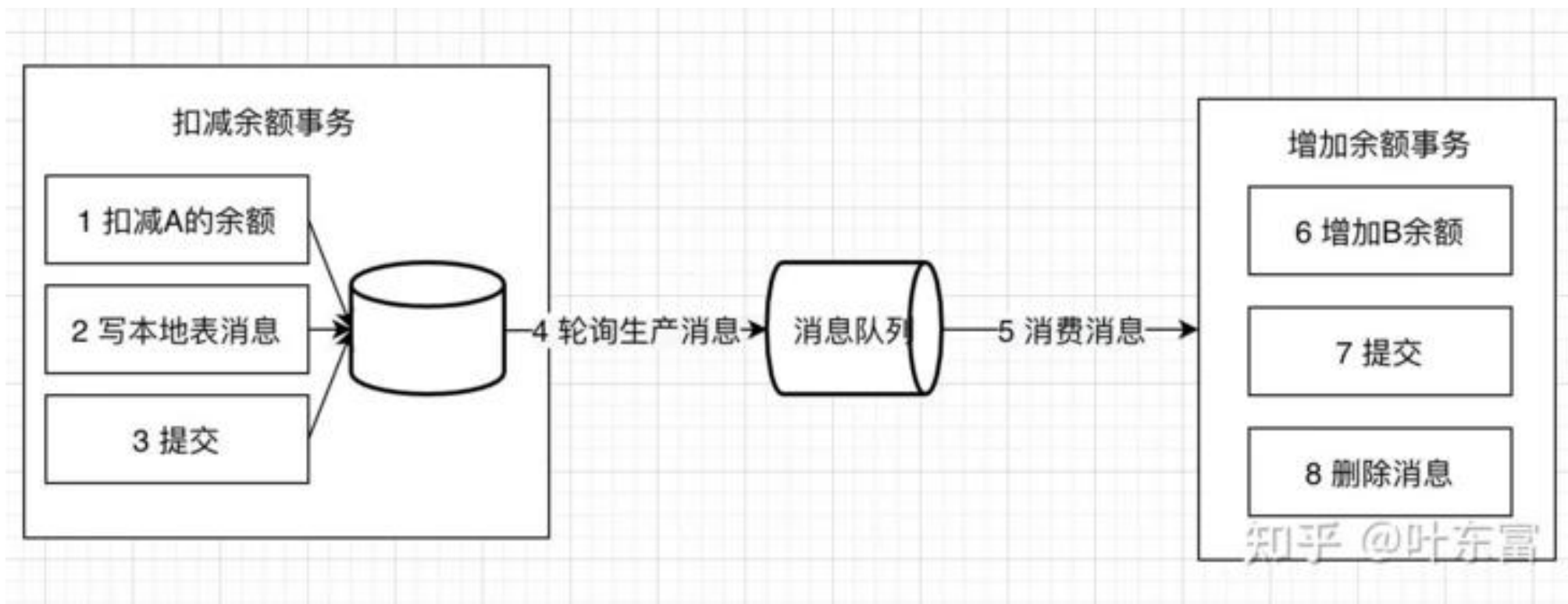
- 入口需要统一，类似CPU的缓存，或磁盘的缓存
- 所有的数据查询都通过rockscache的Fetch访问
- 所有的数据修改，只有当TagAsDeleted成功后，才告知用户成功

# 消息一致性

# 问题场景

- 跨行转账的业务场景，无法用本地数据库
- 假设：转出可能失败，转入不会失败
- 如何在转出成功时，确保转入一定成功？

# 经典的发件箱模式

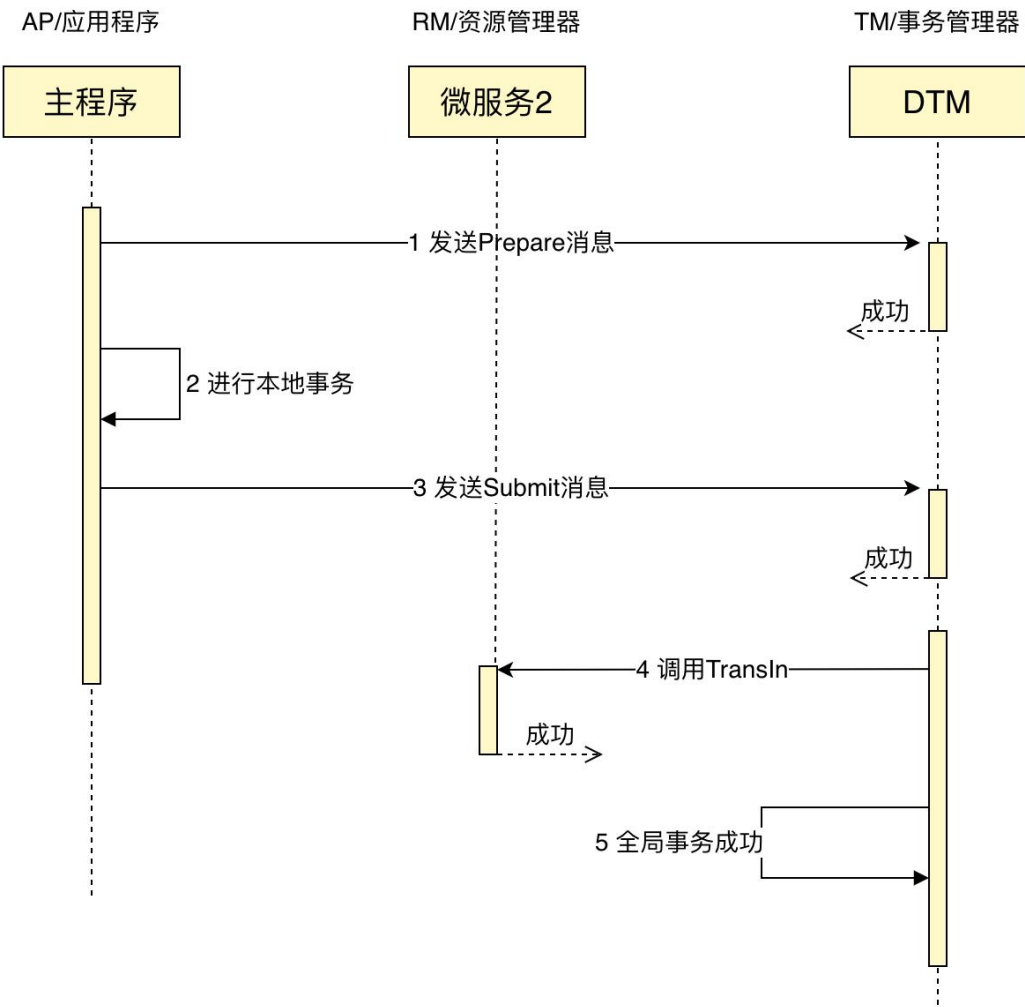


# 现存的问题

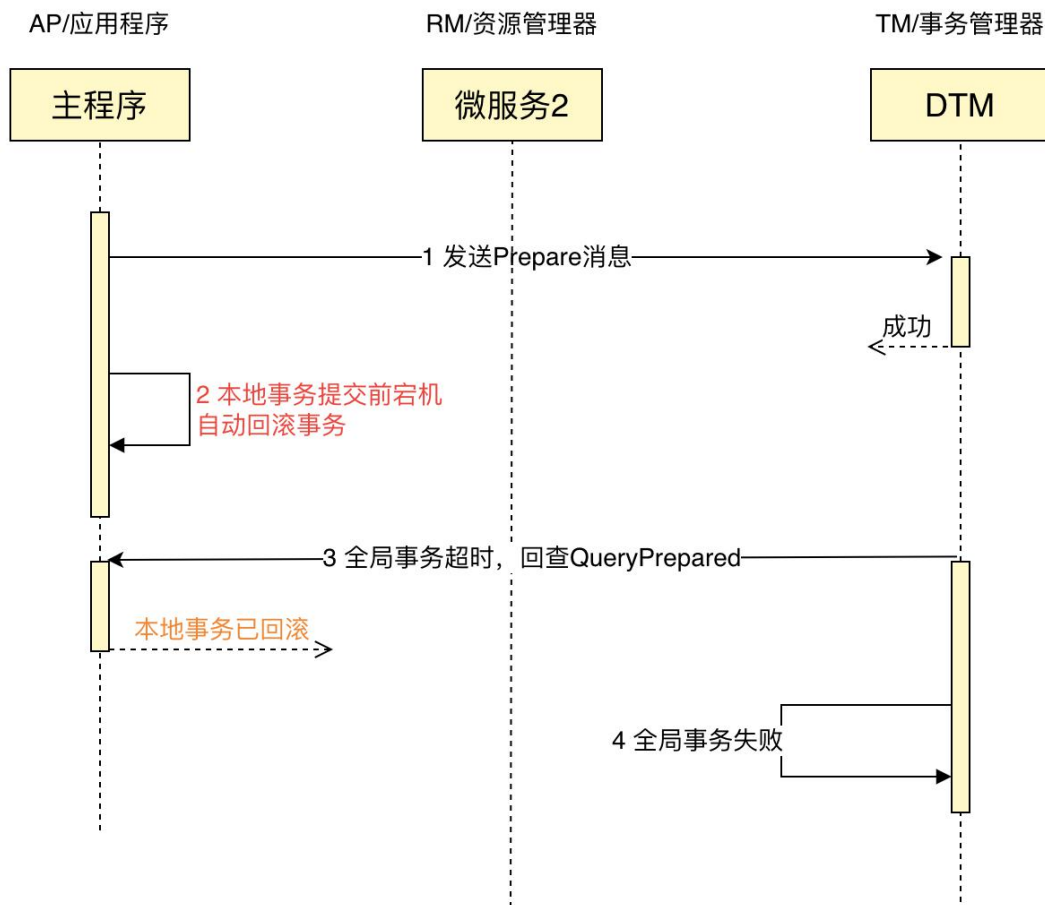
- 轮询生产消息难实现：
  - 轮询逻辑：延时大，难以高效
  - binlog方式：重，难维护，只适用于SQL类
- 难维护：每个数据库实例，都需要维护生产消息的任务



# 二阶段消息原理--正常



## 二阶段消息原理--异常



# 与RocketMQ事务消息异同

- 相同点：
  - 都有回查
- 不同点：
  - DTM的回查是SDK自动完成，事务消息需要业务自己写
  - 对于已回滚，DTM一次查出结果，事务消息查询很多次
  - 极端情况，DTM保证正确结果，事务消息不能确保

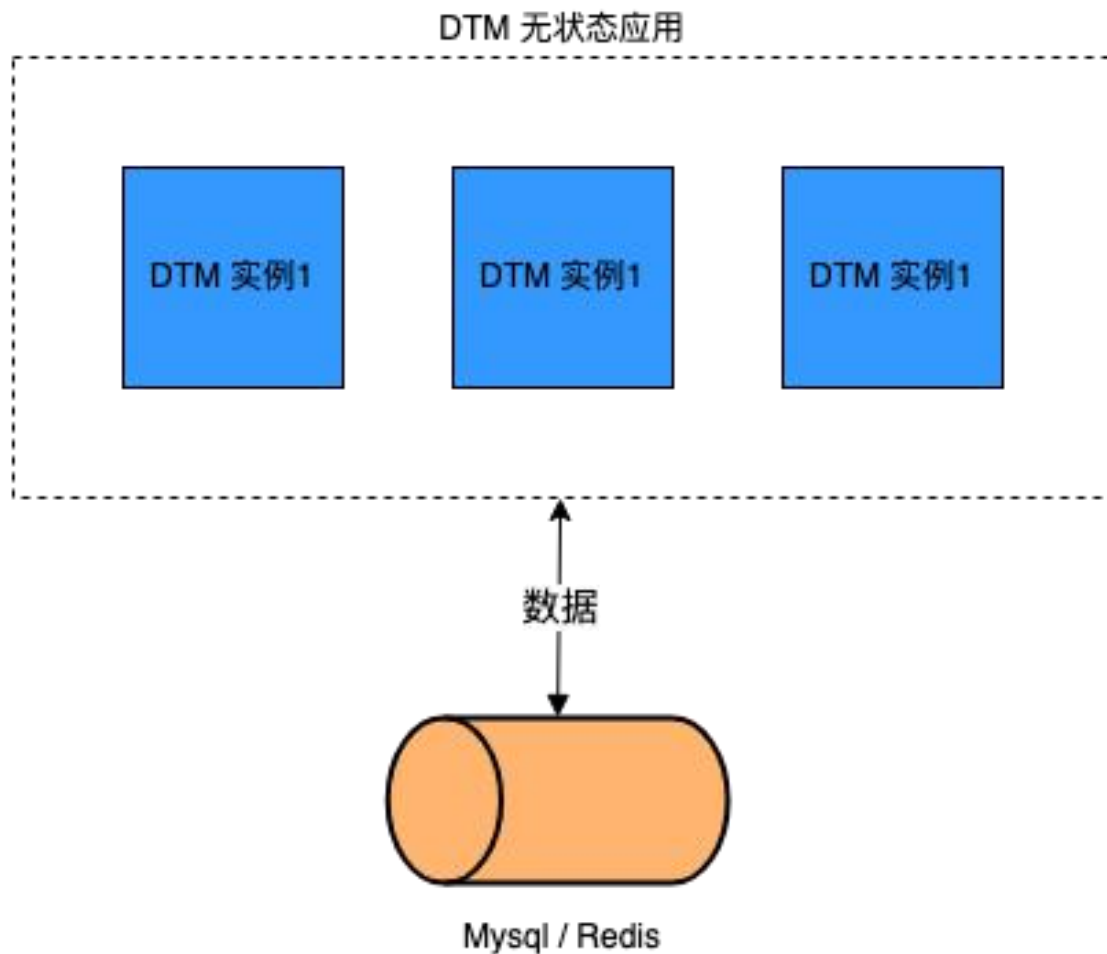
## 二阶段消息使用

```
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")  
})
```

- DoAndSubmit保证，本地事务执行成功的话，指定的API一定会最终完成

# 架构简单易用

- 简单优雅!!!
- 仅依赖dtm分布式事务管理器
- 无消息队列
- 全部是简单的服务调用



# 支持情况

- 不仅支持数据库，还支持Redis, Mongo
- 已支持Go, C#, Nodejs, PHP




# 异构存储的一致性

# 支持的存储引擎

- SQL类: Mysql、Postgres、SQL-Server(C#)
- Redis
- Mongo

# 总体效果

 Order			
 Coupon	 Payment	 Account	 Stock
 mongoDB	 MySQL	PostgreSQL 	 redis

# 扩展

- 很容易就可以将DTM扩展到其他的存储引擎
- 只需要实现SDK中的“子事务屏障”即可
- 存储引擎要求：支持事务操作
- 例如：TiKV，SQLite

# 小结

# DTM：一站式的数据一致性解决方案

- 支持各种分布式事务模式，以及各模式的混合使用
- 支持Go、C#、PHP、Java等多种语言
- 支持SQL、MongoDB、Redis各种存储引擎



# DTM vs Seata

- DTM比Seata晚两年多开源，借鉴了Seata的很多优点
- 更加轻量，多语言的支持比Seata更完善
- 支持的模式多两种，二阶段消息及Workflow
- 拓宽了应用场景，例如缓存一致性
- 对云原生、对微服务的支持更友好

# DTM 目标

## 成为微服务和云原生架构的核心中间件

- 地址：[github.com/dtm-labs/dtm](https://github.com/dtm-labs/dtm)
- 欢迎 PR、Issue、Star。





THANKS