**MQ统一消息发布、订阅接口设计**

**一、原始需求描述**

**抽象消息订阅和监听接口，提供统一的sdk下沉到消息中心中。各业务方引用sdk进行统一的消息订阅监听。**

**需要支持以下特性：**

1. 统一接口：对上层提供一致的消息订阅 / 发布 API
2. 中间件隔离：底层实现与业务代码解耦
3. 自动适配：根据配置动态选择中间件实现
4. 扩展性：新增中间件只需添加新的适配器实现

**二、设计目标**

本项目旨在构建一个统一的消息队列消费者框架，用于简化和标准化不同消息队列系统的消息消费过程。该框架支持多种主流消息队列系统，包括Redis、RocketMQ、Kafka、RabbitMQ、ActiveMQ等，并提供统一的接口和可靠的消息处理机制。

· 提供统一的消息生产和消费接口

· 支持多种消息队列系统

· 实现可靠的消息处理机制（ack）

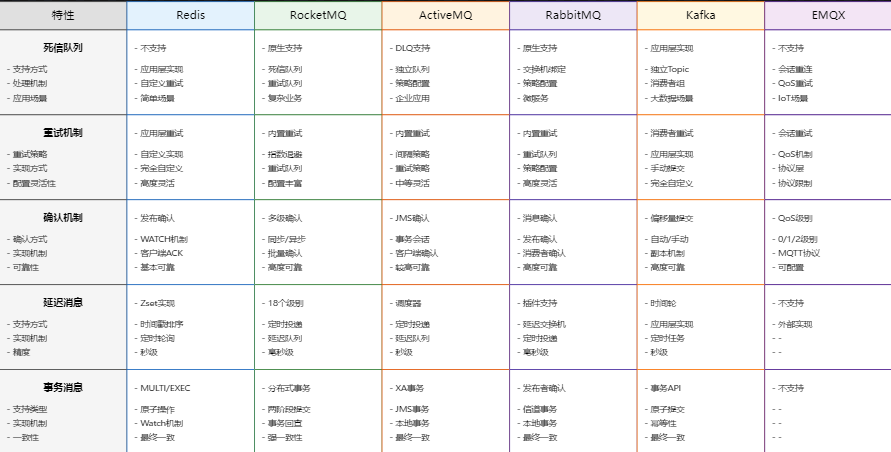
· 支持灵活的配置管理

~~· 提供完善的监控和异常处理~~

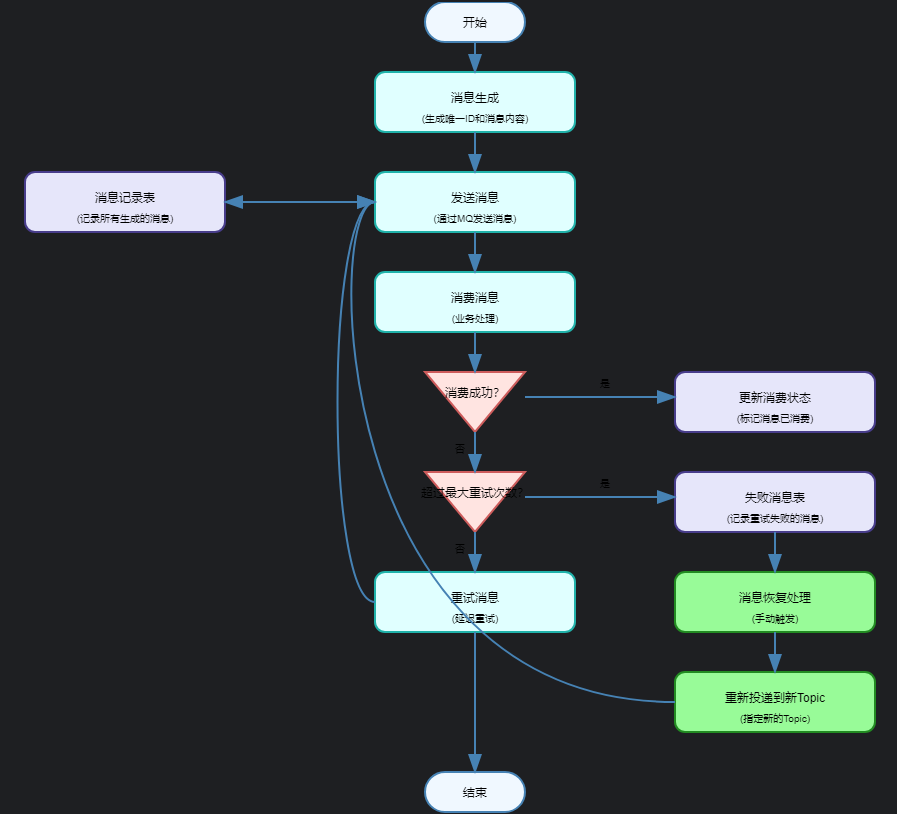
~~· 统一的信息管理界面~~

· 支持重试，死信队列持久化

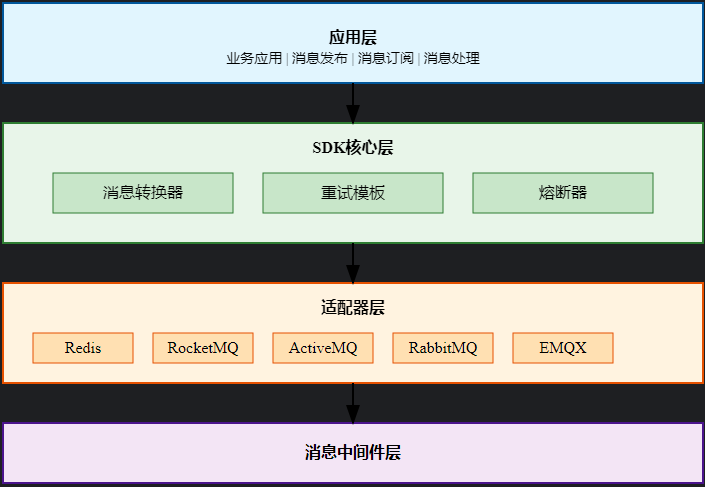
**三、各mq特性对比**



**四、消息生命周期**



**五、组件架构图**



**4.1sdk核心层**

**消息转换器（MessageConverter）**

职责：

* 处理不同消息中间件的消息格式转换
* 提供序列化和反序列化能力
* 支持多种消息格式（JSON、Protobuf等）
* 实现消息协议的适配转换

**重试模板（RetryTemplate）**

功能：

* 统一的重试策略管理
* 支持指数退避算法
* 可配置最大重试次数
* 重试间隔时间控制
* 异常处理机制

**熔断器（CircuitBreaker）**

核心特性：

* 故障检测：监控消息处理失败率
* 状态管理：关闭、半开、开启三种状态
* 阈值控制：可配置熔断触发阈值
* 自动恢复：支持自动恢复机制
* 降级处理：提供服务降级能力

**配置管理**

* MessageProperties ：
* 消息中间件连接配置
* 重试策略配置
* 熔断器参数配置
* 消息转换器配置
* MessageAutoConfiguration ：
* 自动装配核心组件
* 条件化配置加载
* 默认配置提供

**4.2 适配器层（Adapter Layer）**

**支持的 MQ 列表：**



**点击图片可查看完整电子表格**

**核心逻辑处理层**

**(1) 重试机制（Retry Mechanism）**

* 消费失败后自动重试（次数可配置）
* 支持指数退避策略（如 1s, 2s, 4s...）
* 记录重试次数、错误原因、时间戳

**(2) 死信队列（Dead Letter Queue）**

* 当达到最大重试次数仍未成功时，将消息写入死信队列
* 支持 DB 存储或转发到另一个 Topic
* 提供人工查看和重放能力

**(3) 延迟消息（Delay Message）**

* 若底层 MQ 不支持延迟，则使用 Redis ZSet 或数据库定时任务模拟
* 延迟级别由用户配置（如 1m, 5m, 10m）

**(4) 顺序消息（Ordered Message）**

* 按 key 分区（如 orderId）保证顺序
* 在消费者端串行处理同一分区消息

**(5) 幂等性（Idempotent）**

* 每条消息携带唯一 ID（msgId）
* 消费端记录已处理 msgId，避免重复消费

**(6) 事务消息（Transaction Message）**

* 本地事务状态表记录事务状态
* 依赖回调机制确认提交或回滚
* 看具体的qm之前情况，理论上可以借鉴rocketMq都实现事务消息

**4.2 数据库设计**

用于存储失败消息、重试记录、死信队列等信息。

**示例表结构：**

**表：message\_log（消息日志）**



**点击图片可查看完整电子表格**

**表：dead\_letter\_queue（死信队列）**

字段同上，仅用于归档失败消息

**表：retry\_history（重试历史）**



**点击图片可查看完整电子表格**

**五、配置**

**5.1 配置方式**

每种qm类型一个配置

通过 application.yml 或 JSON 文件配置使用的 MQ 类型及参数：

|  |
| --- |
| Plain Text mq:  type: rocketmq  config:  namesrvAddr: 127.0.0.1:9876  producerGroup: MY\_PRODUCER\_GROUP  consumerGroup: MY\_CONSUMER\_GROUP  maxRetry: 3  delayLevels: "1s 5s 10s" |

**六、接口设计**

**1.发布接口**

|  |
| --- |
| Plain Text */\*\**  *\* MQ生产者接口*  *\*/* public interface MQProducer {  void syncSend(String topic, String tag, MQEvent event);   void asyncSend(String topic, String tag, MQEvent event);  /\*\*  \*异步支持延迟  \*/  void asyncSend(String topic, String tag, MQEvent event, Integer delaySeconds); } |

发布接口使用实例

|  |
| --- |
| JavaScript // 订单信息 JSONObject orderInfo = new JSONObject(); shopInfo.put("id", "1001"); shopInfo.put("name", "测试店铺"); shopInfo.put("address", "北京市海淀区"); shopInfo.put("phone", "13800138000");  // 创建订单同步事件 OrderSyncEvent event = OrderSyncEvent.*builder*()  .orderId("1001")  .orderNum("123")  .shopInfo(orderInfo)  .build();  // 设置租户ID event.setTenantId("tenant\_001"); // 设置消息ID String messageId = "msg\_" + System.*currentTimeMillis*(); event.setMessageId(messageId);  // 同步发布事件 mqProducer.syncSend(event.getTopic(), event.getTag(), event); |

**2.订阅接口**

|  |
| --- |
| TypeScript */\*\**  *\* MQ消费者接口*  *\*/* public interface MQConsumer {   */\*\**  *\* 订阅消息*  *\**  *\* @param topic 主题*  *\* @param tag 标签*  *\* @param handler 消息处理器*  *\*/* void subscribe(String topic, String tag, Consumer<String> handler);   */\*\**  *\* 取消订阅*  *\**  *\* @param topic 主题*  *\* @param tag 标签*  *\*/* void unsubscribe(String topic, String tag);   */\*\**  *\* 启动消费者*  *\*/* void start();   */\*\**  *\* 停止消费者*  *\*/* void stop(); } |

订阅示例

|  |
| --- |
| TypeScript @Slf4j @Component public class OrderSyncConsumer {   @Autowired  private MQConsumer mqConsumer;   @PostConstruct  public void init() {  mqConsumer.subscribe("mall", "order\_sync", message -> {  try {  OrderSyncEvent event = JSON.*parseObject*(message, OrderSyncEvent.class);  *log*.info("收到订单同步消息：shopId={}, shopName={}",   event.getShopId(), event.getShopName());  // 处理订单同步逻辑  handleOrderSync(event);  } catch (Exception e) {  *log*.error("处理订单同步消息失败：message={}", message, e);  }  });    // 启动消费者  mqConsumer.start();  }   private void handleOrderSync(ShopSyncEvent event) {  // 实现店铺同步的具体业务逻辑  *log*.info("订单同步处理完成：oerderId={}", event.getOrderId());  } } |

具体事件

|  |
| --- |
| TypeScript public class OrderSyncEvent extends MQEvent {   */\*\**  *\* 事件类型，增删改等*  *\*/*  private Enum EventEnum    */\*\**  *\* 订单ID*  *\*/* private String orderId;    */\*\**  *\* 订单号*  *\*/* private String oederNum;    */\*\**  *\* 消息体*  *\*/* private JSONObject data;   @Override  public String getTopic() {  return "mall";  }   @Override  public String getTag() {  return "oerder\_sync";  } } |

**七、具体实现样例**

redis

|  |
| --- |
| TypeScript @Slf4j @Component @ConditionalOnProperty(name = "mq.type", havingValue = "redis") public class RedisProducerImpl implements MQProducer {   @Autowired  private RedisTemplate<String, String> redisTemplate;   @Override  public void syncSend(String topic, String tag, MQEvent event) {  try {  String channel = buildChannel(topic, tag);  redisTemplate.convertAndSend(channel, JSON.*toJSONString*(event));  *log*.info("Redis同步发送消息成功：channel={}, event={}", channel, event);  } catch (Exception e) {  *log*.error("Redis同步发送消息失败：topic={}, tag={}, event={}", topic, tag, event, e);  throw e;  }  }   @Override  public void asyncSend(String topic, String tag, MQEvent event) {  // Redis的发布订阅模式本身就是异步的，所以直接调用同步方法  syncSend(topic, tag, event);  }   */\*\**  *\* 构建Redis channel*  *\*/* private String buildChannel(String topic, String tag) {  return tag != null && !tag.isEmpty() ? topic + ":" + tag : topic;  } } |

Rabbitmq

|  |
| --- |
| TypeScript @Slf4j @Component @ConditionalOnProperty(name = "mq.type", havingValue = "rabbitmq") public class RabbitMQProducerImpl implements MQProducer {   @Autowired  private RabbitTemplate rabbitTemplate;   @Override  public void syncSend(String topic, String tag, MQEvent event) {  try {  rabbitTemplate.convertAndSend(topic, tag, JSON.*toJSONString*(event));  *log*.info("RabbitMQ同步发送消息成功：topic={}, tag={}, event={}", topic, tag, event);  } catch (Exception e) {  *log*.error("RabbitMQ同步发送消息失败：topic={}, tag={}, event={}", topic, tag, event, e);  throw e;  }  }   @Override  public void asyncSend(String topic, String tag, MQEvent event) {  try {  rabbitTemplate.convertAndSend(topic, tag, JSON.*toJSONString*(event), message -> {  message.getMessageProperties().setCorrelationId(event.getMessageId());  return message;  });  *log*.info("RabbitMQ异步发送消息成功：topic={}, tag={}, event={}", topic, tag, event);  } catch (Exception e) {  *log*.error("RabbitMQ异步发送消息失败：topic={}, tag={}, event={}", topic, tag, event, e);  throw e;  }  } } |

**八、MQ初始化方式**

初始化所有的mq，开发者发送消息的时候需要指定mq（业务需要）

**九、消息监测**

· 消息接收数

· 处理成功数

· 处理失败数

· 重试次数

· 死信数量

· 处理时间

· 消息大小