消息队列架构设计文档

一、业务背景

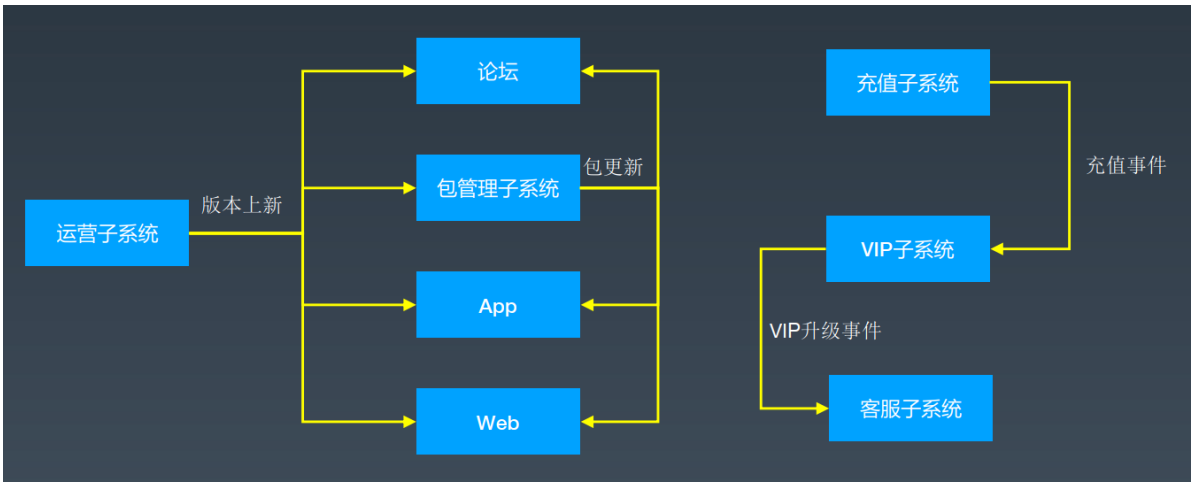
2014年左右，游戏业务发展很快，系统也越来越多，系统间协作的效率很低，例如：
【新版本发布了】

- 游戏厂家更新游戏版本后，运营人员获取最新的游戏包，更新版本信息，然后上传包到包管理系统打测试包，运营人员进行基本测试。运营子系统通知论坛有新的包将要发布，进行预热。
- 测试完成后，运营管理子系统要通知包管理系统进行打包
- 游戏准点正式发布的时候，运营子系统要通知App、Web站点等即时更新到新版本

【玩家充钱了】

- 玩家进行充值，充值完成后充值子系统通知VIP子系统；
- VIP子系统判断玩家等级，达到VIP后，等级子系统要通知福利子系统进行奖品发放，要通知客服子系统安排专属服务人员，要通知商品子系统进行商品打折处理……
- 等级子系统的开发人员也是不胜其烦。

下图是目前系统的架构，基于以上的需求，整体架构需要引入消息队列，降低系统之前的耦合，削峰填谷。



二、约束和限制

- 业务优先考虑可用性
- 各种维护操作要方便，例如收发消息情况、权限控制、上下线等
- 开发投入人力和时间不能太长
- 中间件团队规模不大，大约6人左右。
- 中间件团队熟悉Java语言，但有一个同事C/C++很牛。
- 开发平台是Linux，数据库是MySQL。
- 目前整个业务系统是单机房部署，没有双机房。
- 刚刚被阿里以创纪录的金额收购。

三、总体架构

3.1 架构分析

3.1.1 高性能

游戏新版本发布和VIP充值的消息并不多，所以不需要高性能。

3.1.2 高可用

游戏新版本发布和VIP都是公司的营收业务，属于高优先级业务，架构需要高可用，避免单点故障。

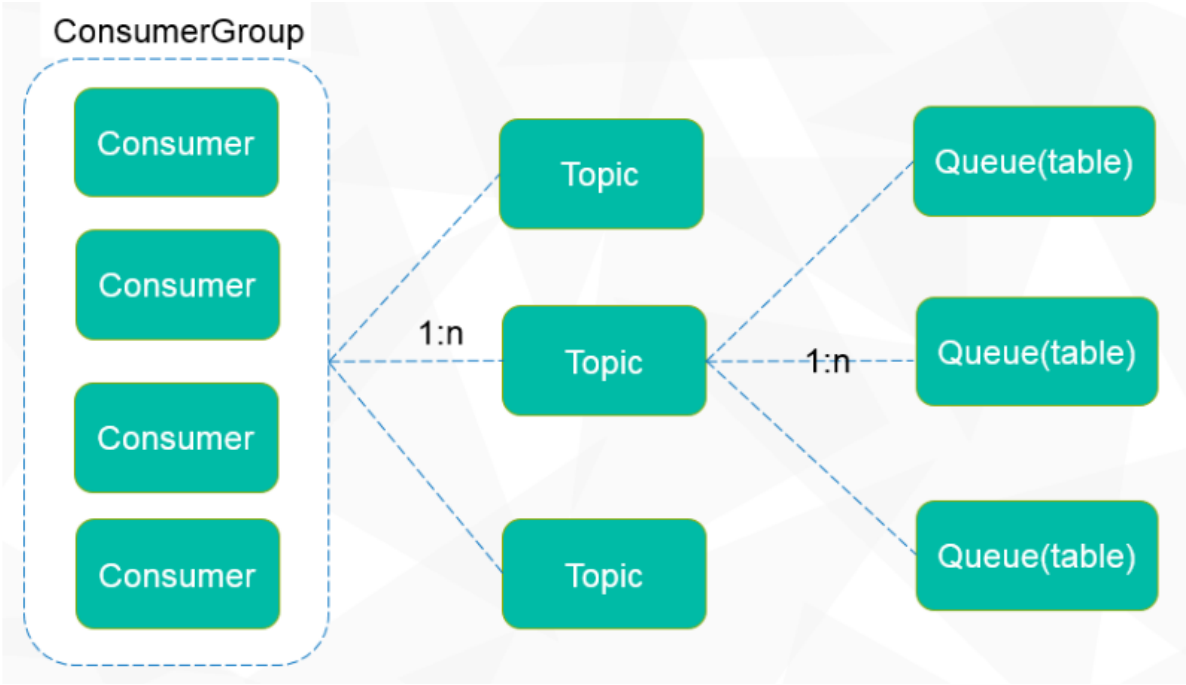
3.1.3 可拓展

消息队列的功能基本明确，无需可拓展。

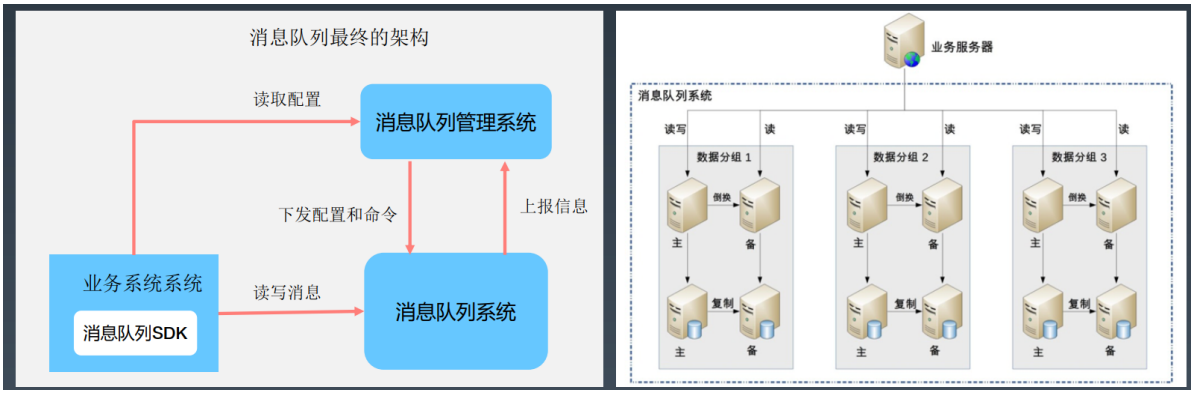
3.1.4 成本

该项目是在已有架构的基础上，进行优化，开发投入的人力和时间不能太久。

3.2 总体架构



- ConsumerGroup：表示一些topic集合，被一组consumer订阅，这组consumer就形成了一个消费者组。
- Topic：即消息主题，topic包含多个queue（类似于kafka中Partition）。
- Queue: 即存储消息的队列，每个queue对应数据库中的一张表，存储在MySQL表中。



整体的系统架构分成三部分：业务系统（也就是提供的消息队列SDK）、消息队列系统（包括：消息处理、消息持久化）、消息队列管理系统。

3.2.1 消息队列SDK

- 采取轮询的策略写入和读取消息。
- SDK心跳和偏移提交。
- 失败消息重试和发送。

3.2.2 消息队列系统

采用数据分散集群的架构，集群中的服务器进行分组，每个分组存储一部分消息数据。

每个分组包含一台主 MySQL 和一台备 MySQL，分组内主备数据复制，分组间数据不同步。

正常情况下，分组内的主服务器对外提供消息写入和消息读取服务，备服务器不对外提供服务；主服务器宕机的情况下，备服务器对外提供消息读取的服务。

3.2.3 消息队列管理系统

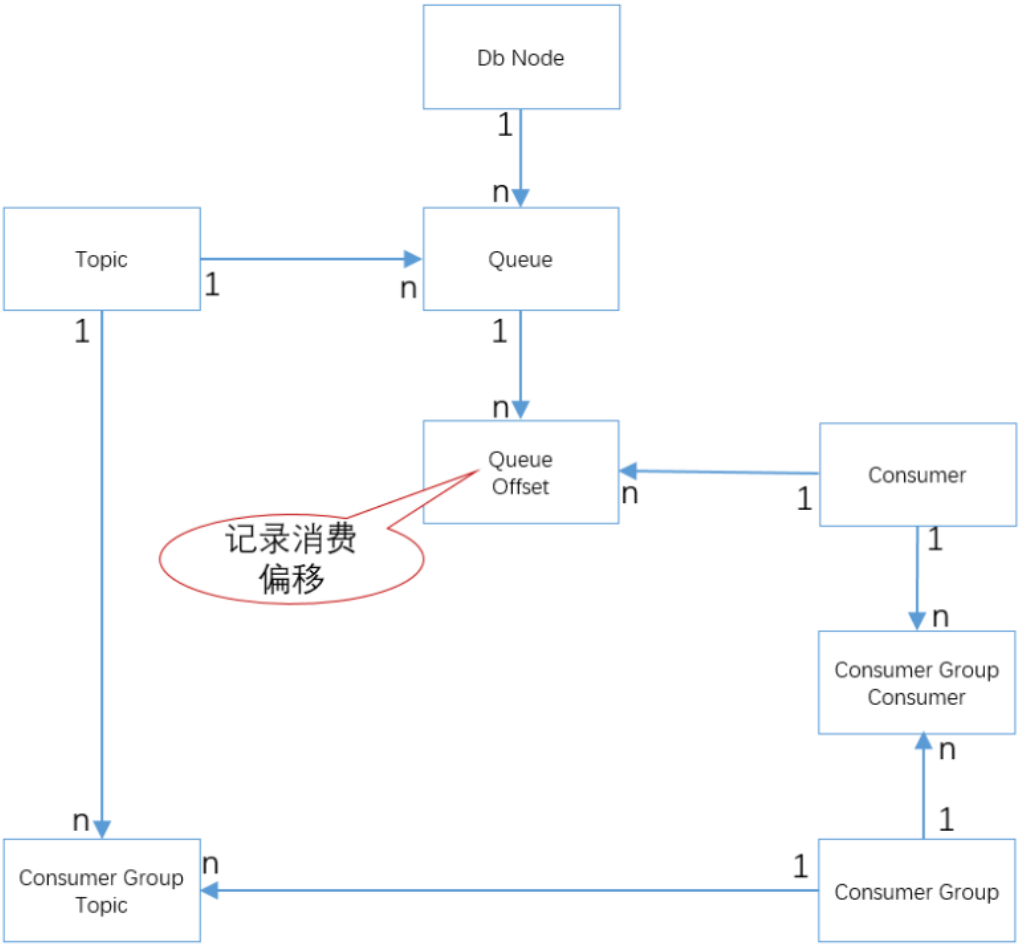
消息队列管理系统为该系统提供图形化管理界面。

- consumerGroup、topic、数据节点的初始化、创建、治理
- 消费治理：偏移调整、消费启停、消费端监控等
- 消息查询、失败消息的手动重新发送
- 界面操作的审计日志和权限控制
- 各种监控统计报表
- 历史消息清理的定时器、元数据处理的定时器

4. 详细设计

4.1 核心功能

4.1.1 元数据库设计



- DbNode

  记录消息库的节点信息，每一条记录表示一个消息库节点（记录了消息库的：ip、端口、库名、等连接信息）。

- Queue

  记录queue(消息队列)的信息（每个queue对应数据库中的一张表）。记录topic与queue的分配信息。

- Topic

  消息主题，拥有多个queue来存储消息。

- QueueOffset

  记录consumer和queue的对应关系（即某个consumer消费某个queue的消息）。记录消费偏移（即consumer消费到了什么位置）。

- Consumer

  消费者。

- ConsumerGroup

  一些topic集合，被一组consumer订阅，这组consumer就形成了一个consumerGroup。

- ConsumerGroupConsumer

  记录ConsumerGroup和Consumer的对应关系。

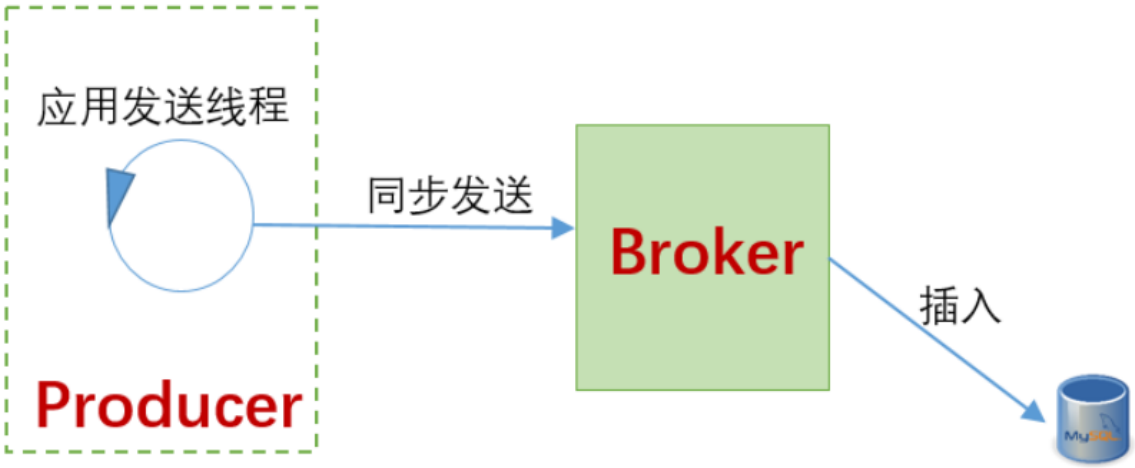
- ConsumerGroupTopic

  记录consumerGroup和topic的订阅关系。

4.1.2 消息发送

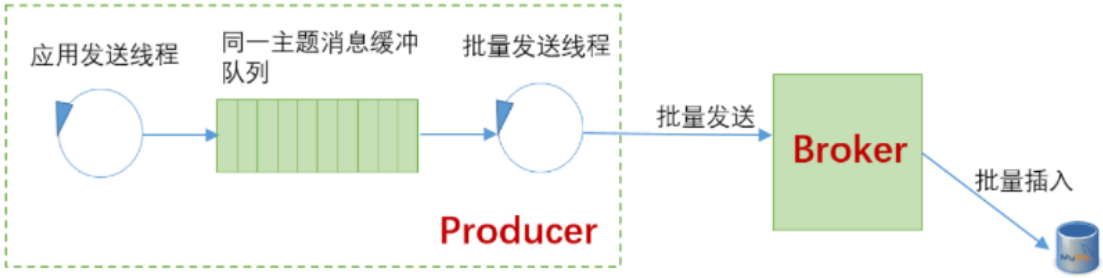
消息队列系统提供SDK供各业务系统调用，SDK从消息管理系统中读取所有消息队列系统的服务器信息，SDK采取轮询算法发起消息写入请求给主服务器。如果某个主服务器无响应或者返回错误，SDK将发起请求发送到下一台主服务，相当于在客户端实现了分片的功能 。

（1）同步发送



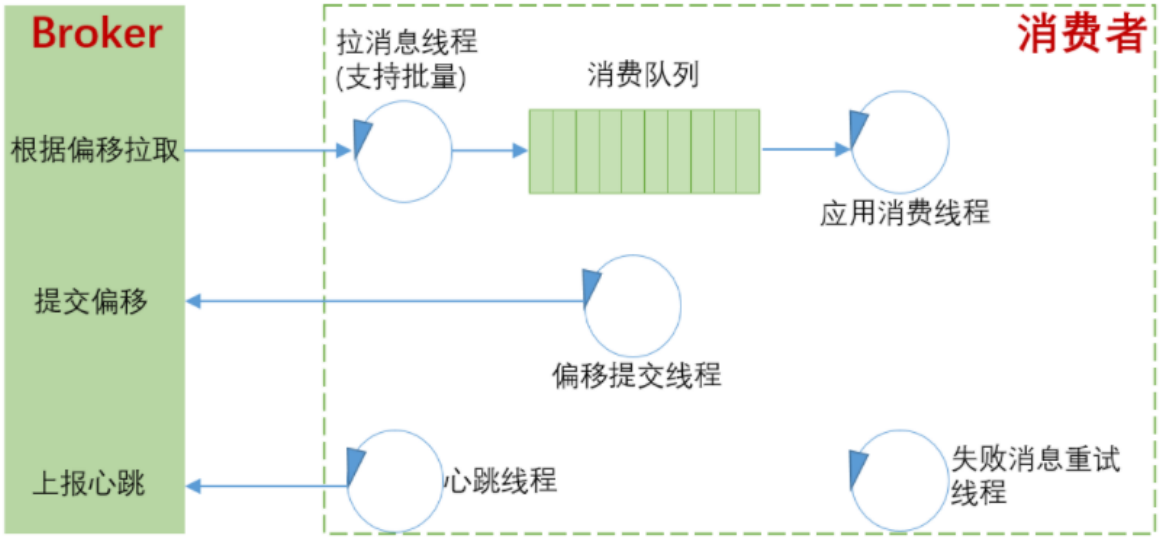
发送端把消息发送到broker，broker把消息存储到消息库，然后返回存储结果。

（2）异步发送



1. 应用发送线程：把消息发送到本地缓冲队列。
2. 批量发送线程：把缓冲队列中的消息批量发送到broker。
3. broker把消息批量插入到消息库。

4.1.3 消息消费



1. 消费线程：根据消费偏移（即该消费者当前的消费位置），从broker拉取消息到本地消费队列。

2. 应用消费线程：从本地消费队列中拉取消息，进行消费。
3. 偏移提交线程：定时提交当前的消费位置到broker。
4. 心跳线程：定时上报心跳到broker。
5. 失败消息重试线程：对失败消息进行重试消费。

4.2 关键设计

1）消息发送可靠性

业务服务器中嵌入消息队列系统提供的 SDK，SDK 支持轮询发送消息，当某个分组的主服务器无法发送消息时，SDK 挑选下一个分组主服务器重发消息，依次尝试所有主服务器直到发送成功；如果全部主服务器都无法发送，SDK 可以缓存消息，也可以直接丢弃消息，具体策略可以在启动 SDK 的时候通过配置指定。

如果 SDK 缓存了一些消息未发送，此时恰好业务服务器又重启，则所有缓存的消息将永久丢失，这种情况 SDK 不做处理，业务方需要针对某些非常关键的消息自己实现永久存储的功能。

2）消息存储可靠性

消息存储在 MySQL 中，每个分组有一主一备两台 MySQL 服务器，MySQL 服务器之间复制消息以保证消息存储高可用。如果主备间出现复制延迟，恰好此时 MySQL 主服务器宕机导致数据无法恢复，则部分消息会永久丢失，这种情况不做针对性设计，DBA 需要对主备间的复制延迟进行监控，当复制延迟超过 30 秒的时候需要及时告警并进行处理。

3）消息如何存储

每个消息队列对应一个 MySQL 表，消息队列名就是表名，表结构设计为id(顺序自增)，biz_id(业务查询主键)，body(消息体)，send_time(发送时间)，receive_time(接收时间)。

4）Broker服务器高可用

- 同一组的主从服务器配置相同的group名称，在ZooKeeper建立对应的PERSISENT节点
- 主从服务器启动后，在ZooKeeper对应的group节点下建立EPHEMERAL节点，名称分为为master和slave
- 从服务器watch主服务器的master节点状态，当master节点超时被删除后，从服务器接管读消息，收到客户端SDK的读消息请求后返回
  消息，收到客户端SDK的写请求直接拒绝。

4.3 设计规范

1）消息队列服务器使用 Spring Boot + Netty 开发

2）MySQL 使用 Innodb 存储引擎

3）TCP 包的结构设计

5. 质量设计

5.1 消息队列管理后台

5.2 成本

6. 演进规划

6.1 消息队列一期

消息队列先实现上述规划的核心功能。

6.2 消息队列二期

消息队列二期考虑实现重平衡。