可靠事件系统 TEVENT

2016.1024

一、可靠事件模式

- 可靠事件投递有两个特性:

1)每个服务原子性的完成业务操作和发布事件;

2)消息代理确保事件投递至少一次(at least once)。

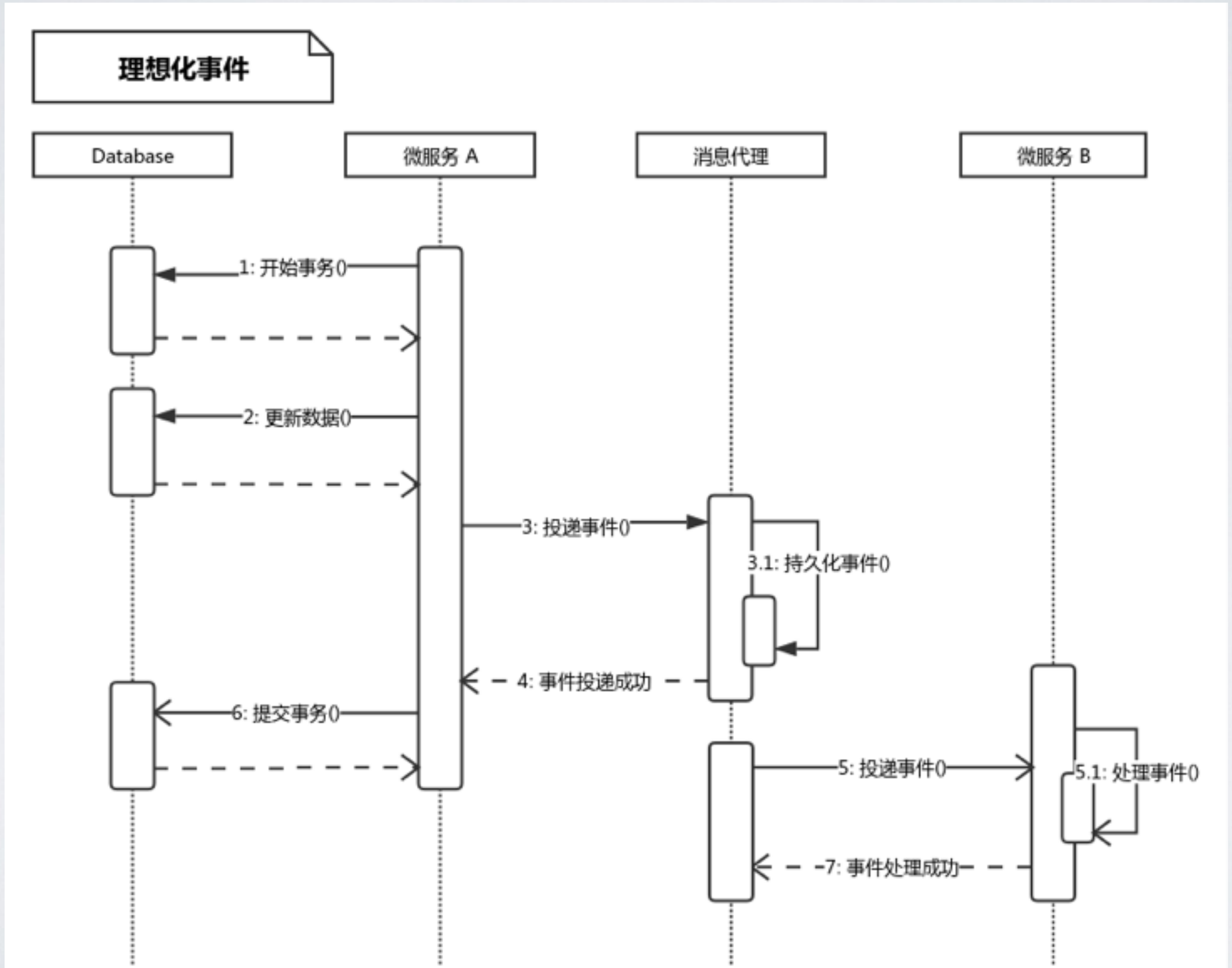
- 避免重复消费要求消费事件的服务实现幂等性。

二、可靠事件投递

潜在风险

```
public void trans() {
   try {
     // 1. 操作数据库
     bool result = dao.update(mode1);// 操作数据库失败，会抛出异常
     // 2. 如果第一步成功，则操作消息队列（投递消息）
     if(result){
       mq.append(mode1);// 如果mq.append方法执行失败，会抛出异常
     }
   } catch (Exception e) {
     roolback();// 如果发生异常，就回滚
   }
}
```

理想化事件


理想化事件
Database
微服务 A
消息代理
微服务 B
1: 开始事务()
2: 更新数据()
3: 投递事件()
3.1: 持久化事件()
4: 事件投递成功
6: 提交事务()
5: 投递事件()
5.1: 处理事件()
7: 事件处理成功

理想化事件

根据上面代码和时序图，理想化的情况会出现3中情况：

1、操作数据库成功，向消息代理投递事件也成功

2、操作数据库失败，不会向消息代理中投递事件了

3、操作数据库成功，但是向消息代理中投递事件时失败，向外抛出了异常，刚刚执行的更新数据库的操作将被回滚

不理想的事件

服务做完本地事务后，出现两种不理想情况

1、投递消息失败或同步获取消息系统返回的投递结果时超时

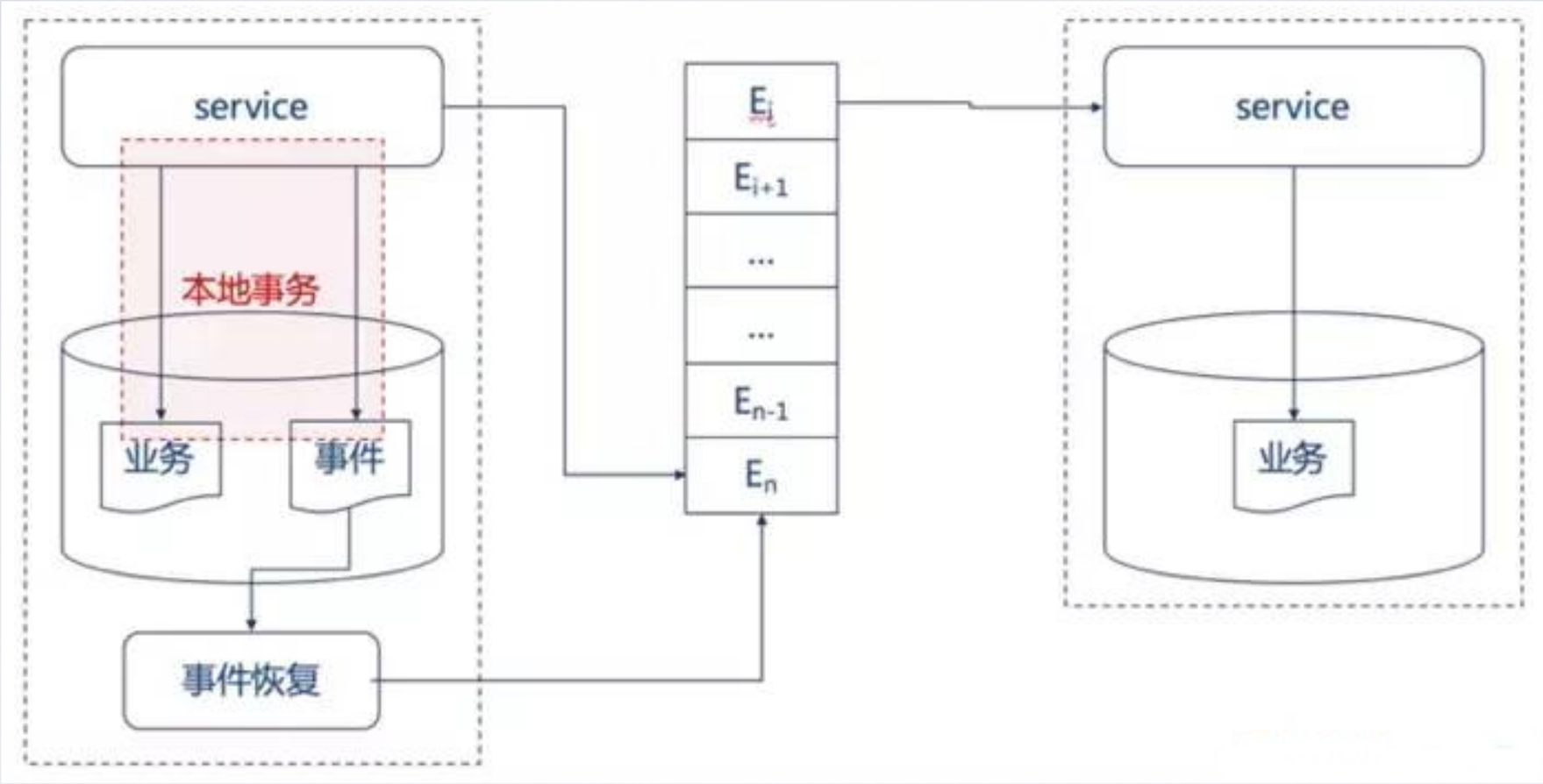
2、应用宕机或者发布原因重启，没来得及把消息发出去

不知道消息是否投递成功或者没有投递消息

三、可靠事件投递的两种实现

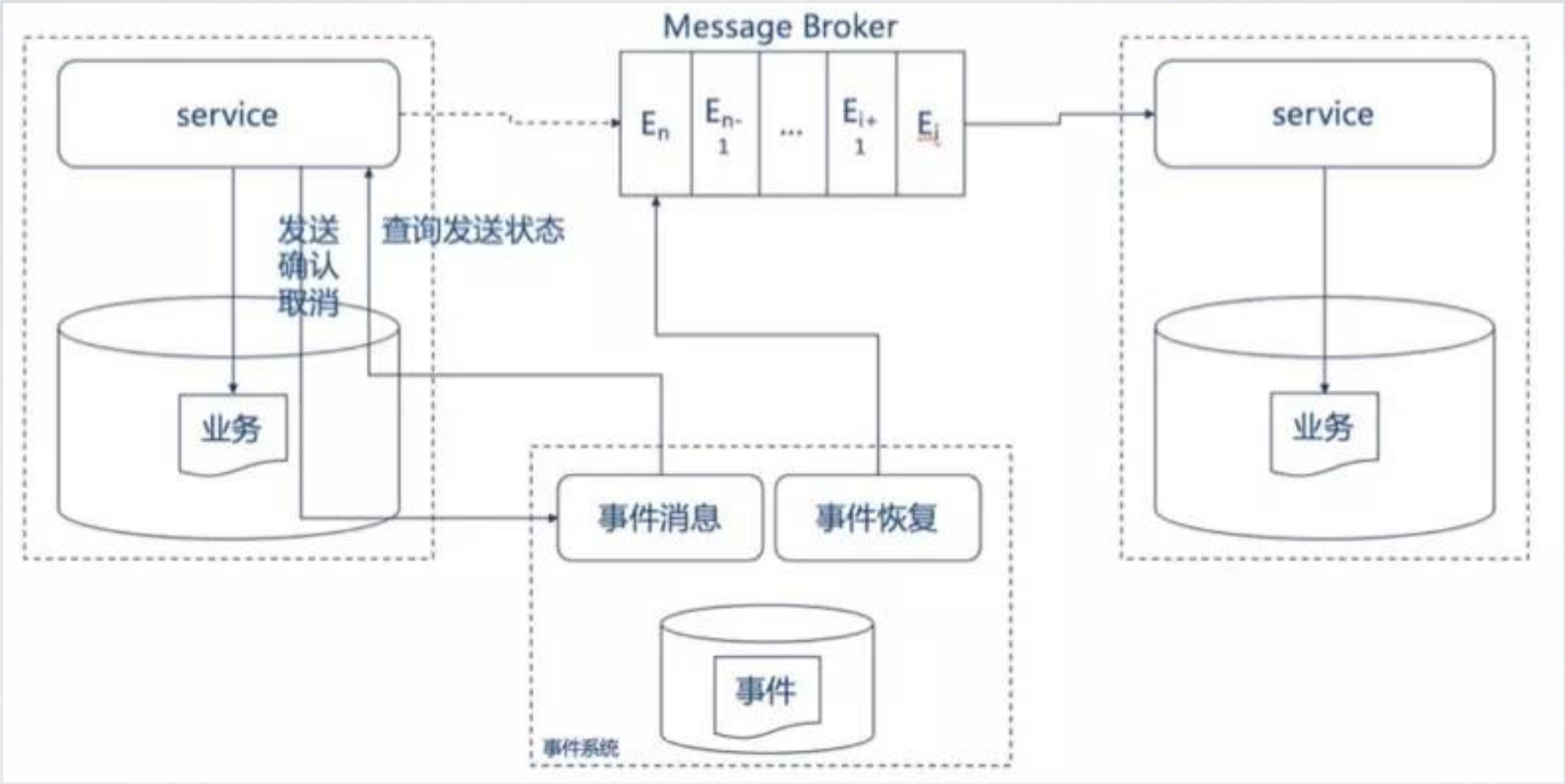
- 1、本地事件表
- 2、外部事件表

1、本地事件表



- 微服务在同一个本地事务中记录业务数据和事件数据
- 微服务实时发布一个事件关联业务服务中，如果事件发布成功立即删除记录的事件，这样能够保证事件投递的实时性
- 事件恢复服务定时从事件表中恢复未发布成功的事件，重新发布，重新发布成功后删除记录的事件，这样能够保证事件一定能够被投递。

2、外部事件表



- 业务服务在事务提交前，通过实时事件服务向事件系统请求发送事件，事件系统只记录事件并不真正发送
- 业务服务在提交后，通过实时事件服务向事件系统确认发送，事件得到确认后事件系统才真正发布事件到消息代理
- 业务服务在业务回滚时，通过实时事件向事件系统取消事件
- 事件系统的事件恢复服务会定期找到未确认发送的事件向业务服务查询状态，根据业务服务返回的状态决定事件是要发布还是取消

四、可靠事件系统TEVENT原理

原理：生产者的事务消息，保证消息被（MQ）可靠地投递

2PC保证了消息发送与本地事务同时成功或同时失败

第1阶段 消息发送者发送Prepared消息到 EventSystem,得到事务消息Id;
第2阶段 消息发送者执行本地事务;
第3阶段 根据本地事务执行结果，发送确认或者回滚消息(事务消息Id)。

如果第三阶段确认消息发送失败了怎么办?

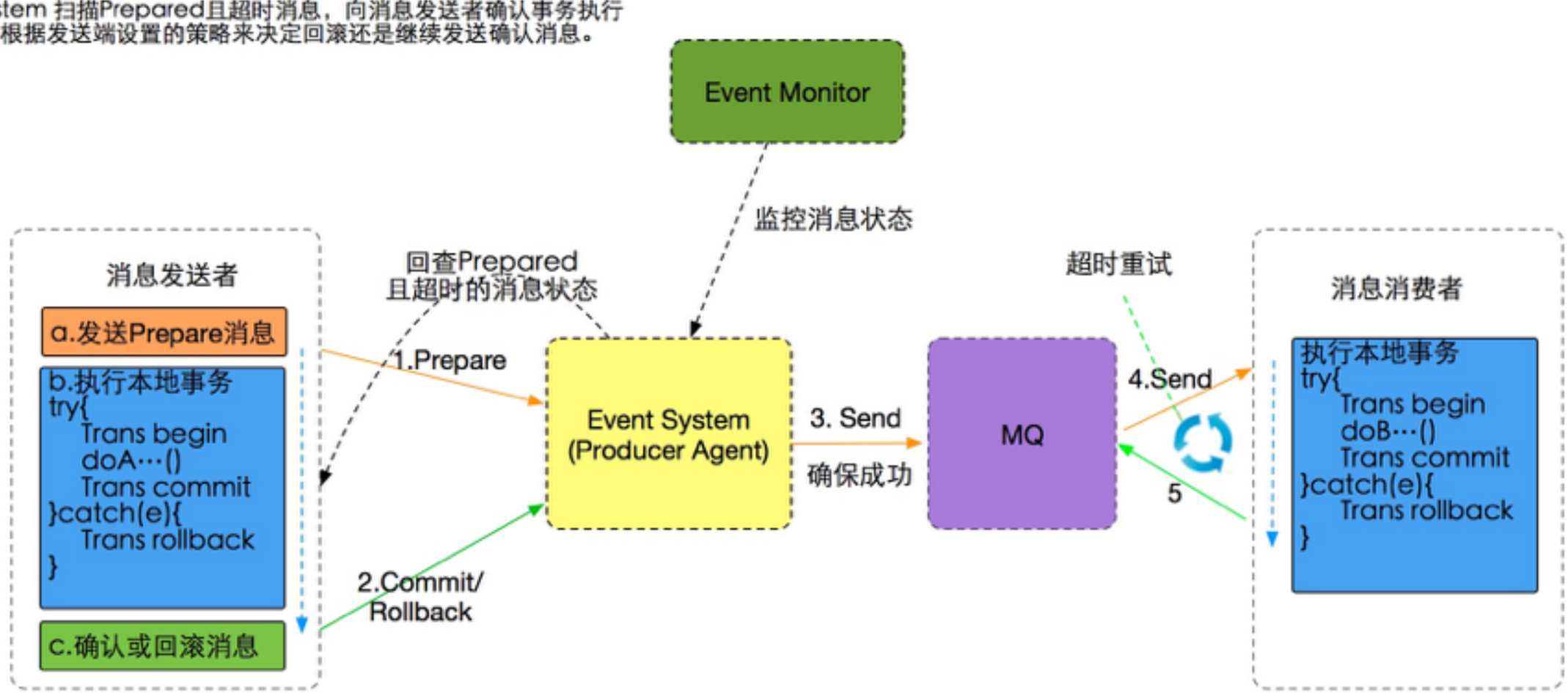
可靠事件系统 扫描处于Prepared且超时消息，向消息发送者确认事务执行状态，并根据发送端设置的策略来决定回滚还是继续发送确认消息。

生产者事务消息

生产者事务消息

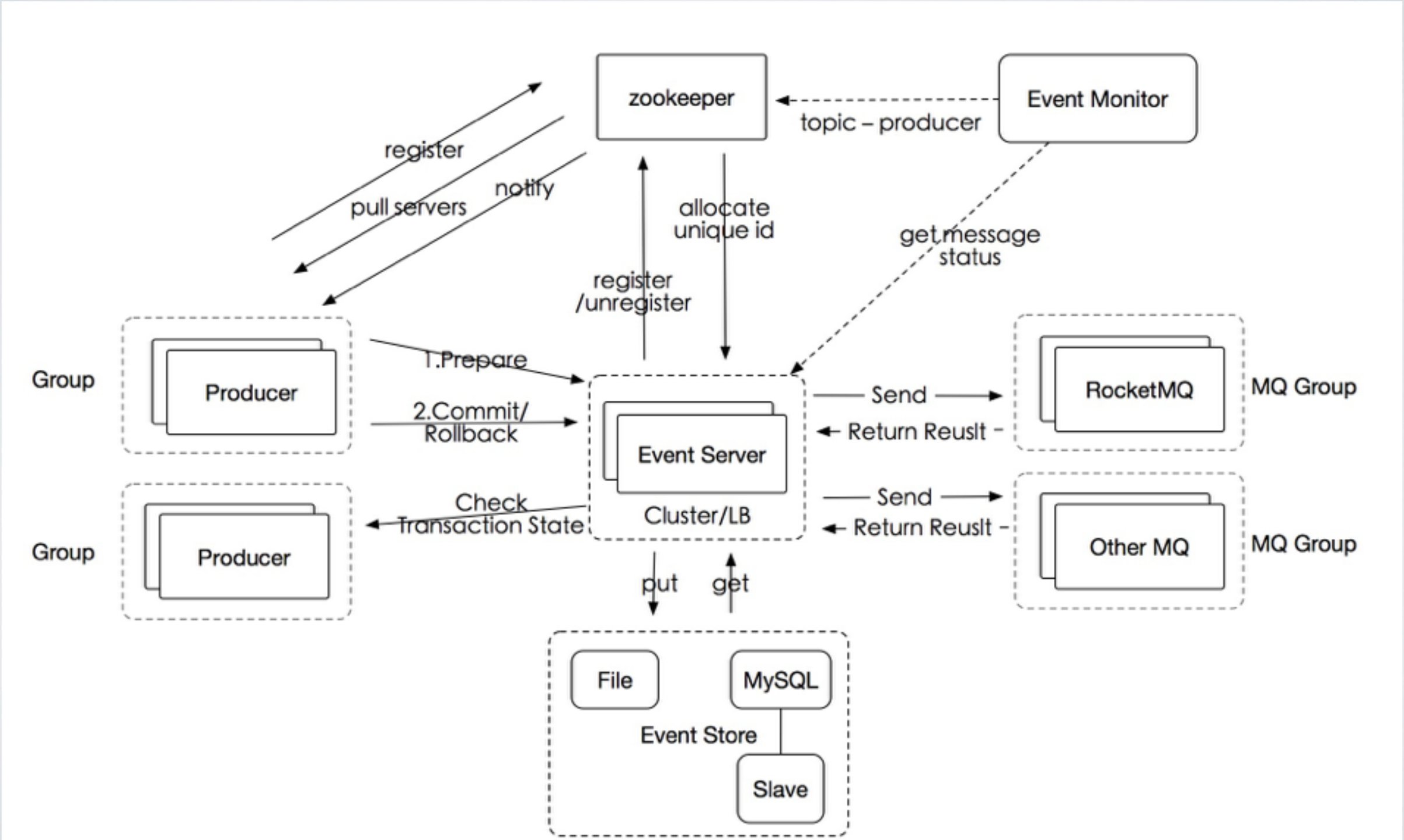
2PC保证了消息发送与本地事务同时成功或同时失败
第一阶段 消息发送者发送Prepared消息到 EventSystem,得到事务消息Id;
第二阶段 消息发送者执行本地事务;
第三阶段 根据本地事务执行结果，发送确认或者回滚消息(事务消息Id)。

如果第三阶段确认消息发送失败了怎么办?
EventSystem 扫描Prepared且超时消息，向消息发送者确认事务执行状态，并根据发送端设置的策略来决定回滚还是继续发送确认消息。



五、TEVENT设计与实现

1、TEVENT设计


zookeeper
Event Monitor
topic – producer
register
pull servers
notify
allocate
unique id
get message
status
register
/unregister
Group
Producer
1.Prepare
2.Commit/
Rollback
Event Server
Cluster/LB
Send
Return Reuslt
RocketMQ
MQ Group
Check
Transaction State
Group
Producer
Send
Return Reuslt
Other MQ
MQ Group
put
get
File
MySQL
Event Store
Slave

2、代码结构

- client: 客户端，集成到应用，事务消息发送接口
- cluster: 集群策略和负载均衡
- common: 公用的代码
- registry：注册中心，提供注册和自动发现
- rpc：远程通讯框架，基于NIO
- server：服务端，消息接受和代理发送，事务控制和事务状态回查
- store：数据存储

高性能传输

- 底层用NIO的Socket通讯，基于Netty框架封装
- 远程调用支持Sync, Async和OneWay

高可用

- 1、Server集群，多master部署

- 2、失败处理策略

数据可靠和一致性

- 使用数据库存储，如MySQL
- 由DB保证数据存储可靠，避免多server数据的同步问题

服务注册和发现

- 注册中心（如ZooKeeper）来集群注册和发现
- 客户端自动感知服务上下线

重点功能

- I、网络传输协议
- 2、集群注册和发现（zk）
- 3、分布式ID生成（twitter snowflake）
- 4、分表分库
- 5、负载均衡 LoadBalance
- 6、服务调用失败策略 Failover

1、网络传输协议

协议

com.tongbanjie.tevent.rpc.protocol

1、出入参对象RpcCommand（Request／Response）

2、序列化（Json／ProtoStuff）

协议包格式

```java
/**
 * 协议格式 <p>
 * 协议包格式
 * <length> <protocol type> <header length> <header data> <body data>
 *  1 int   1 byte            1 int
 *
 * @author zixiao
 * @date 16/9/28
 */
public abstract class Protocol {

    //数据总长度为int 占用4个字节
    public static final int TOTAL_LENGTH_SIZE = 4;

    //协议类型为byte 占用1个字节
    public static final int PROTOCOL_TYPE_SIZE = 1;

    //header长度为int 占用4个字节
    public static final int HEADER_LENGTH_SIZE = 4;

}
```

2、集群注册和发现

注册中心

com.tongbanjie.tevent.registry

基于ZooKeeper的注册应用的地址（IP端口）

1、主动拉取可用Server／Client列表

2、通过Watcher监听存储父目录，实时变更可用列表

3、客户端定时检测错误，剔除不可用服务

存储结构

```
* 在ZooKeepr目录结构
* + /tevent
*           + /servers
*                     + /server-000001  [Ephemeral & Seq]
*                     + /server-000002  [Ephemeral & Seq]
*                     ...
*           + /clients
*                     + /client-000001  [Ephemeral & Seq]
*                     + /client-000002  [Ephemeral & Seq]
*                     ...
*           + /serverIds
*                     + /0
*                     + /1
*                     ...
*                     + /31
```

3、分布式ID生成

分布式ID算法

com.tongbanjie.tevent.common.util.IdWorker

改进twitter snowflake算法，生成集群唯一的64位Long型 时间递增ID

Twitter Snowflake算法

Snowflake原理结构如下

0 -0000000000 0000000000 0000000000 0000000000 0

- 00000 -00000 -0000000000 00

第1位为未使用（可作为long的符号位），接下来的41位为毫秒级时间，然后5位datacenter标识位，5位机器ID（并不算标识符，实际是为线程标识），然后12位该毫秒内的当前毫秒内的计数，加起来刚好64位，为一个Long型。

整体上按照时间自增排序，并且整个分布式系统内不会产生ID碰撞（由datacenter和机器ID作区分），并且效率较高，经测试，snowflake每秒能够产生26万ID左右，完全满足需要

TEvent分布式ID算法改动

- 1、［时间前缀］改为44位，使用时间支持从当前时间起278年，［机器Id］5位，［数据中心Id］改为3位

- 2、［毫秒内的计数］随机从0-127开始自增，支持128张分表

4、分表分库

暂未实现

考虑使用分布式ID作为主键和分表Key

缺点使用业务主键或者MessageKey查询需要扫描所有分表

5、负载均衡

LoadBalance算法

com.tongbanjie.tevent.cluster.loadbalance

1、Random

2、RoundRobin

3、WeightedRandom

6、调用失败策略

Cluster策略

com.tongbanjie.tevent.cluster.cluster

1、Failfast：快速失败，只发起一次调用，失败立即报错

2、Failover： 失败转移，当出现失败，重试其它服务

DEMO

Case 1: 集群调用

负载均衡，调用失败策略

Case 2: 事务消息

模拟本地事务各种情况

Server向生产者反查事务状态

源码

- https://github.com/beston123/tevent