消息队列 Pulsar 版

SDK 文档

腾讯云

文档目录

SDK 文档
SDK 概览

最近更新时间：2024-10-14 10:01:52

TDMQ Pulsar 版支持 TCP 协议（Pulsar 社区版）。以下是 TDMQ Pulsar 版所支持的多语言 SDK：

> **注意**
> 为了更好地和 Pulsar 开源社区统一，自2021年4月30日起，腾讯云版 SDK 停止功能更新，TDMQ Pulsar 版推荐您使用社区版本的 SDK。

协议类型	SDK 语言
TCP 协议（Pulsar 社区版）	Go SDK
	Java SDK
	C++ SDK
	Python SDK
	Node.js SDK

SDK 配置参数推荐

最近更新时间：2024-12-31 17:04:02

TDMQ Pulsar 为您推荐 SDK 常见的参数说明，以及建议配置。

生产者配置

参数	描述	默认	配置建议
topicName	发送消息主题。	无	必填，由用户自定义。
producerName	生产者名称。	集群名-随机字符	保持默认，不要设置。
sendTimeoutMs	发送超时。	30s	建议用户根据实际情况调整。通常建议用户将超时时间至少设置为 5s 以上（非独占集群建议设置 10s 以上），防止偶尔网络抖动影响发送成功率。
maxPendingMessages	最大消息发送队列长度。	1000	保持默认。
maxPendingMessagesAcrossPartitions	所有分区的最大消息发送队列长度。	50000	保持默认。
blockIfQueueFull	消息发送队列满了之后是否阻塞。	false	保持默认。
messageRoutingMode	消息路由模式。	pulsar.RoundRobinDistribution	保持默认。
hashingScheme	hash 模式。	HashingScheme.JavaStringHash	保持默认。
CompressionType	压缩类型，默认不压缩。	No compression	保持默认。
batchingEnabled	是否开启批量发送。	true	在线场景建议改为 `false`，关闭批量发送。 批量发送会导致延迟消息、消息过滤等功能失效。
batchingMaxPublishDelayMicros	批量发送最长等待时间。	TimeUnit.MILLISECONDS.toMicros(1)	如果采用批量发送，建议调整为 10ms 以上。该阈值太小，会造成 producer cpu 负载增加。
batchingMaxMessages	批量发送最大一批消息数量。	1000	保持默认。
batchingMaxBytes	批量发送最大一批消息长度。	128k	保持默认。

消费者配置

参数	描述	默认值	建议
topicNames	订阅主题名列表。	无	必填，由用户自定义。
topicsPattern	订阅主题模式匹配。	无	保持默认。
subscriptionName	订阅名。	无	必填，由用户自定义。

subscriptionType	订阅类型。	SubscriptionType.Exclusive	SubscriptionType.Shared，必填，建议生产优先选用 `Shared` 消费模式。
receiverQueueSize	接收队列长度。	1000	保持默认。
acknowledgementsGroupTimeMicros	ack 请求的批量等待时间，设置的过小，会导致集群整体性能下降。	100ms	保持默认。
negativeAckRedeliveryDelayMicros	调用 nack 接口，多久之后重推消息（如果 unack 之后，对应的consumer 实例重启或宕机，消息会被立即重推）。	1min	保持默认。
maxTotalReceiverQueueSizeAcrossPartitions	整个消息接收队列的长度。	50000	保持默认。
consumerName	消费者名字。	集群名-随机字符	保持默认，不要设置。
ackTimeoutMillis	ack 超时时间，超过该阈值后，消息会被重新消费。0 表示没有超时时间	0	保持默认。
priorityLevel	消费者优先级，服务端会优先推送消息到优先级高的消费者实例上。数字越小优先级越高，0 表示最高优先级。	0	保持默认。
properties	消费者配置，方便服务端查询展示。tag 和消息过滤通过这个字段设置过滤规则。	无	保持默认。
subscriptionInitialPosition	初始化订阅位置，消费者首次消费的位置。默认从最新位置开始消费，主题在开始消费前发送的消息不会被消费。	SubscriptionInitialPosition.Latest	保持默认。
patternAutoDiscoveryPeriod	自动感知分区数量变化间隔时间。内部定期获取主题的分区元数据，及时感知主题扩容。	1min	保持默认。
regexSubscriptionMode	主题订阅模式，有三种模式，订阅持久化主题、订阅非持久化主题、订阅所有类型主题，默认订阅持久化主题。	RegexSubscriptionMode.PersistentOnly	保持默认。
retryEnable	是否开启重试功能。	false	保持默认。按需选择，参考重试和死信功能。
deadLetterPolicy	重试和死信策略。	无	保持默认。按需选择，参考重试和死信功能。
autoUpdatePartitions	自动更新分区。	true	保持默认。
replicateSubscriptionState	开启夸集群同步后，是否同步消费进度。	false	保持默认。

TCP 协议（Pulsar 社区版）Spring Boot Starter 接入

最近更新时间：2025-04-25 16:44:22

操作场景

本文以 Spring Boot Starter 接入为例介绍实现消息收发的操作过程，帮助您更好地理解消息收发的完整过程。

前提条件

- 完成资源创建与准备
- 安装1.8或以上版本 JDK
- 安装2.5或以上版本 Maven
- 下载 Demo

操作步骤

步骤1：添加依赖

在项目中引入 Pulsar Starter 相关依赖。

```
<dependency>
    <groupId>io.github.majusko</groupId>
    <artifactId>pulsar-java-spring-boot-starter</artifactId>
    <version>1.0.7</version>
</dependency>
<!-- https://mvnrepository.com/artifact/io.projectreactor/reactor-core -->
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>3.4.11</version>
</dependency>
```

步骤2：准备配置

在配置文件 application.yml 中添加 Pulsar 相关配置信息。

```
server:
  port: 8081
pulsar:
  # 命名空间名称
  namespace: namespace_java
  # 服务接入地址
  service-url: http://pulsar-w7eognxxx.tdmq.ap-gz.public.tencenttdmq.com:8080
  # 授权角色密钥
  token-auth-value: eyJrZXlJZC......
  # 集群id
  tenant: pulsar-w7eognxxx
```

参数	说明
namespace	命名空间名称，在控制台 命名空间 管理页面中复制。

service-url	集群接入地址，可以在控制台 **集群管理** 页面查看并复制。 
token-auth-value	角色密钥，在 **角色管理** 页面复制密钥列复制。 
tenant	集群 ID，在控制台 **集群管理** 页面中获取。

步骤3：生产消息

在 ProducerConfiguration.java 中配置生产者

```
package com.tencent.cloud.tdmq.pulsar.config;

import io.github.majusko.pulsar.producer.ProducerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;


/**
 * 生产者相关配置
 * 1.topic要提前在控制台中完成创建
 * 2.消息类型需要实现Serializable接口
 * 3.如果一个topic不能绑定不同的数据类型
 */
@Configuration
public class ProducerConfiguration {

    @Bean
    public ProducerFactory producerFactory() {
        return new ProducerFactory()
                // topic1 生产者
                .addProducer("topic1")
                // topic2 生产者
                .addProducer("topic2");
```

说明：

Topic 名称需要填入完整路径，即“persistent://clusterid/namespace/topic”，clusterid/namespace/topic 的部分可以从控制台上 **Topic 管理** 页面直接复制。

编译并运行生产消息程序 MyProducer.java。

```
package com.tencent.cloud.tdmq.pulsar.service;

import io.github.majusko.pulsar.producer.PulsarTemplate;
import org.apache.pulsar.client.api.MessageId;
import org.apache.pulsar.client.api.PulsarClientException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.nio.charset.StandardCharsets;
import java.util.concurrent.CompletableFuture;

@Service
public class MyProducer {

    /**
     * 1.发送消息的topic是在生产者配置中已经声明的topic
     * 2.PulsarTemplate类型应于发送消息的类型一致
     * 3.发送消息到指定topic时，消息类型需要与生产者工厂配置中的topic绑定的消息类型对应
     */

    @Autowired
    private PulsarTemplate<byte[]> defaultProducer;

    public void syncSendMessage() throws PulsarClientException {
        defaultProducer.send("topic1", "Hello pulsar client.".getBytes(StandardCharsets.UTF_8));
    }

    public void asyncSendMessage() {
        String msg = "Hello pulsar client.";
        CompletableFuture<MessageId> completableFuture = defaultProducer.sendAsync("topic1",
msg.getBytes(StandardCharsets.UTF_8));
        // 通过异步回调得知消息发送成功与否
        completableFuture.whenComplete(((messageId, throwable) -> {
            if( null != throwable ) {
                System.out.println("delivery failed, value: " + msg );
                // 此处可以添加延时重试的逻辑
            } else {
                System.out.println("delivered msg " + messageId + ", value:" + msg);
            }
        }));
    }

    /**
     * 顺序消息需要使用顺序类型topic来完成，顺序类型的topic支撑全局顺序和局部顺序两种类型，根据实际情况选择合适
的类型即可
     */
    public void sendOrderMessage() throws PulsarClientException {
        for (int i = 0; i < 5; i++) {
            defaultProducer.send("topic2", ("Hello pulsar client, this is a order message" + i +
".").getBytes(StandardCharsets.UTF_8));
        }
    }
}
```

> **注意：**
> - 发送消息的 Topic 是在生产者配置中已经声明的 Topic。
> - PulsarTemplate 类型应与发送消息的类型一致。
> - 发送消息到指定 Topic 时，消息类型需要与生产者工厂配置中的 Topic 绑定的消息类型对应。

步骤4：消费消息

编译并运行消费消息程序 MyConsumer.java。

```
package com.tencent.cloud.tdmq.pulsar.service;

import io.github.majusko.pulsar.annotation.PulsarConsumer;
import io.github.majusko.pulsar.constant.Serialization;
import org.apache.pulsar.client.api.SubscriptionType;
import org.springframework.stereotype.Service;

/**
 * 消费者配置
 */
@Service
public class MyConsumer {

    @PulsarConsumer(topic = "topic1",  // 订阅topic名称
            subscriptionName = "sub_topic1", // 订阅名称
            serialization = Serialization.JSON, // 序列化方式
            subscriptionType = SubscriptionType.Shared, // 订阅模式，默认为独占模式
            consumerName = "firstTopicConsumer", // 消费者名称
            maxRedeliverCount = 3, // 最大重试次数
            deadLetterTopic = "sub_topic1-DLQ" // 死信topic名称
    )
    public void firstTopicConsume(byte[] msg) {
        // TODO process your message
        System.out.println("Received a new message. content: [" + new String(msg) + "]");
        // 如果消费失败，请抛出异常，这样消息会进入重试队列，之后可以重新消费，直到达到最大重试次数之后，进入死信
队列。前提是要创建重试和死信topic
    }


    /**
     * 顺序类型的消息可借助顺序类型的topic来完成，支持全局顺序和局部顺序两种类型
     */
    @PulsarConsumer(topic = "topic2", subscriptionName = "sub_topic2")
    public void orderTopicConsumer(byte[] msg) {
        // TODO process your message
        System.out.println("Received a order message. content: [" + new String(msg) + "]");
    }


    /**
     * 监听死信topic，处理死信消息
     */
```

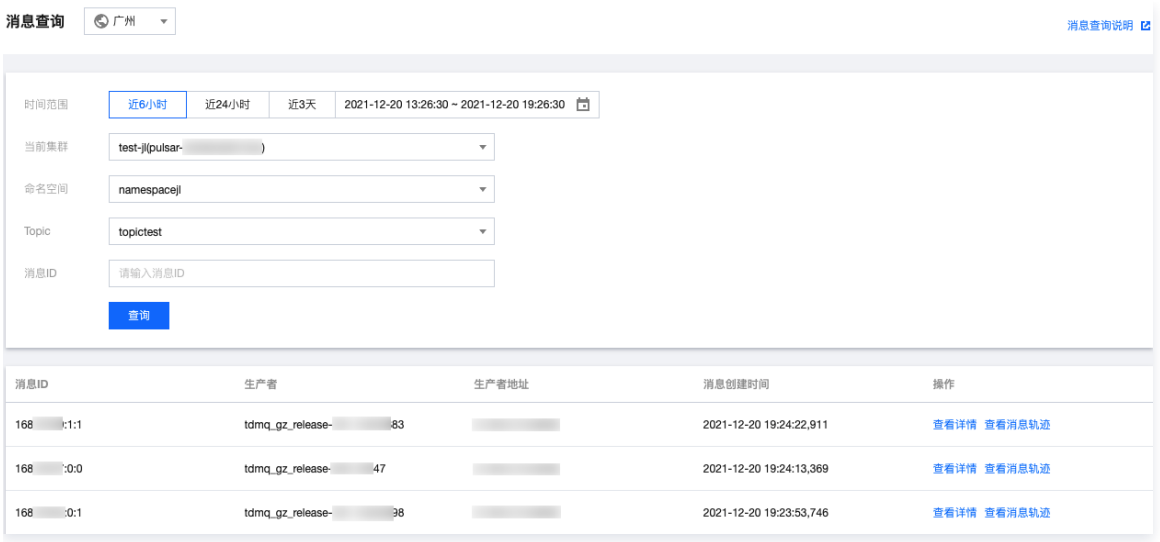
```
    @PulsarConsumer(topic = "sub_topic1-DLQ", subscriptionName = "dead_sub")
    public void deadTopicConsumer(byte[] msg) {
        // TODO process your message
        System.out.println("Received a dead message. content: [" + new String(msg) + "]");
    }
}
```

参数	说明
topic	Topic 名称需要填入完整路径，即 `persistent://clusterid/namespace/topic`，`clusterid/namespace/topic` 的部分可以从控制台上 Topic 管理 页面直接复制。 
subscriptionName	subscriptionName 需要写入订阅名，可在消费者界面复制获取。 
deadLetterTopic	若您新建订阅时，开启了**自动创建重试&死信队列**，系统会自动创建好一个重试队列和死信队列。此处可填写死信 Topic 名称， 需要填入完整路径，即 `persistent://clusterid/namespace/topic`，`clusterid/namespace/topic` 的部分可以从控制台上 Topic 管理页面直接复制。 

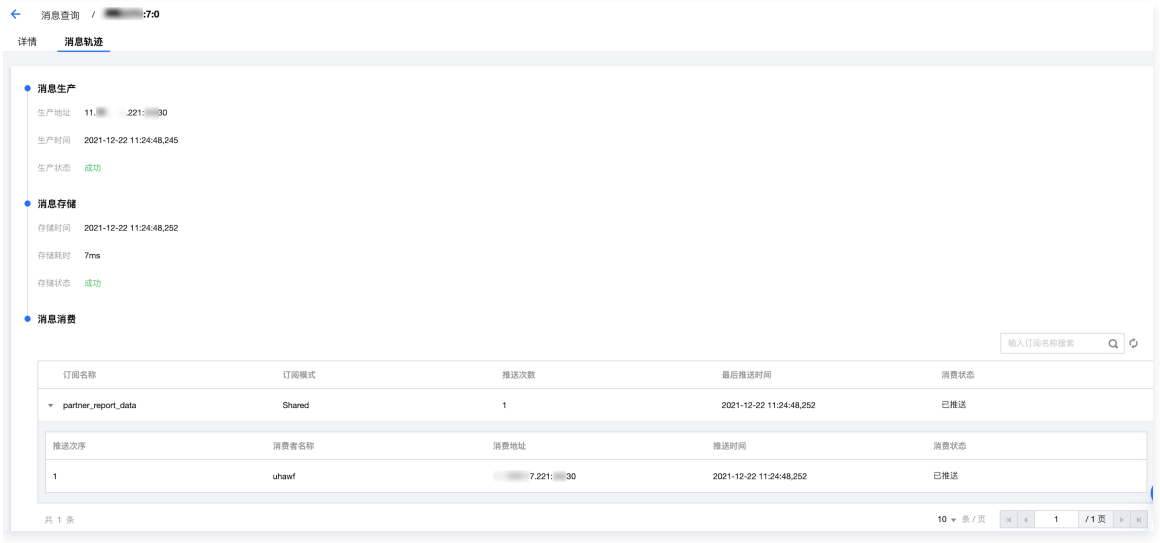
说明：
以上是基于 Springboot Starter 方式对 Pulsar 简单使用的配置。详细使用可参见 Demo 、 Starter Github、Starter Gitee。

步骤5：查询消息

登录控制台，进入 消息查询 页面，可查看 Demo 运行后的消息轨迹。



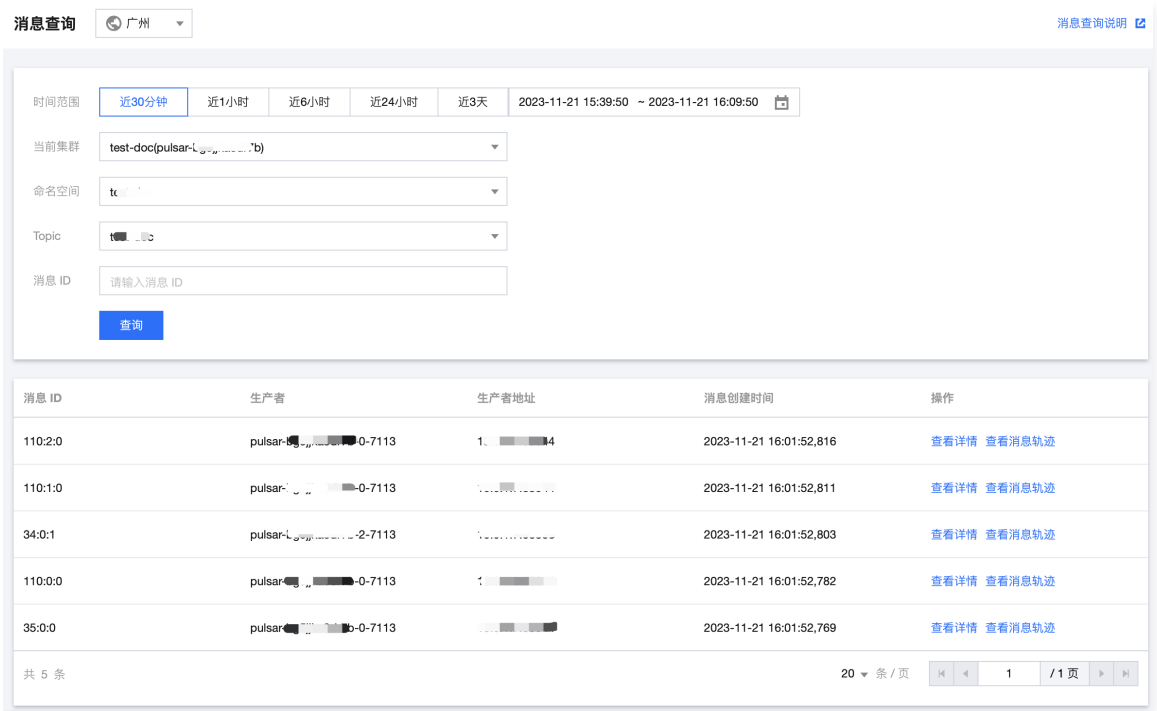
消息轨迹如下：



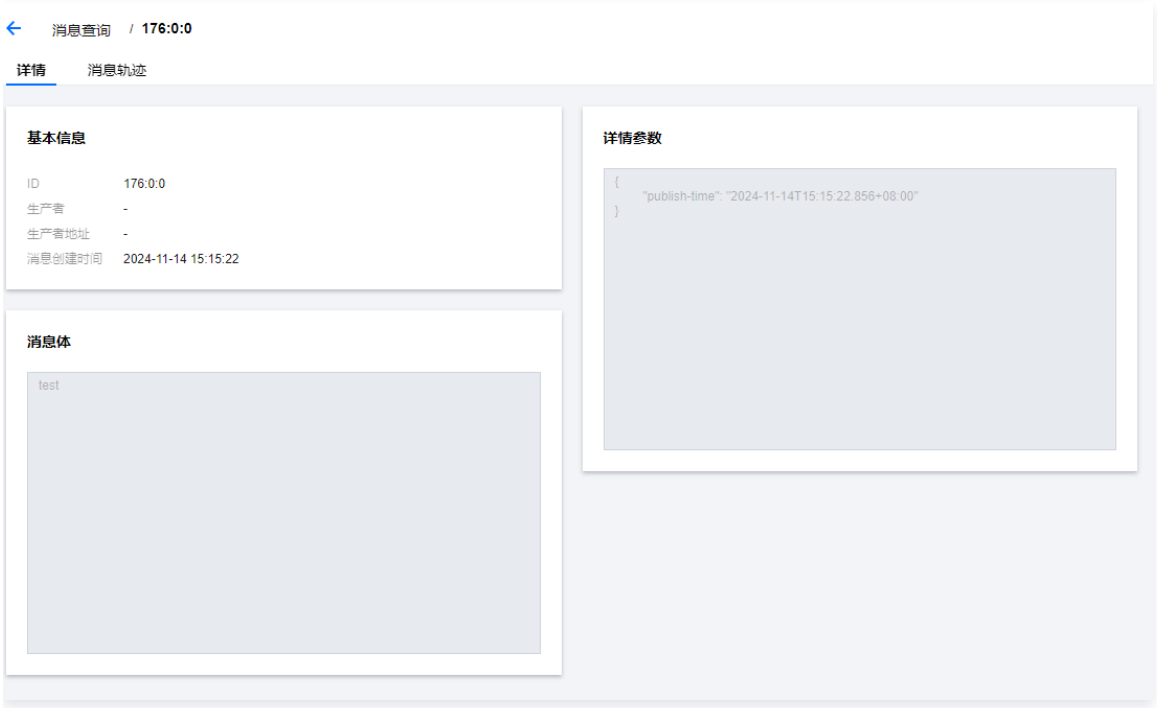
> **说明：**
> 以上是基于 Springboot Starter 方式对 Pulsar 简单使用的配置。详细使用可参见 Demo 或 Starter 文档。

步骤6：查看消费情况

进入 消息查询 页面，可查看消息详情。


消息查询
广州
消息查询说明
时间范围
近30分钟
近1小时
近6小时
近24小时
近3天
2023-11-21 15:39:50 ~ 2023-11-21 16:09:50
当前集群
命名空间
Topic
消息 ID
请输入消息 ID
查询
消息 ID
生产者
生产者地址
消息创建时间
操作
110:2:0
2023-11-21 16:01:52,816
查看详情 查看消息轨迹
110:1:0
2023-11-21 16:01:52,811
查看详情 查看消息轨迹
34:0:1
2023-11-21 16:01:52,803
查看详情 查看消息轨迹
110:0:0
2023-11-21 16:01:52,782
查看详情 查看消息轨迹
35:0:0
2023-11-21 16:01:52,769
查看详情 查看消息轨迹
共 5 条
20 条 / 页
1
/ 1 页

消息详情如下：


消息查询 / 176:0:0
详情
消息轨迹
基本信息
ID 176:0:0
生产者 -
生产者地址 -
消息创建时间 2024-11-14 15:15:22
详情参数
{
"publish-time": "2024-11-14T15:15:22.856+08:00"
}
消息体
test

Java SDK

最近更新时间：2025-06-27 11:23:41

操作场景

本文以调用 Java SDK 为例介绍通过开源 SDK 实现消息收发的操作过程，帮助您更好地理解消息收发的完整过程。

前提条件

- 完成资源创建与准备
- 安装1.8或以上版本 JDK
- 安装2.5或以上版本 Maven
- 下载 Demo

操作步骤

步骤1：安装 Java 依赖库

Java 项目中引入相关依赖，以 Maven 工程为例，在 pom.xml 添加以下依赖：

```
<dependency>
        <groupId>org.apache.pulsar</groupId>
        <artifactId>pulsar-client</artifactId>
        <version>3.0.8</version>
</dependency>
```

说明：

- 建议使用 3.0.8 及以上版本。
- 如果在客户端中使用批量收发消息功能（BatchReceive），则应使用 3.0.8 及以上版本的 SDK。

步骤2：修改配置参数

修改 Constant.java 参数。

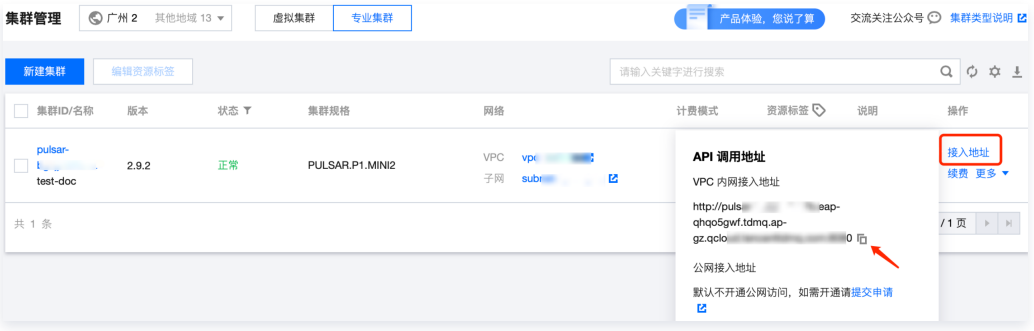
```
package com.tencent.cloud.tdmq.pulsar;

import org.apache.pulsar.client.api.AuthenticationFactory;
import org.apache.pulsar.client.api.PulsarClient;
import org.apache.pulsar.client.api.PulsarClientException;

public class Constant {
    /**
     * 服务接入地址，位于【集群管理】页面接入地址
     */
    private static final String SERVICE_URL = "http://pulsar-xxx.tdmq-pulsar.ap-
sh.public.tencenttdmq.com:8080";

    /**
     * 要使用的命名空间授权的角密钥，位于【角色管理】页面
     */
    private static final String AUTHENTICATION = "eyJrZXlJZC......";
```

```
    /**
     * 初始化pulsar客户端
     *
     * @return pulsar客户端
     */
    public static PulsarClient initPulsarClient() throws PulsarClientException {
        // 一个Pulsar client对应一个客户端链接
        // 原则上一个进程一个client，尽量避免重复创建，消耗资源
        // 关于客户端和生产消费者的实践教程，可以参考官方文档
https://cloud.tencent.com/document/product/1179/58090
        PulsarClient pulsarClient = PulsarClient.builder()
                // 服务接入地址
                .serviceUrl(SERVICE_URL)
                // 授权角色密钥
                .authentication(AuthenticationFactory.token(AUTHENTICATION)).build();
        System.out.println(">> pulsar client created.");
        return pulsarClient;
    }
}
```

参数	说明
SERVICE_URL	集群接入地址，可以在控制台集群管理页面查看并复制。 
AUTHENTICATION	角色的密钥，角色密钥可以在角色管理中复制。 

步骤3. 生产消息

创建并编译运行 SimpleProducer.java。

```
package com.tencent.cloud.tdmq.pulsar.simple;

import com.tencent.cloud.tdmq.pulsar.Constant;
import org.apache.pulsar.client.api.MessageId;
import org.apache.pulsar.client.api.Producer;
import org.apache.pulsar.client.api.PulsarClient;
import org.apache.pulsar.client.api.PulsarClientException;
```

```
import java.nio.charset.StandardCharsets;

/**
 * 同步发送消息
 */
public class SimpleProducer {

    public static void main(String[] args) throws PulsarClientException, InterruptedException {

        // 初始化pulsar客户端
        PulsarClient pulsarClient = Constant.initPulsarClient();
        // 构建生产者
        Producer<byte[]> producer = pulsarClient.newProducer()
                // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称
                .topic("persistent://pulsar-xxx/sdk_java/topic1").create();
        System.out.println(">> pulsar producer created.");
        for (int i = 0; i < 10; i++) {
            String value = "my-sync-message-" + i;
            // 发送消息
            MessageId msgId = producer.newMessage().key("key" +
i).value(value.getBytes(StandardCharsets.UTF_8)).send();
            System.out.println("deliver msg " + msgId + ",value:" + value);

            Thread.sleep(500);
        }
        // 关闭生产者
        producer.close();
        // 关闭客户端
        pulsarClient.close();
    }
}
```

topic：填写创建好的 topic 名称，需要填入完整路径，即 `persistent://clusterid/namespace/Topic` **，** `clusterid/namespace/topic` **的部分可以从控制台上 Topic 管理页面直接复制。**

步骤4：消费消息

创建并编译运行 SimpleConsumer.java。

```
package com.tencent.cloud.tdmq.pulsar.simple;

import com.tencent.cloud.tdmq.pulsar.Constant;
import org.apache.pulsar.client.api.Consumer;
import org.apache.pulsar.client.api.Message;
import org.apache.pulsar.client.api.MessageId;
import org.apache.pulsar.client.api.PulsarClient;
import org.apache.pulsar.client.api.PulsarClientException;
import org.apache.pulsar.client.api.SubscriptionInitialPosition;
```

```java
import org.apache.pulsar.client.api.SubscriptionType;

/**
 * 消费者
 */
public class SimpleConsumer {


    public static void main(String[] args) throws PulsarClientException {
        // 初始化pulsar客户端
        PulsarClient pulsarClient = Constant.initPulsarClient();
        // 构建消费者
        Consumer<byte[]> consumer = pulsarClient.newConsumer()
                // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管理】处复制
                .topic("persistent://pulsar-xxx/sdk_java/topic1")
                // 需要在控制台Topic详情页创建好一个订阅，此处填写订阅名
                .subscriptionName("sub1_topic1")
                // 声明消费模式为exclusive（独占）模式
                .subscriptionType(SubscriptionType.Exclusive)
                // 配置从最早开始消费，否则可能会消费不到历史消息
                .subscriptionInitialPosition(SubscriptionInitialPosition.Earliest).subscribe();
        System.out.println(">> pulsar consumer created.");
        for (int i = 0; i < 10; i++) {
            // 接收当前offset对应的一条消息
            Message<byte[]> msg = consumer.receive();
            MessageId msgId = msg.getMessageId();
            String value = new String(msg.getValue());
            System.out.println("receive msg " + msgId + ",value:" + value);
            // 接收到之后必须要ack，否则offset会一直停留在当前消息，导致消息积压
            consumer.acknowledge(msg);
        }
        // 关闭消费者
        consumer.close();
        // 关闭客户端
        pulsarClient.close();
    }
}
```

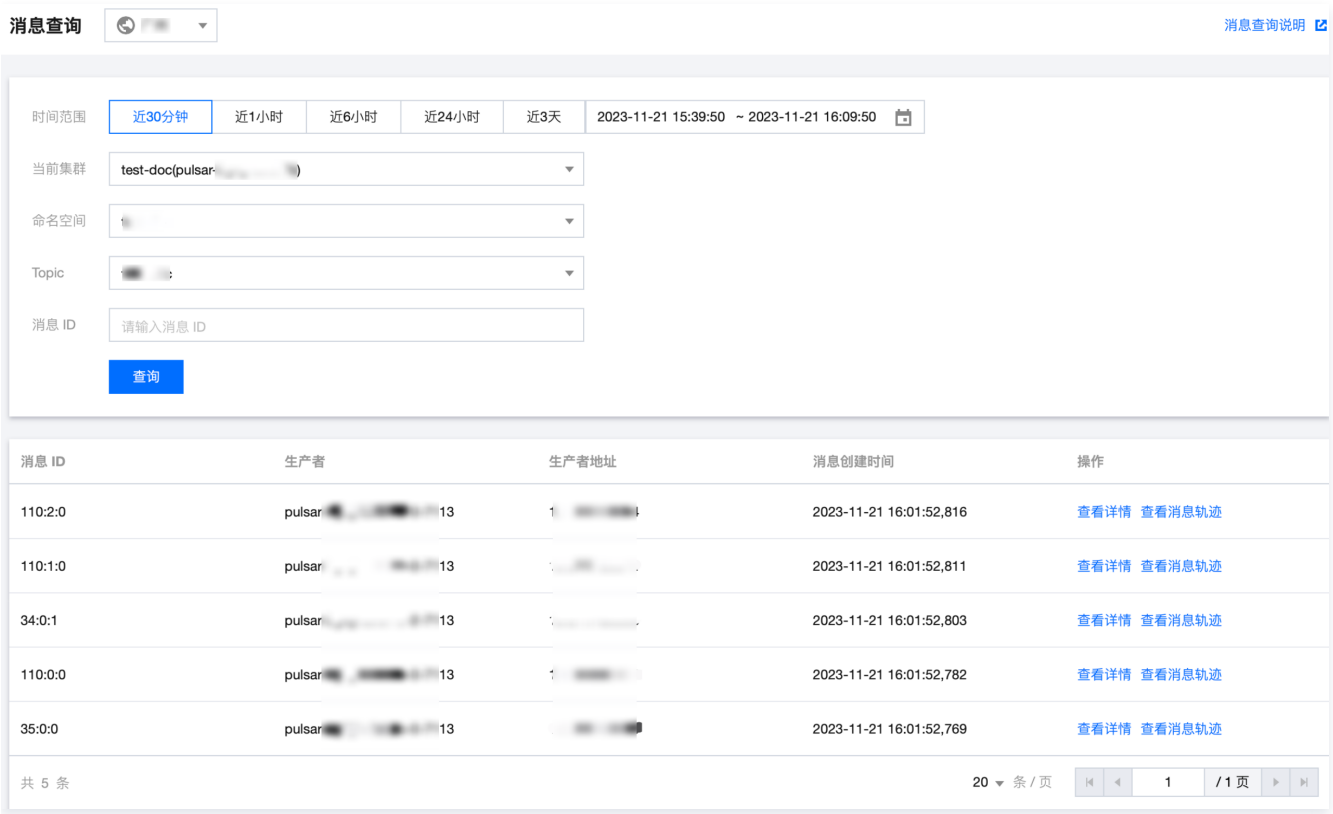
Topic 名称需要填入完整路径，即 `persistent://clusterid/namespace/Topic` ， `clusterid/namespace/topic` 的部分可以从控制台上 Topic 管理 页面直接复制。

subscriptionName 需要写入订阅名，可在消费者界面查看。

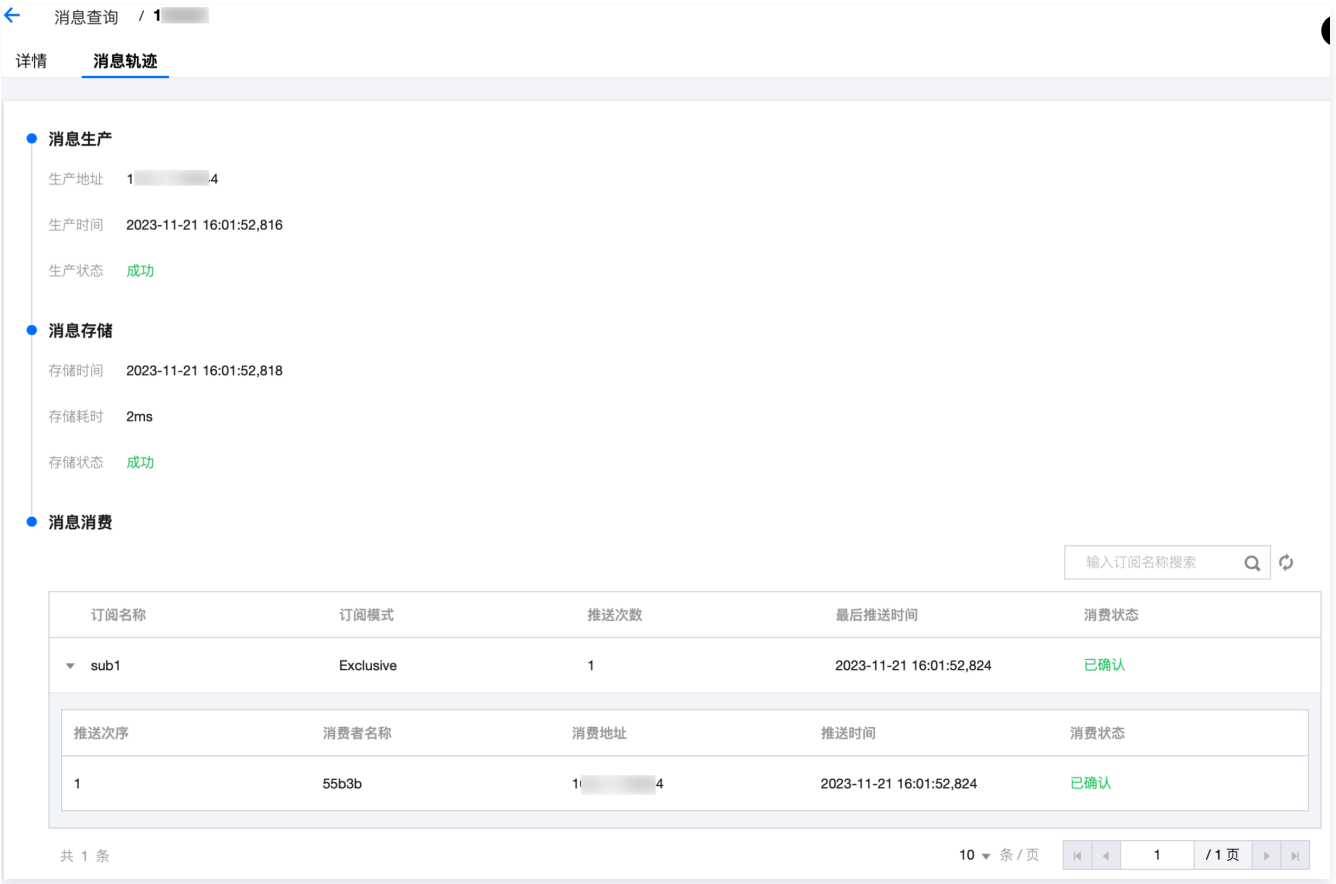
步骤5：查看消费情况

进入 消息查询 页面，可查看消息详情。

说明：
消息轨迹的查询只支持单条消息，如果用户在 Producer 侧开启了 Batch 功能，则在消息查询中，同一个 Batch 的消息只可以查询到 Batch 中的第一条消息。



消息轨迹如下：



说明：
上述是对消息的发布和订阅方式的简单介绍。更多操作可参见 Demo 或 Pulsar 官方文档。

SDK 版本相关

社区 issue 及优化点

Java SDK 2.7.2 及以下版本，强烈建议升级到 3.0.8 版本及更高版本，较高版本客户端已修复下列严重问题：

1. 修复 broker 重启导致 Consumer 消费暴涨问题（相关参考文档）。
2. 修复消费者因批量消息确认中的竞争条件而丢失消息确认问题（相关参考文档）。
3. 修复由于 IO 线程竞争条件导致生产者/消费者停止重新连接或发布/订阅的问题（相关参考文档）。
4. 修复重试队列和死信队列创建命名不规范问题（相关参考文档）。
5. 修复未 unack 后批量消息全部重推问题（相关参考文档）。

完整的社区 issue 参见：相关参考文档。

低版本风险隐患

Java SDK 2.7.x 及以下（包含 2.7.x）版本，对于极端场景异常处理覆盖不够全面，当 broker 升级重启或网络故障等场景下，有极小概率客户端和服务端重连过程出现异常，导致发送超时或者停止消费等问题。

这里强烈建议您先将客户端升级到 3.0.8 新版本后，再进行 broker 集群版本的更新。

除此之外，由于 2.7.x 和 2.9.x 以上版本死信队列和重试队列默认格式有变化，最好在业务代码中指定老的重试队列和死信队列的名称从而兼容，否则会出现无法消费到老重试队列/死信队列消息的情况，具体信息可参考 消息重试与死信机制。

低版本隐患处理手段

新版本在 broker 升级过程中能够正常重连，基本做到业务无感知。但如果您的客户端 SDK 确实无法升级到新版本，建议您在 broker 集群升级后，关注客户端的日志输出及控制台的生产消费相关指标。

如果出现生产消费卡住的情况，请及时重启客户端，通常客户端在重启后即可恢复正常，但可能会出现少量重复生产/消费消息的情况。如果重启客户端后还是无法改善，请及时 提交工单 进一步定位处理。

Go SDK

最近更新时间：2025-06-09 15:47:02

操作场景

本文以调用 Go SDK 为例介绍通过开源 SDK 实现消息收发的操作过程，帮助您更好地理解消息收发的完整过程。

前提条件

- 完成资源创建与准备
- 安装 Go
- 下载 Demo

> **说明：**
> 推荐使用 0.13.1 及以上版本。

操作步骤

1. 在客户端环境引入 `pulsar-client-go` 库。
   1.1 在客户端环境执行如下命令下载 Pulsar 客户端相关的依赖包。

```
go get -u "github.com/apache/pulsar-client-go/pulsar"
```

   1.2 安装完成后，即可通过以下代码引用到您的 Go 工程文件中。

```
import "github.com/apache/pulsar-client-go/pulsar"
```

2. 创建 Pulsar Client。

```
// 创建pulsar客户端
client, err := pulsar.NewClient(pulsar.ClientOptions{
    // 服务接入地址
    URL: serviceUrl,
    // 授权角色密钥
    Authentication:    pulsar.NewAuthenticationToken(authentication),
    OperationTimeout:  30 * time.Second,
    ConnectionTimeout: 30 * time.Second,
})

if err != nil {
    log.Fatalf("Could not instantiate Pulsar client: %v", err)
}

defer client.Close()
```

参数	说明

serviceUrl	集群接入地址，可以在控制台 **集群管理** 页面查看并复制。 
Authentication	角色密钥，在 **角色管理** 页面复制密钥列复制。 

3. 创建生产者。

```
// 使用客户端创建生产者
producer, err := client.CreateProducer(pulsar.ProducerOptions{
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称
    Topic: "persistent://pulsar-mmqwr5xx9n7g/sdk_go/topic1",
})

if err != nil {
    log.Fatal(err)
}
defer producer.Close()
```

> **说明**
> Topic 名称需要填入完整路径，即 `persistent://clusterid/namespace/Topic`，`clusterid/namespace/topic` 的部分可以从控制台上 **Topic管理** 页面直接复制。

4. 发送消息。

```
// 发送消息
_, err = producer.Send(context.Background(), &pulsar.ProducerMessage{
    // 消息内容
    Payload: []byte("hello go client, this is a message."),
    // 业务key
    Key: "yourKey",
    // 业务参数
    Properties: map[string]string{"key": "value"},
})
```

5. 创建消费者。

```
// 使用客户端创建消费者
consumer, err := client.Subscribe(pulsar.ConsumerOptions{
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称
    Topic:            "persistent://pulsar-mmqwr5xx9n7g/sdk_go/topic1",
    // 订阅名称
    SubscriptionName: "topic1_sub",
    // 订阅模式
```

```
    Type:              pulsar.Shared,
})
if err != nil {
    log.Fatal(err)
}
defer consumer.Close()
```

说明

- Topic 名称需要填入完整路径，即 `persistent://clusterid/namespace/Topic`，`clusterid/namespace/topic` 的部分可以从控制台上 Topic管理 页面直接复制。

- subscriptionName 需要写入订阅名，可在**消费管理**界面查看。

6. 消费消息。

```
// 获取消息
msg, err := consumer.Receive(context.Background())
if err != nil {
    log.Fatal(err)
}
// 模拟业务处理
fmt.Printf("Received message msgId: %#v -- content: '%s'\n",
            msg.ID(), string(msg.Payload()))

// 消费成功，回复ack，消费失败根据业务需要选择回复nack或ReconsumeLater
consumer.Ack(msg)
```

7. 登录 TDMQ Pulsar 版控制台，依次点击 **Topic 管理** > **Topic 名称**进入消费管理页面，点开订阅名下方右三角号，可查看生产消费记录。

说明

上述是对消息的发布和订阅方式的简单介绍。更多操作可参见 Demo 或 Pulsar 官方文档 。

自定义日志文件输出

使用场景

很多用户在使用 Pulsar Go SDK 时，未能自定义指定日志输出，Go SDK 默认将日志输出到了 os.Stderr 中去，具体如下：

```
// It's recommended to make this a global instance called `log`.
func New() *Logger {
    return &Logger{
        Out:          os.Stderr, // 默认输出
        Formatter:    new(TextFormatter),
        Hooks:        make(LevelHooks),
        Level:        InfoLevel,
        ExitFunc:     os.Exit,
        ReportCaller: false,
    }
}
```

由于日志信息的默认输出大都为 `os.Stderr`，如果用户没有自定义日志 lib 的话，Go SDK 的日志就会和业务日志混淆到一起，增加了问题定位的难度。

解决方案

Go SDK 在 Client 侧暴露了一个 logger 的接口，可以支持用户自定义自己的 log 输出的格式以及位置等功能，同时也支持使用 logrus 以及 zap 等不同的日志 lib，具体参数如下：

1. 自定义 log lib 实现 Pulsar Go SDK 提供的 log.Logger 的接口：

```
// ClientOptions is used to construct a Pulsar Client instance.
type ClientOptions struct {
    // Configure the logger used by the client.
    // By default, a wrapped logrus.StandardLogger will be used, namely,
    // log.NewLoggerWithLogrus(logrus.StandardLogger())
    // FIXME: use `logger` as internal field name instead of `log` as it's more idiomatic
    Logger log.Logger
}
```

所以用户在使用 Go SDK 时，可以通过自定义 `logger` 接口的形式，自定义 `log lib`，来达到将日志重定向到指定位置的目的。下面以 `logrus` 为例，自定义一个 `log lib`，将 Go SDK 的日志输出到指定文件：

```
package main

import (
    "fmt"
    "io"
    "os"

    "github.com/apache/pulsar-client-go/pulsar/log"
    "github.com/sirupsen/logrus"

)

// logrusWrapper implements Logger interface
// based on underlying logrus.FieldLogger
type logrusWrapper struct {
```

```
    l logrus.FieldLogger
}

// NewLoggerWithLogrus creates a new logger which wraps
// the given logrus.Logger
func NewLoggerWithLogrus(logger *logrus.Logger, outputPath string) log.Logger {
    writer1 := os.Stdout
    writer2, err := os.OpenFile(outputPath, os.O_WRONLY|os.O_CREATE, 0755)
    if err != nil {
        logrus.Error("create file log.txt failed: %v", err)
    }
    logger.SetOutput(io.MultiWriter(writer1, writer2))
    return &logrusWrapper{
        l: logger,
    }
}

func (l *logrusWrapper) SubLogger(fs log.Fields) log.Logger {
    return &logrusWrapper{
        l: l.l.WithFields(logrus.Fields(fs)),
    }
}

func (l *logrusWrapper) WithFields(fs log.Fields) log.Entry {
    return logrusEntry{
        e: l.l.WithFields(logrus.Fields(fs)),
    }
}

func (l *logrusWrapper) WithField(name string, value interface{}) log.Entry {
    return logrusEntry{
        e: l.l.WithField(name, value),
    }
}

func (l *logrusWrapper) WithError(err error) log.Entry {
    return logrusEntry{
        e: l.l.WithError(err),
    }
}

func (l *logrusWrapper) Debug(args ...interface{}) {
    l.l.Debug(args...)
}

func (l *logrusWrapper) Info(args ...interface{}) {
    l.l.Info(args...)
}

func (l *logrusWrapper) Warn(args ...interface{}) {
    l.l.Warn(args...)
}

func (l *logrusWrapper) Error(args ...interface{}) {
    l.l.Error(args...)
```

```
}

func (l *logrusWrapper) Debugf(format string, args ...interface{}) {
    l.l.Debugf(format, args...)
}

func (l *logrusWrapper) Infof(format string, args ...interface{}) {
    l.l.Infof(format, args...)
}

func (l *logrusWrapper) Warnf(format string, args ...interface{}) {
    l.l.Warnf(format, args...)
}

func (l *logrusWrapper) Errorf(format string, args ...interface{}) {
    l.l.Errorf(format, args...)
}

type logrusEntry struct {
    e logrus.FieldLogger
}

func (l logrusEntry) WithFields(fs log.Fields) log.Entry {
    return logrusEntry{
        e: l.e.WithFields(logrus.Fields(fs)),
    }
}

func (l logrusEntry) WithField(name string, value interface{}) log.Entry {
    return logrusEntry{
        e: l.e.WithField(name, value),
    }
}

func (l logrusEntry) Debug(args ...interface{}) {
    l.e.Debug(args...)
}

func (l logrusEntry) Info(args ...interface{}) {
    l.e.Info(args...)
}

func (l logrusEntry) Warn(args ...interface{}) {
    l.e.Warn(args...)
}

func (l logrusEntry) Error(args ...interface{}) {
    l.e.Error(args...)
}

func (l logrusEntry) Debugf(format string, args ...interface{}) {
    l.e.Debugf(format, args...)
}

func (l logrusEntry) Infof(format string, args ...interface{}) {
```

```
    l.e.Infof(format, args...)
}

func (l logrusEntry) Warnf(format string, args ...interface{}) {
    l.e.Warnf(format, args...)
}

func (l logrusEntry) Errorf(format string, args ...interface{}) {
    l.e.Errorf(format, args...)
}
```

2. 在创建 client 的时候，指定自定义的 log lib。

```
client, err := pulsar.NewClient(pulsar.ClientOptions{
    URL:    "pulsar://localhost:6650",
    Logger: NewLoggerWithLogrus(log.StandardLogger(), "test.log"),
})
```

通过上述 Demo 示例，即可将 Pulsar Go SDK 的日志文件重定向到了当前目录的 test.log 的文件中，用户可以根据自己的需要将日志文件重定向到指定的位置。

SDK 版本相关

社区 issue 及优化点

Go SDK 版本 v0.9.0 以下版本需要先升级到 v0.13.1 再升级 broker，v0.9 及以上版本没遇到问题情况下，可以不用升级客户端。
较高版本已修复下列严重问题：

1. 修复重连阻塞问题，会导致发送超时（相关参考文档）。
2. 修复连接泄露问题，会导致消费者暴涨（相关参考文档）。
3. 集群变更或者节点故障等会导致消费阻塞，unload 或者消费者重启可以恢复（相关参考文档）。
4. 连接重连的时候可能会触发发送超时（相关参考文档）。
5. 修复消息被discarded的时候情况下，availablePermits 泄露导致消费阻塞问题（相关参考文档）。
6. 修复某些异常情况下 availablePermitsCh 阻塞问题（相关参考文档）。
7. 修复部分异常情况下，同步发送阻塞的问题（相关参考文档）。
8. 支持 group ack 能力，较大提升客户端消费性能（相关参考文档）。
9. 修复回退时间失效问题，每次都按照 100ms 退避重试（相关参考文档）。
10. 修复非 batch 模式下，消息key设置失败的问题（相关参考文档）。
11. 修复发送失败的时候，将发送 pending 队列中的消息全部设置为失败（相关参考文档）。
12. 修复大量重复发送消息的 bug（解决延迟消息超过服务端限制或者客户端断连重连之后产生的 bug）（相关参考文档）。

完整的社区 issue 参见：相关参考文档。

低版本风险隐患

Go SDK v0.9.0 以下（不包含 v0.9.0 ）版本，对于极端场景异常处理覆盖不够全面，当 broker 升级重启或网络故障等场景下，有极小概率客户端和服务端重连过程出现异常，导致发送超时或者停止消费等问题。这里强烈建议您先将客户端升级到 v0.13.1 新版本后，再进行 broker 集群版本的更新。

低版本隐患处理手段

高版本的客户端在 broker 升级过程中能够正常重连，基本做到业务无感知。但如果您的客户端 SDK 确实无法升级到新版本，建议您在 broker 集群升级后，关注客户端的日志输出及控制台的生产消费相关指标。

如果出现生产消费卡住的情况，请及时重启客户端，理论上客户端在重启后可恢复正常。如果重启客户端后依旧无法改善，请及时 提交工单 进一步定位处理。

tRPC go pulsar SDK

最近更新时间：2025-07-01 11:53:52

操作场景

本文以调用 tRPC-Go Pulsar SDK 为例，介绍通过 tRPC 框架实现消息收发的操作过程，帮助您更好地理解使用插件在 TDMQ Pulsar 进行消息收发的完整过程。

插件介绍

插件优势

Pulsar 插件主要是将社区版本的 pulsar-go-sdk 封装成 tRPC 插件的形式，主要有以下优点：

1. 封装后调用 Pulsar 服务能更加符合 RPC 开发者的习惯；
2. 插件在执行生产消费逻辑时会添加消息 RPC 的相关元数据信息，这样能够使用 RPC 原生的可观测性轨迹链路，更有利于开发调试定位问题；
3. 使用 RPC 封装后的插件，能够复用 RPC 原生拦截器，配置，监控等功能，提高业务代码的开发效率。

实现逻辑

tRPC Pulsar 插件的主要实现逻辑为：

1. 声明新传输协议 Pulsar protocol；
2. 实现 Pulsar protocol 传输 clientTransport.RoundTrip 接口。在客户端使用 Pulsar 协议生产消息时，transport 中会在内部初始化 Pulsar producer 客户端，将 tRPC 消息解析成 Pulsar 消息的格式，并使用该 producer 生产消息到配置文件对应的 Pulsar 集群，随后将 Pulsar 集群返回的消息 id 封装成 tRPC 的消息返回结构；
3. 实现 Pulsar protocol 传输 serverTransport.ListenAndServe 接口。在 tRPC 初始化 NewServer 读取配置文件识别 service 的协议是 Pulsar 时，在启动的 transport ListenAndServe 就会使用配置文件中 Pulsar 相关配置创建 Pulsar consumer 对象。当用户调用 RegisterConsumerService 方法绑定 service handler 作为接收消息后的回调处理函数，并且执行 s.Serve 启动服务后，serverTransport 会不断调用 consumer.Receive() 方法，每拉取一条消息就新建一个 go 协程执行用户传入的 handler 回调函数。

从实现逻辑中可以推理出如下结论： tRPC Pulsar 消费者/生产者没有在服务发现配置里注册，而是作为当前 server 的一个 service 对当前客户端提供服务，因此一个 tRPC 客户端中配置的一个 Pulsar service 仅对应 Pulsar 订阅中的一个消费者/topic 的一个生产者。不同 tRPC 客户端之间生产/消费资源相互隔离，不会出现 service 并发抢占的情况。

前提条件

- 完成 资源创建与准备 。
- 安装 Go 运行环境，推荐 1.23 以上版本 Go 客户端。
- 参考 tRPC-database 中的 Pulsar example，在业务代码中引用 trpc-go/trpc-database/pulsar tRPC 插件。

> **说明：**
> tRPC-Pulsar 推荐使用 0.3.2 及以上版本。

典型案例

生产消息

生产消息到指定的 topic：

```
# trpc_go.yaml
client:
  service:
    - name: trpc.pulsar.producer.service
      target: pulsar://pulsar-address          # 注意生产者的 target 需要有前缀 pulsar://
```

```
      protocol: pulsar

plugins:
  log:
    default:
      - writer: console
        level: info
    pulsar_logger:
      - writer: console
        level: info
      - writer: file                          # pulsar 日志输出到文件 ./pulsar.log
        level: info
        writer_config:
          filename: ./pulsar.log
          max_age: 7                          # max expire days
          max_size: 100                       # size of local rolling file, unit MB
          max_backups: 10                     # max number of log files
  mq:
    pulsar:
      clients:
        - name: pulsar-address                                                    # 和
service 中的 target 对应
          url: http://pulsar-xxxxxxxx.eap-xxxxxxxx.tdmq.ap-xx.qcloud.tencenttdmq.com:8080  # 参
考 Go SDK，在控制台获取接入点地址
          topic: pulsar-xxxxxxxx/<namespaceName>/<topicName>                      # 参
考 Go SDK 在控制台获取获取 topic 名称
          logger_name: pulsar_logger
          auth_name: token
          auth_param: '{"token":"eyJrxxxxxx"}'                                    # 参
考 Go SDK 在控制台获取获取角色 token，注意角色 token 需要有生产权限
```

生产代码如下所示：

```
// Package main for example
package main

import (
    "context"
    "fmt"
    "strconv"
    "time"

    "git.code.oa.com/trpc-go/trpc-database/pulsar"
    "git.code.oa.com/trpc-go/trpc-go"
    "git.code.oa.com/trpc-go/trpc-go/codec"
    "git.code.oa.com/trpc-go/trpc-go/log"
    "git.code.oa.com/trpc-go/trpc-go/plugin"
    pulsargo "github.com/apache/pulsar-client-go/pulsar"
)

func main() {
    // 注意：plugin.Register 要在 trpc.NewServer 之前执行
    plugin.Register("pulsar_logger", log.DefaultLogFactory)
    trpc.NewServer()
```

```
    defer func() {
        //  注意：这里需要显式调用 pulsar.close() 方法，刷新所有缓冲区的消息投递到 broker，从而避免丢失消息
        if err := pulsar.Close(); err != nil {
            log.Errorf("close pulsar client producters err:%v", err)
        }
    }()

    proxy := pulsar.NewClientProxy("trpc.pulsar.producer.service")
    for i := 0; i < 3; i++ {
        ctx, _ := codec.WithCloneMessage(context.Background())
        msgIdByte, err := proxy.Produce(ctx, &pulsar.ProducerMessage{
            Payload: []byte("value" + strconv.Itoa(i)),
            Key:     strconv.Itoa(i),
        })
        msgId, _ := pulsargo.DeserializeMessageID(msgIdByte)
        if err != nil {
            panic(err)
        }
        fmt.Printf("messasgeId = %s, key = %d, payload = %s\n", msgId, i,
"value"+strconv.Itoa(i))
    }
}
```

消费消息（不重推场景）

从指定 topic 中消费消息。本示例主要针对消费者业务逻辑可以自行处理消息，不需要服务端进行重推的场景。

```
# trpc_go.yaml
server:
  service:
    - name: trpc.pulsar.consumer.service  # 如果需要同一个 server 创建多个消费者，这里 name 需要不同
      address: pulsar-address             # 与下方 mq.pulsar.servers.name 一致
      protocol: pulsar                    # 应用层协议，需要为 pulsar

plugins:
  log:
    default:
      - writer: console
        level: info
    pulsar_logger:
      - writer: console
        level: info
      - writer: file
        level: info
        writer_config:
          filename: ./pulsar.log          # pulsar 日志输出到文件 ./pulsar.log
          max_age: 7                          # max expire days
          max_size: 100                       # size of local rolling file, unit MB
          max_backups: 10                     # max number of log files
  mq:
    pulsar:
      servers:
        - name: pulsar-address
```

```
          url: http://pulsar-xxxxxxxx.eap-xxxxxxxx.tdmq.ap-xx.qcloud.tencenttdmq.com:8080     #
参考 Go SDK，在控制台获取接入点地址
          subscription_name: sub-test                                                        #
订阅名称
          subscription_type: shared
          queue_size: 100                                                                    #
通常配置为 500/当前订阅下消费者总数，例如有 5 个消费者，就配置为 100
          topic: pulsar-xxxxxxxx/<namespaceName>/<topicName>                                 #
参考 Go SDK，在控制台获取 topic 完整名称
          logger_name: pulsar_logger
          auth_name: token
          auth_param: '{"token":"eyJrxxxxxx"}'                                               #
参考 Go SDK，在控制台获取角色 token，注意角色 token 需要有消费权限
```

消费代码如下所示，注意在 handle 函数中需要业务代码捕获可能的异常，并且需要返回 nil。否则如果 handle 回调函数返回 err 时，会导致当前消息调用 unack 方法，在 60s 后重新推送给消费者。

```
package main

import (
    "context"
    "git.code.oa.com/trpc-go/trpc-database/pulsar"
    "git.code.oa.com/trpc-go/trpc-go"
    "git.code.oa.com/trpc-go/trpc-go/log"
    "git.code.oa.com/trpc-go/trpc-go/plugin"
    pulsargo "github.com/apache/pulsar-client-go/pulsar"
)

func main() {
    // 注意：plugin.Register 要在 trpc.NewServer 之前执行
    plugin.Register("pulsar_logger", log.DefaultLogFactory)
    s := trpc.NewServer()
    pulsar.RegisterConsumerService(s.Service("trpc.pulsar.consumer.service"), &consumer{}) // 需
要和配置文件中的 service 同名
    //  这里不需要显式调用 close 相关方法，
    //  在关闭当前 trpc server 时候，trpc 会执行 context cancel() 保证内部的 pulsar consumer 优雅退出
    if err := s.Serve(); err != nil {
        panic(err)
    }
}

type consumer struct{}
func (consumer) Handle(ctx context.Context, message *pulsar.Message) error {
    messageId, _ := pulsargo.DeserializeMessageID(message.MsgID)
    log.Infof("[key]: %s,\t[payload]: %s,\t[topic]: %s,\t[msgId]: %s",
        message.Key, string(message.PayLoad), message.Topic, messageId)
    return nil
}
```

消费消息（重推场景）

本示例主要针对消费者业务可能存在一些轮询或异步逻辑，对于当前消息无法马上进行处理，需要等待一段时间后重新消费的场景。

```
# trpc_go.yaml
server:
  service:
    - name: trpc.pulsar.consumer.service  # 如果需要同一个 server 创建多个消费者，这里 name 需要不同
      address: pulsar-address             # 与下方 mq.pulsar.servers.name 一致
      protocol: pulsar                    # 应用层协议，需要为 pulsar

plugins:
  log:
    default:
      - writer: console
        level: info
    pulsar_logger:
      - writer: console
        level: info
      - writer: file
        level: info
        writer_config:
          filename: ./pulsar.log          # pulsar 日志输出到文件 ./pulsar.log
          max_age: 7                        # max expire days
          max_size: 100                     # size of local rolling file, unit MB
          max_backups: 10                   # max number of log files
  mq:
    pulsar:
      servers:
        - name: pulsar-address
          url: http://pulsar-xxxxxxxx.eap-xxxxxxxx.tdmq.ap-xx.qcloud.tencenttdmq.com:8080    #
参考 Go SDK，在控制台获取接入点地址
          subscription_name: sub-test                                                        #
订阅名称
          subscription_type: shared
          queue_size: 100                                                                    #
通常配置为 500/当前订阅下消费者总数，例如有 5 个消费者，就配置为 100
          topic: pulsar-xxxxxxxx/<namespaceName>/<topicName>                                 #
参考 Go SDK，在控制台获取 topic 完整名称
          logger_name: pulsar_logger
          auth_name: token
          auth_param: '{"token":"eyJrxxxxxx"}'                                               #
参考 Go SDK，在控制台获取角色 token，注意由于涉及重推消息，角色 token 需要同时有生产和消费权限
          dlq_policy_rlq_topic: pulsar-xxxxxxxx/<namespaceName>/<topicName>                  #
参考 Go SDK，通常为 -RETRY 结尾，可参考
https://cloud.tencent.com/document/product/1179/49607#.E8.87.AA.E5.AE.9A.E4.B9.89.E5.8F.82.E6.95
.B0.E8.AE.BE.E7.BD.AE
          dlq_policy_topic: pulsar-xxxxxxxx/<namespaceName>/<topicName>                      #
参考 Go SDK，通常为 -DLQ 结尾，可参考
https://cloud.tencent.com/document/product/1179/49607#.E8.87.AA.E5.AE.9A.E4.B9.89.E5.8F.82.E6.95
.B0.E8.AE.BE.E7.BD.AE
          dlq_policy_max_deliveries: 10                                                      #
重试最大次数，超过最大次数后，会丢弃到死信队列，不再主动消费
          dlq_random_producer_name_enabled: true                                             #
兼容性考虑默认为 true
```

当前消费模式下共分为三个 topic：原始主题，重试主题，死信主题。

当前重推模式下，如果消息调用 reconsumeLater，消费者会 Ack 当前消息，并使用 reconsumeLater 中的延时时间，发送一条新的延迟消息到重试主题。如果客户端超过最大重试次数 dlq_policy_max_deliveries 配置下，还没有在 handle 函数中返回 nil 将消息 Ack，这条消息就会被发送到死信主题。默认情况下死信主题不会被订阅，需要用户手动处理。

```
// Package main for example
package main

import (
    "context"
    "time"

    "git.code.oa.com/trpc-go/trpc-database/pulsar"
    "git.code.oa.com/trpc-go/trpc-go"
    "git.code.oa.com/trpc-go/trpc-go/log"
    "git.code.oa.com/trpc-go/trpc-go/plugin"
    pulsargo "github.com/apache/pulsar-client-go/pulsar"
)

func main() {
    // 注意: plugin.Register 要在 trpc.NewServer 之前执行
    plugin.Register("pulsar_logger", log.DefaultLogFactory)
    s := trpc.NewServer()
    pulsar.RegisterConsumerService(s.Service("trpc.pulsar.consumer.service"), &consumer{})
    //  这里不需要显式调用 close 相关方法，
    //  在关闭当前 trpc server 时候，trpc 会执行 context cancel() 保证内部的 pulsar consumer 优雅退出
    if err := s.Serve(); err != nil {
        panic(err)
    }
}

type consumer struct{}
func (consumer) Handle(ctx context.Context, message *pulsar.Message) error {
    messageId, _ := pulsargo.DeserializeMessageID(message.MsgID)
    log.Infof("[key]: %s,\t[payload]: %s,\t[topic]: %s,\t[id]: %s,\t[reconsumeTimes]: %d",
message.Key, string(message.PayLoad), message.Topic, messageId, message.ReconsumeTimes)
    message.ReconsumeLater(1 * time.Minute) // 延迟时间最小值 1s，最大值 10 天，不过由于客户端和服务端时
间可能不一致，这里尽量配置延迟消息最长时间在 9 天之内
    // 如果不调用 ReconsumeLater，而是返回 nil，则当前消息被 Ack 并不再被重推
    return nil
}
```

消息 Id 转换

目前插件返回的消息格式为 byte 数组，需要将消息转换成可读的消息格式，即 (ledgerId: entryId: partitionIdx)，才能在控制台消息查询里查询到对应的消息轨迹，转换示例如下：

```
package main

import (
    "context"
    pulsargo "github.com/apache/pulsar-client-go/pulsar"  // 这里需要重命名，防止与 trpc pulsar 包名
冲突
)
```

```
func main() {
    s := "08e3bf6710da9d02180020003001"
    data, _ := hex.DecodeString(s)                          // data 为插件返回的 msgId byte[] 格式
    messageId, _ := pulsargo.DeserializeMessageID(data)
    log.Println(fmt.Sprintf("messageId = %s", messageId)) // 这里输出的结果即为控制台消息查询中的
msgId
}
```

SDK 版本相关

低版本风险隐患

tRPC Pulsar 0.1.3 及以下（changelog 中标注 deprecated）版本，对于极端场景异常处理覆盖不够全面，当 broker 升级重启或网络故障等场景下，有极小概率客户端和服务端重连过程出现异常，导致发送超时或者停止消费等问题。这里强烈建议您先将客户端升级到 0.3.2 新版本后，再进行 broker 集群版本的更新。

低版本隐患处理手段

高版本的客户端在 broker 升级过程中能够正常重连，基本做到业务无感知。但如果您的客户端 SDK 确实无法升级到新版本，建议您在 broker 集群升级后，关注客户端的日志输出及控制台的生产消费相关指标。
如果出现生产消费卡住的情况，请及时重启客户端，理论上客户端在重启后可恢复正常。如果重启客户端后依旧无法改善，请及时 提交工单 进一步定位处理。

tRPC go TDMQ SDK

最近更新时间：2025-07-01 11:53:52

操作场景

本文以调用 tRPC-Go TDMQ SDK 为例介绍通过 tRPC 框架实现消息收发的操作过程，帮助您更好地理解使用插件在 TDMQ Pulsar 进行消息收发的完整过程。

插件介绍

TDMQ 插件后续仅增加稳定性需求更新，基本不进行新特性迭代开发。同时 tRPC Go Pulsar 插件对于客户端的优雅退出会更完善，对于消息的重试逻辑上会更加灵活。

> **注意：**
> 如果是新增 tRPC Go 用户，建议优先选用 tRPC Go Pulsar 插件 。

插件优势

TDMQ 插件主要是将社区版本的 pulsar-go-sdk 封装成 tRPC 插件的形式，主要有以下优点：

1. 封装后调用 Pulsar 服务能更加符合 RPC 开发者的习惯。
2. 插件在执行生产消费逻辑时会添加消息 RPC 的相关元数据信息，这样能够使用 RPC 原生的可观测性轨迹链路，更有利于开发调试定位问题。
3. 使用 RPC 封装后的插件，能够复用 RPC 原生拦截器，配置，监控等功能，提高业务代码的开发效率。

实现逻辑

tRPC TDMQ 插件的主要实现逻辑为：

1. 声明新传输协议 TDMQ protocol。
2. 实现 TDMQ protocol 传输 `clientTransport.RoundTrip` 接口。在客户端使用 TDMQ 协议生产消息时，transport 中会在内部初始化 Pulsar producer 客户端，将 tRPC 消息解析成 Pulsar 消息的格式，并使用该 producer 生产消息到配置文件对应的 Pulsar 集群，随后将 Pulsar 集群返回的消息 id 封装成 tRPC 的消息返回结构。
3. 实现 TDMQ protocol 传输 `serverTransport.ListenAndServe` 接口。在 tRPC 初始化 NewServer 读取配置文件识别 service 的协议是 TDMQ 时，在启动的 transport ListenAndServe 就会使用配置文件中 TDMQ 相关配置创建 Pulsar consumer 对象。当用户调用 RegisterConsumerService 方法绑定 service handler 作为接收消息后的回调处理函数，并且执行 s.Serve 启动服务后，serverTransport 会不断调用 `consumer.Receive()` 方法，每拉取一条消息就新建一个 go 协程执行用户传入的 handler 回调函数。

从实现逻辑中可以推理出如下结论：
tRPC TDMQ 消费者/生产者没有在服务发现配置里注册，而是作为当前 server 的一个 service 对当前客户端提供服务，因此一个 tRPC 客户端中配置的一个 TDMQ service 仅对应 Pulsar 订阅中的一个消费者/topic 的一个生产者。不同 tRPC 客户端之间生产/消费资源相互隔离，不会出现 service 并发抢占的情况。

前提条件

- 完成资源创建与准备 。
- 安装 Go 运行环境，推荐 1.23 以上版本 Go 客户端。
- 参考 tRPC-database 中的 TDMQ example，在业务代码中引用 trpc-go/trpc-database/tdmq tRPC 插件。

> **说明：**
> tRPC-TDMQ 推荐使用 0.3.2 及以上版本。

典型案例

生产消息

生产消息到指定的 topic：

```
client:
  service:
    - name: trpc.tdmq.producer.service
      target: tdmq://tdmq-address              # 注意生产者的 target 需要有前缀 tdmq://

plugins:
  log:
    default:
      - writer: console
        level: info
    tdmq_logger:
      - writer: console
        level: info
      - writer: file                           # pulsar 日志输出到文件 ./tdmq.log
        level: info
        writer_config:
          filename: ./tdmq.log
          max_age: 7                           # max expire days
          max_size: 100                        # size of local rolling file, unit MB
          max_backups: 10                      # max number of log files
  mq:
    tdmq:
      clients:
        - name: tdmq-address
# 和 service 中的 target 对应
          url: http://pulsar-xxxxxxxx.eap-xxxxxxxx.tdmq.ap-xx.qcloud.tencenttdmq.com:8080
# 参考 Go SDK，在控制台获取接入点地址
          topic: pulsar-xxxxxxxx/<namespaceName>/<topicName>
# 参考 Go SDK 在控制台获取获取 topic 名称
          logger_name: tdmq_logger
          auth_name: token
          auth_params: >-
# 参考 Go SDK 在控制台获取获取角色 token，注意角色 token 需要有生产权限
            {"token":"eyJrxxxxxxx"}
```

生产代码如下所示：

```
// Package main comment.
package main

import (
    "context"
    "fmt"
    "strconv"

    "git.code.oa.com/trpc-go/trpc-database/tdmq"
    "git.code.oa.com/trpc-go/trpc-go"
    "git.code.oa.com/trpc-go/trpc-go/codec"
    "git.code.oa.com/trpc-go/trpc-go/log"
    "git.code.oa.com/trpc-go/trpc-go/plugin"
)
```

```
func main() {
    plugin.Register("tdmq_logger", log.DefaultLogFactory)
    trpc.NewServer()
    proxy := tdmq.NewClientProxy("trpc.tdmq.producer.service")
    for i := 0; i < 3; i++ {
        ctx, _ := codec.WithCloneMessage(context.Background())
        msgIdByte, err := proxy.Produce(ctx, &tdmq.ProducerMessage{
            Payload: []byte("value" + strconv.Itoa(i)),
            Key:     strconv.Itoa(i),
        })
        msgId, _ := tdmq.DeserializeMessageID(msgIdByte)
        if err != nil {
            panic(err)
        }
        fmt.Printf("messasgeId = %s, key = %d, payload = %s\n", msgId, i,
"value"+strconv.Itoa(i))
    }
}
```

消费消息（不重推场景）

从指定 topic 中消费消息。本示例主要针对消费者业务逻辑可以自行处理消息，不需要服务端进行重推的场景。

```
# trpc_go.yaml
server:
  service:
    - name: trpc.tdmq.consumer.service
      protocol: tdmq
      address: tdmq-address

plugins:                                    # 插件配置
  log:                                      # 日志配置
    default:                                # 默认日志的配置，可支持多输出
      - writer: console                     # 控制台标准输出 默认
        level: info                         # 标准输出日志的级别
    tdmq_logger:
      - writer: console                     # tdmq 日志输出到控制台
        level: info
      - writer: file                        # tdmq 日志输出到文件 ./tdmq.log
        level: info
        writer_config:
          filename: ./tdmq.log
          max_age: 7                        # max expire days
          max_size: 100                     # size of local rolling file, unit MB
          max_backups: 10                   # max number of log files
  mq:
    tdmq:
      servers:
        - name: tdmq-address                                                    # 和
service 中的 address 对应
          url: http://pulsar-xxxxxxxx.eap-xxxxxxxx.tdmq.ap-xx.qcloud.tencenttdmq.com:8080    # 参
考 Go SDK，在控制台获取接入点地址
```

```
          topic: pulsar-xxxxxxxx/<namespaceName>/<topicName>                    # 参
考 Go SDK，在控制台获取 topic 完整名称
          subscription_name: sub-test                                           # 订
阅名称
          subscription_type: shared
          queue_size: 100                                                       # 通
常配置为 500/当前订阅下消费者总数，如果有 5 个消费者，就配置为 100
          logger_name: tdmq_logger
          auth_name: token
          auth_params: '{"token":"eyJrxxxxxxx"}'                                # 参
考 Go SDK，在控制台获取角色 token，注意角色 token 需要有消费权限
```

消费代码如下所示，注意在 handle 函数中需要业务代码捕获可能的异常，并且需要返回 nil。否则如果 handle 回调函数返回 err 时，会导致当前消息调用 unack 方法，在 60s 后重新推送给消费者。

```go
// Package main comment.
package main

import (
    "context"

    "git.code.oa.com/trpc-go/trpc-go"
    "git.code.oa.com/trpc-go/trpc-go/log"
    "git.code.oa.com/trpc-go/trpc-go/plugin"
    "git.code.oa.com/trpc-go/trpc-database/tdmq"
)

func main() {
    // 注意：plugin.Register 要在 trpc.NewServer 之前执行
    plugin.Register("tdmq_logger", log.DefaultLogFactory)
    s := trpc.NewServer()
    tdmq.RegisterConsumerService(s.Service("trpc.tdmq.consumer.service"), &Consumer{})
    s.Serve()
}

type Consumer struct {
}

func (Consumer) Handle(ctx context.Context, message *tdmq.Message) error {
    messageId, _ := tdmq.DeserializeMessageID(message.MsgID)
    log.Infof("[key]: %s,\t[payload]: %s,\t[topic]: %s,\t[id]: %s",
        message.Key, string(message.PayLoad), message.Topic, messageId)
    // 返回 nil 才会确认消费成功
    return nil
}
```

消费消息（重推场景）

本示例主要针对消费者业务可能存在一些轮询或异步逻辑，对于当前消息无法马上进行处理，需要等待一段时间后重新消费的场景。

```
# trpc_go.yaml
server:
  service:
```

```
    - name: trpc.tdmq.consumer.service
      protocol: tdmq
      address: tdmq-address

plugins:                                    # 插件配置
  log:                                      # 日志配置
    default:                                # 默认日志的配置，可支持多输出
      - writer: console                     # 控制台标准输出 默认
        level: info                         # 标准输出日志的级别
    tdmq_logger:
      - writer: console                     # tdmq 日志输出到控制台
        level: info
      - writer: file                        # tdmq 日志输出到文件 ./tdmq.log
        level: info
        writer_config:
          filename: ./tdmq.log
          max_age: 7                        # max expire days
          max_size: 100                     # size of local rolling file, unit MB
          max_backups: 10                   # max number of log files
  mq:
    tdmq:
      servers:
        - name: tdmq-address                                                    # 和
service 中的 address 对应
          url: http://pulsar-xxxxxxxx.eap-xxxxxxxx.tdmq.ap-xx.qcloud.tencenttdmq.com:8080    # 参
考 Go SDK，在控制台获取接入点地址
          topic: pulsar-xxxxxxxx/<namespaceName>/<topicName>                    # 参
考 Go SDK，在控制台获取 topic 完整名称
          subscription_name: sub-test                                           # 订
阅名称
          subscription_type: shared
          enable_retry: 1
          dlq_policy_topic: pulsar-xxxxxxxx/<namespaceName>/<topicName>         # 参
考 Go SDK，通常为 -DLQ 结尾，可参考
https://cloud.tencent.com/document/product/1179/49607#.E8.87.AA.E5.AE.9A.E4.B9.89.E5.8F.82.E6.95
.B0.E8.AE.BE.E7.BD.AE
          dlq_policy_max_deliveries: 3                                          # 重
试最大次数，超过最大次数后，会丢弃到死信队列，不再主动消费
          dlq_random_producer_name_enabled: true
          nack_redelivery_delay: 1m                                             # 每
次重推的间隔时间，建议不要超过 10 min，防止出现过多的消息空洞
          queue_size: 100                                                       # 通
常配置为 500/当前订阅下消费者总数，如果有 5 个消费者，就配置为 100
          logger_name: tdmq_logger
          auth_name: token
          auth_params: '{"token":"eyJrxxxxxxx"}'                                # 参
考 Go SDK，在控制台获取角色 token，注意由于涉及重推消息，角色 token 需要同时有生产和消费权限
```

当前消费模式下共分为两个 topic：原始主题，死信主题。
当前重推模式下，如果 handle 返回 err，服务端会在 `nack_redelivery_delay` 后重新推送该消息，并且增加消息中的 `redeliverCount` 次数。当消息超过 `dlq_policy_max_deliveries` handle 函数还是返回 err 时，这条消息就会被发送到死信主题。默认情况下死信主题不会被订阅，需要用户手动处理。

```
// Package main comment.
package main

import (
    "context"
    "errors"

    "git.code.oa.com/trpc-go/trpc-go"
    "git.code.oa.com/trpc-go/trpc-go/log"
    "git.code.oa.com/trpc-go/trpc-go/plugin"
    "git.code.oa.com/trpc-go/trpc-database/tdmq"
)

func main() {
    // 注意：plugin.Register 要在 trpc.NewServer 之前执行
    plugin.Register("tdmq_logger", log.DefaultLogFactory)
    s := trpc.NewServer()
    tdmq.RegisterConsumerService(s.Service("trpc.tdmq.consumer.service"), &Consumer{})
    s.Serve()
}

type Consumer struct {
}

func (Consumer) Handle(ctx context.Context, message *tdmq.Message) error {
    messageId, _ := tdmq.DeserializeMessageID(message.MsgID)
    log.Infof("[key]: %s,\t[payload]: %s,\t[topic]: %s,\t[id]: %s,\t [redeliverCount]: %d",
        message.Key, string(message.PayLoad), message.Topic, messageId, message.RedeliveryCount)
    // 如果业务处理失败需要重推，返回 err；否则返回 nil
    return errors.New("mock handle err")
}
```

消息 Id 转换

目前插件返回的消息格式为 byte 数组，需要将消息转换成可读的消息格式，即 (ledgerId: entryId: partitionIdx)，才能在控制台消息查询里查询到对应的消息轨迹，转换示例如下：

```
package main

import (
    "context"
    "git.code.oa.com/trpc-go/trpc-database/tdmq"
)

func main() {
    s := "08e3bf6710da9d02180020003001"
    data, _ := hex.DecodeString(s)                          // data 为插件返回的 msgId byte[] 格式
    messageId, _ := tdmq.DeserializeMessageID(data)
    log.Println(fmt.Sprintf("messageId = %s", messageId)) // 这里输出的结果即为控制台消息查询中的
msgId
}
```

SDK 版本相关

低版本风险隐患

tRPC TDMQ 0.2.9 及以下（changelog 中标注 deprecated）版本，对于极端场景异常处理覆盖不够全面，当 broker 升级重启或网络故障等场景下，有极小概率客户端和服务端重连过程出现异常，导致发送超时或者停止消费等问题。这里强烈建议您先将客户端升级到 0.3.2 新版本后，再进行 broker 集群版本的更新。

低版本隐患处理手段

高版本的客户端在 broker 升级过程中能够正常重连，基本做到业务无感知。但如果您的客户端 SDK 确实无法升级到新版本，建议您在 broker 集群升级后，关注客户端的日志输出及控制台的生产消费相关指标。
如果出现生产消费卡住的情况，请及时重启客户端，理论上客户端在重启后可恢复正常。如果重启客户端后依旧无法改善，请及时 提交工单 进一步定位处理。

C++ SDK

最近更新时间：2024-10-14 16:43:54

操作场景

本文以调用 C++ SDK 为例介绍通过开源 SDK 实现消息收发的操作过程，帮助您更好地理解消息收发的完整过程。

前提条件

- 完成资源创建与准备
- 安装 GCC
- 下载 Demo

操作步骤

1. 准备环境。
   1.1 在客户端环境安装 Pulsar C++ client，安装过程可参考官方教程 Pulsar C++ client 。
   1.2 在项目中引入 Pulsar C++ client 相关头文件及动态库。
2. 创建客户端。

```
// 客户端配置信息
ClientConfiguration config;
// 设置授权角色密钥
AuthenticationPtr auth = pulsar::AuthToken::createWithToken(AUTHENTICATION);
config.setAuth(auth);
// 创建客户端
Client client(SERVICE_URL, config);
```

参数	说明
SERVICE_URL	集群接入地址，可以在控制台 集群管理 页面查看并复制。 
AUTHENTICATION	角色密钥，在 角色管理 页面复制密钥列复制。 

3. 创建生产者。

```
// 生产者配置
ProducerConfiguration producerConf;
producerConf.setBlockIfQueueFull(true);
producerConf.setSendTimeout(5000);
// 生产者
Producer producer;
```

```
// 创建生产者
Result result = client.createProducer(
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称
    "persistent://pulsar-xxx/sdk_cpp/topic1",
    producerConf,
    producer);
if (result != ResultOk) {
    std::cout << "Error creating producer: " << result << std::endl;
    return -1;
}
```

说明

Topic 名称需要填入完整路径，即 `persistent://clusterid/namespace/Topic`，`clusterid/namespace/topic` 的部分可以从控制台上 Topic管理 页面直接复制。

3. 发送消息。

```
// 消息内容
std::string content = "hello cpp client, this is a msg";
// 构建消息对象
Message msg = MessageBuilder().setContent(content)
    .setPartitionKey("mykey")  // 业务key
    .setProperty("x", "1")    // 设置消息参数
    .build();
// 发送消息
Result result = producer.send(msg);
if (result != ResultOk) {
    // 发送失败
    std::cout << "The message " << content << " could not be sent, received code: " <<
result << std::endl;
} else {
    // 发送成功
    std::cout << "The message " << content << " sent successfully" << std::endl;
}
```

4. 创建消费者。

```
// 消费者配置信息
ConsumerConfiguration consumerConfiguration;
consumerConfiguration.setSubscriptionInitialPosition(pulsar::InitialPositionEarliest);
// 消费者
Consumer consumer;
// 订阅topic
Result result = client.subscribe(
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称
    "persistent://pulsar-xxx/sdk_cpp/topic1",
    // 订阅名称
    "sub_topic1",
    consumerConfiguration,
    consumer);

if (result != ResultOk) {
    std::cout << "Failed to subscribe: " << result << std::endl;
```

```
    return -1;
}
```

说明

- Topic 名称需要填入完整路径，即 `persistent://clusterid/namespace/Topic`，`clusterid/namespace/topic` 的部分可以从控制台上 Topic管理 页面直接复制。

- subscriptionName 需要写入订阅名，可在**消费管理**界面查看。

5. 消费消息。

```
Message msg;
// 获取消息
consumer.receive(msg);
// 模拟业务
std::cout << "Received: " << msg << "  with payload '" << msg.getDataAsString() << "'" <<
std::endl;
// 回复ack
consumer.acknowledge(msg);
// 消费失败回复nack，消息将会重新投递
// consumer.negativeAcknowledge(msg);
```

6. 登录 TDMQ Pulsar 版控制台，依次点击 **Topic 管理** > **Topic 名称**进入消费管理页面，点开订阅名下方右三角号，可查看生产消费记录。

> **说明**
> 上述是对消息的发布和订阅方式的简单介绍。更多操作可参见 Demo 或 Pulsar 官方文档 。

Python SDK

最近更新时间：2024-10-14 16:27:54

操作场景

本文以调用 Python SDK 为例介绍通过开源 SDK 实现消息收发的操作过程，帮助您更好地理解消息收发的完整过程。

前提条件

- 完成资源创建与准备
- 安装 Python
- 安装 pip
- 下载 Demo

操作步骤

1. 准备环境。
   在客户端环境安装 pulsar-client 库，可以使用 pip 进行安装，也可以使用其他方式，参见 Pulsar Python client 。

```
pip install 'pulsar-client==3.1.0'
```

2. 创建客户端。

```
# 创建客户端
client = pulsar.Client(
    authentication=pulsar.AuthenticationToken(
        # 已授权角色密钥
        AUTHENTICATION),
    # 服务接入地址
    service_url=SERVICE_URL)
```

参数	说明
SERVICE_URL	集群接入地址，可以在控制台 集群管理 页面查看并复制。 
AUTHENTICATION	角色密钥，在 角色管理 页面复制密钥列复制。 

3. 创建生产者。

```
# 创建生产者
producer = client.create_producer(
    # topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管理】处复制
```

```
    topic='pulsar-xxx/sdk_python/topic1'
)
```

> **说明**
> Topic 名称需要填入完整路径，即 `persistent://clusterid/namespace/Topic`，`clusterid/namespace/topic` 的部分可以从控制台上 Topic管理 页面直接复制。

4. 发送消息。

```
# 发送消息
producer.send(
    # 消息内容
    'Hello python client, this is a msg.'.encode('utf-8'),
    # 消息参数
    properties={'k': 'v'},
    # 业务key
    partition_key='yourKey'
)
```

还可以使用异步方式发送消息。

```
# 异步发送回调
def send_callback(send_result, msg_id):
    print('Message published: result:{}  msg_id:{}'.format(send_result, msg_id))

# 发送消息
producer.send_async(
    # 消息内容
    'Hello python client, this is a async msg.'.encode('utf-8'),
    # 异步回调
    callback=send_callback,
    # 消息配置
    properties={'k': 'v'},
    # 业务key
    partition_key='yourKey'
)
```

5. 创建消费者。

```
# 订阅消息
consumer = client.subscribe(
    # topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管理】处复制
    topic='pulsar-xxx/sdk_python/topic1',
    # 订阅名称
    subscription_name='sub_topic1'
)
```

> **说明**
> - Topic 名称需要填入完整路径，即 `persistent://clusterid/namespace/Topic`，`clusterid/namespace/topic` 的部分可以从控制台上 Topic管理 页面直接复制。

- subscriptionName 需要写入订阅名，可在**消费管理**界面查看。

6. 消费消息。

```
# 获取消息
msg = consumer.receive()
try:
    # 模拟业务
    print("Received message '{}' id='{}'".format(msg.data(), msg.message_id()))
    # 消费成功，回复ack
    consumer.acknowledge(msg)
except:
    # 消费失败，消息将会重新投递
    consumer.negative_acknowledge(msg)
```

7. 登录 TDMQ Pulsar 版控制台，依次点击 **Topic 管理** > **Topic 名称**进入消费管理页面，点开订阅名下方右三角号，可查看生产消费记录。

说明

上述是对消息的发布和订阅方式的简单介绍。更多操作可参见 Demo 或 Pulsar 官方文档 。

Node.js SDK

最近更新时间：2025-01-22 11:35:52

操作场景

TDMQ Pulsar 版2.7.1及以上版本的集群已支持 Pulsar 社区版 Node.js SDK。本文介绍如何使用 Pulsar 社区版 Node.js SDK 完成接入。

前提条件

- 获取接点地址
  在 TDMQ Pulsar 版控制台 集群管理 页面复制接入地址。
- 获取密钥
  已参考 角色与鉴权 文档配置好了角色与权限，并获取到了对应角色的密钥（Token）。

操作步骤

1. 按照 Pulsar 官方文档 在您客户端所在的环境中安装 Node.js Client。

```
$ npm install pulsar-client
```

2. 在创建 Node.js Client 的代码中，配置准备好的接入地址和密钥。

```
const Pulsar = require("pulsar-client");

(async () => {
  const client = new Pulsar.Client({
 serviceUrl: "http://*",    //更换为接入地址（控制台集群管理页完整复制）
 authentication: new Pulsar.AuthenticationToken({
   token: "eyJh**",        //更换为密钥
 }),
  });
  await client.close();
})();
```

说明：
Node.js SDK 在不同的版本下，对于环境的依赖条件不同。建议您先确定 Pulsar 引擎版本，而后根据 具体文档指引 查看使用姿势。
关于 Pulsar 社区版 Node.js SDK 各种功能的使用方式，请参见 Pulsar 官方文档 。