消息队列 Pulsar 版

SDK 文档

产品文档

腾讯云

文档目录

SDK 文档

SDK 概览

最近更新时间：2024-01-03 14:27:38

TDMQ Pulsar 版现支持 TCP 协议（Pulsar 社区版）和 HTTP 协议。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
HTTP 协议	Go SDK
	Java SDK
	C++ SDK
	Python SDK
	PHP SDK

TCP 协议（Pulsar 社区版）Spring Boot Starter 接入

最近更新时间：2024-01-03 14:27:38

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。

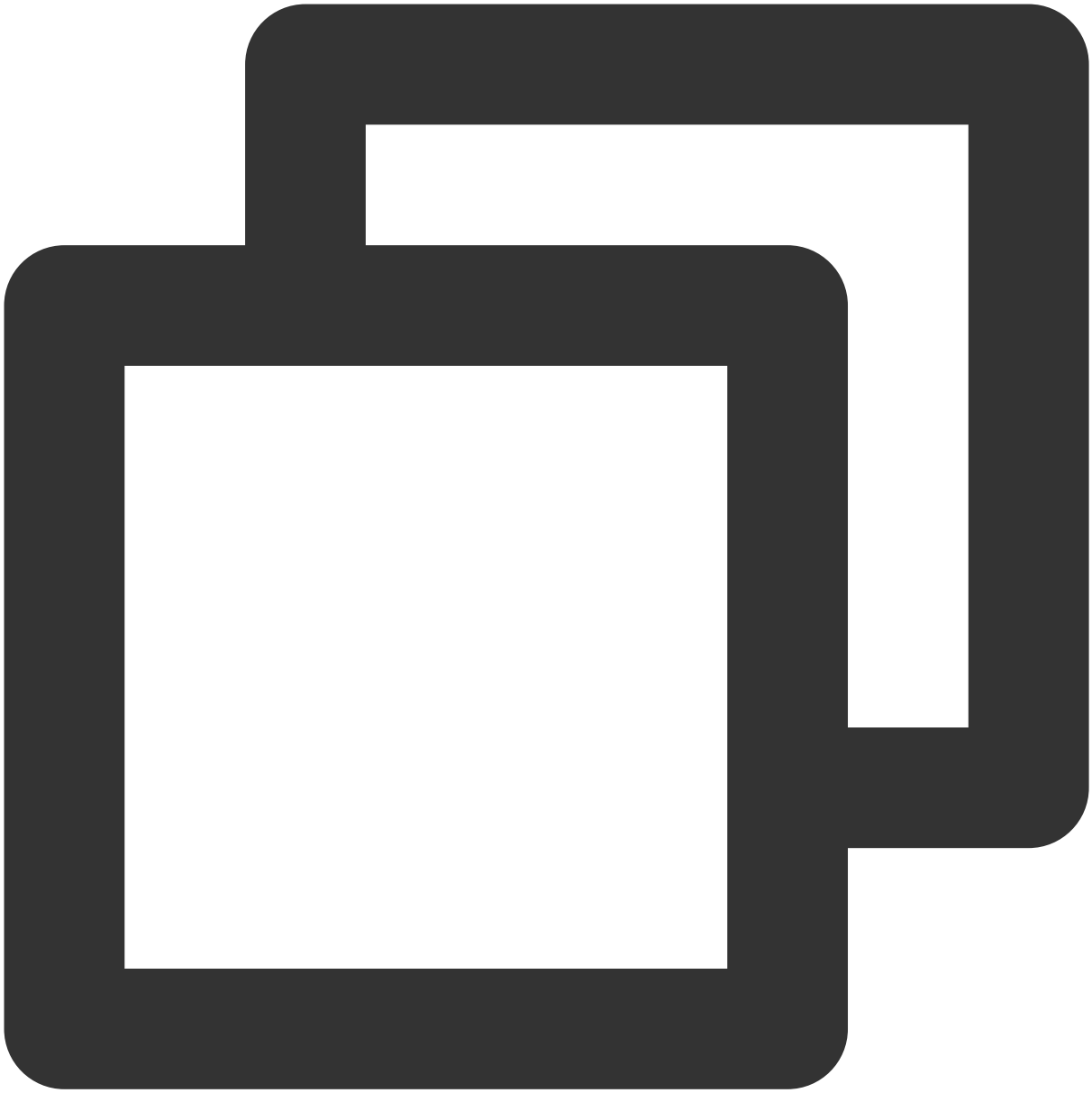
前提条件

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

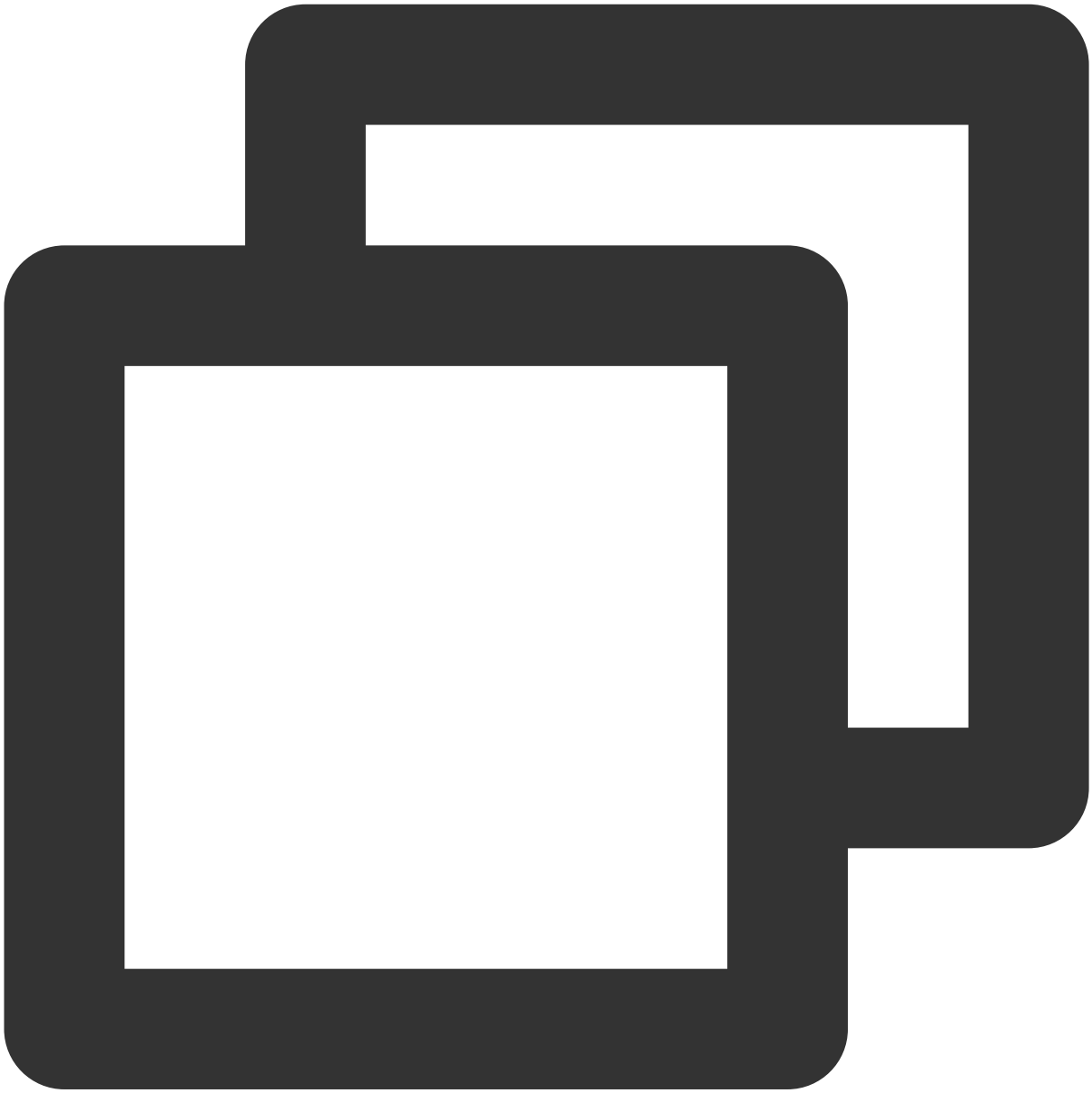
在项目中引入 Pulsar Starter 相关依赖。



```
<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>
```

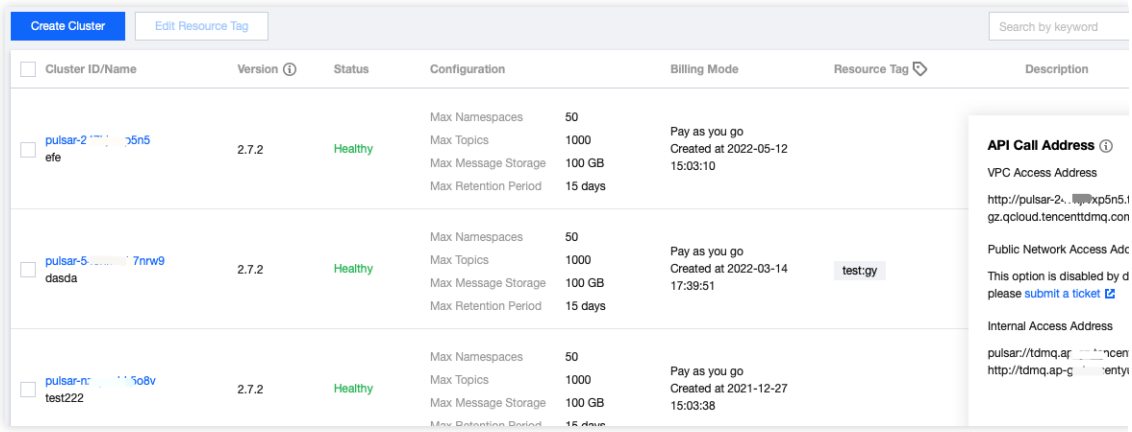
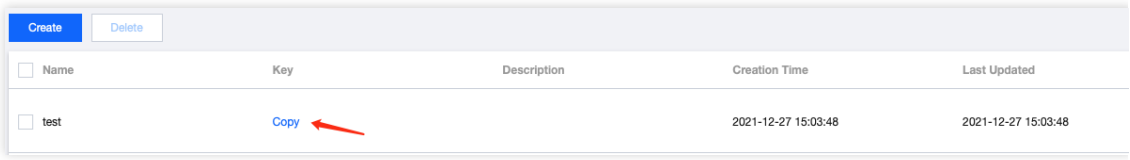
步骤2：准备配置

在配置文件中 添加 Pulsar 相关配置信息。



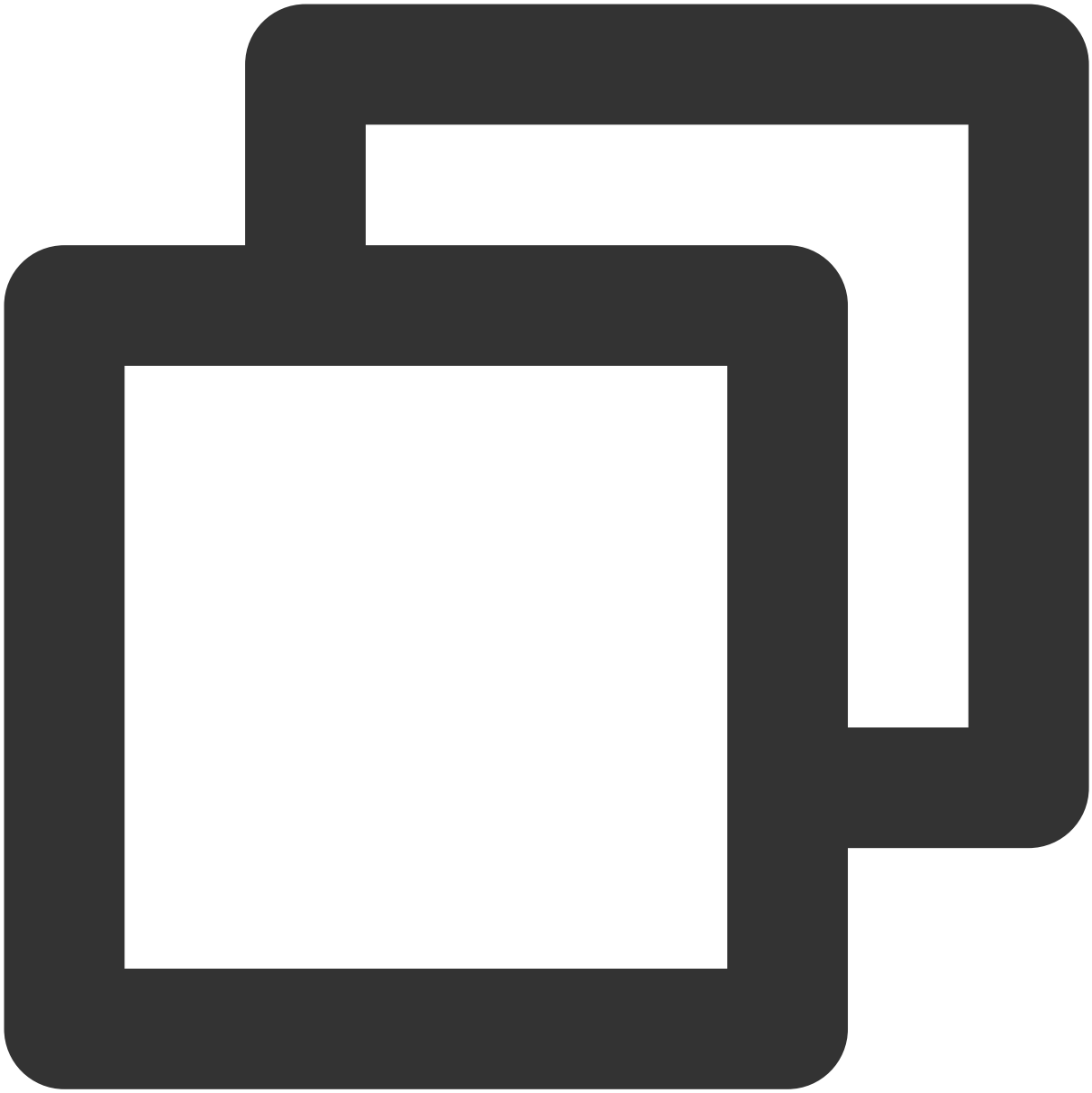
```
pulsar:
  # 命名空间名称
  namespace: namespace_java
  # 服务接入地址
  service-url: http://pulsar-xxx.tdmq.ap-gz.public.tencenttdmq.com:8080
```

```
# 授权角色密钥
token-auth-value: eyJrZXlJZC....
# 集群名称
tenant: pulsar-xxx
```

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

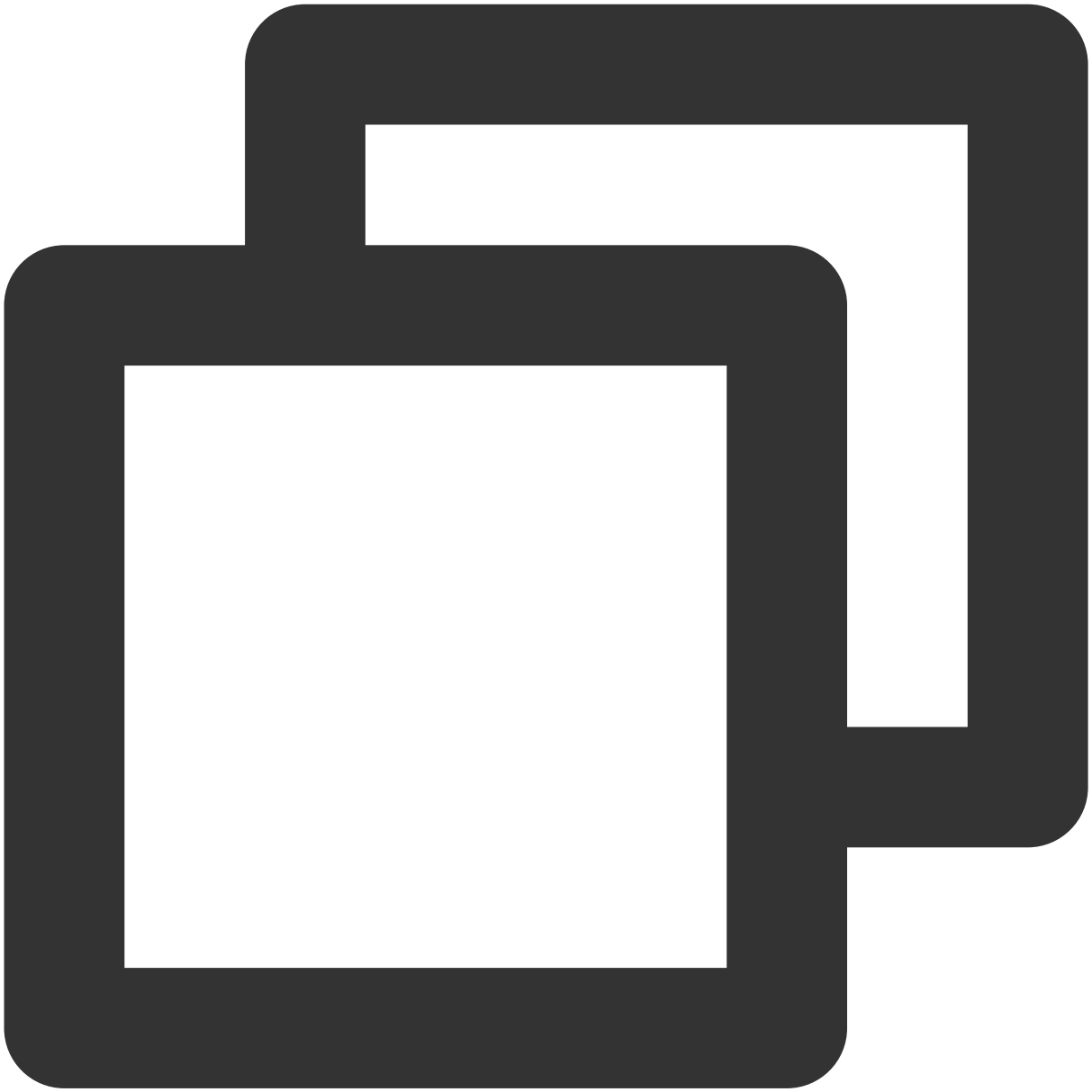
1. 生产者配置。



```
@Configuration
public class ProducerConfiguration {

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

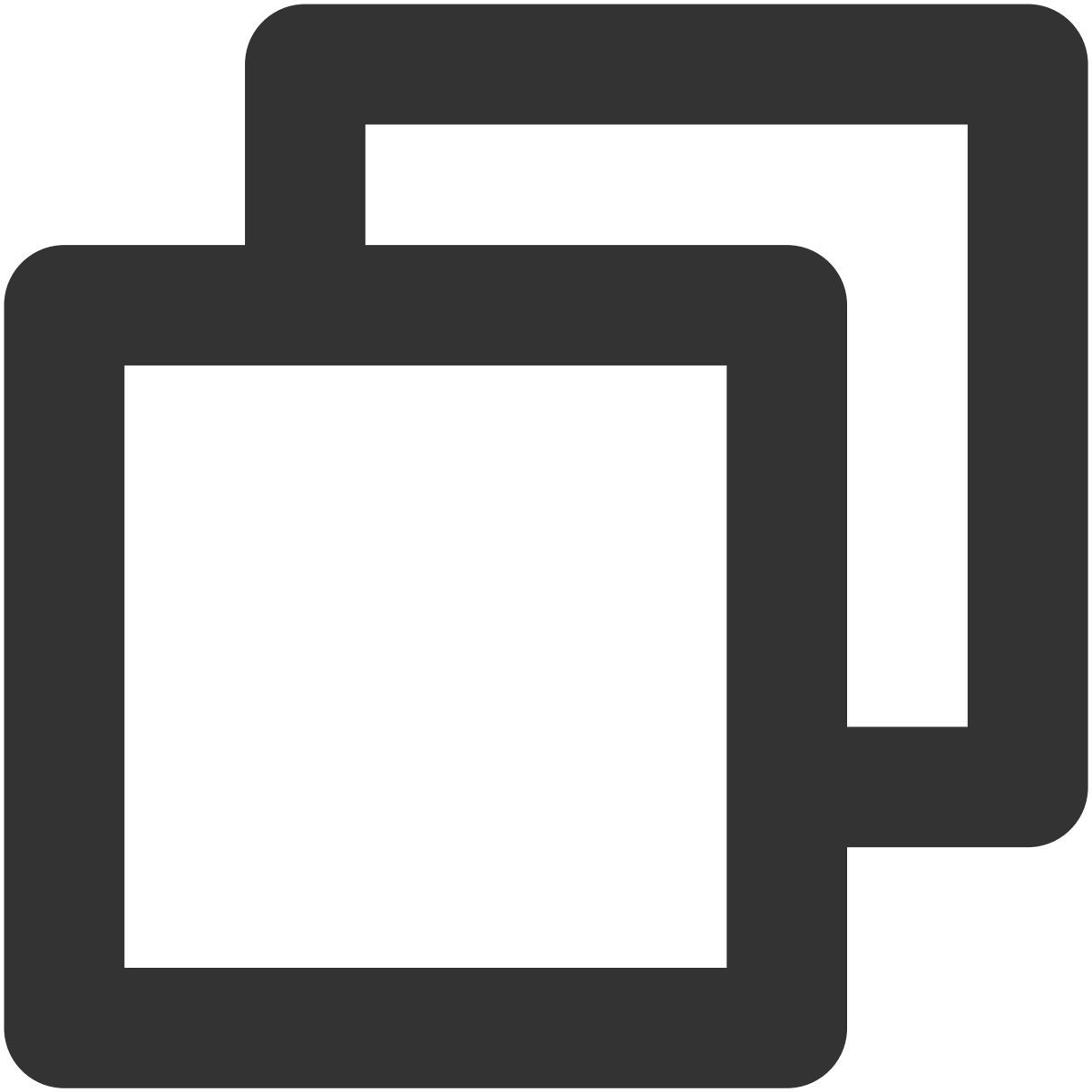
```
}
```

2. 注入生产者。



```
@Autowired
private PulsarTemplate<byte[]> defaultProducer;
```

3. 发送消息。



```
// 发送消息
defaultProducer.send("topic2", ("Hello pulsar client, this is a order message.").ge
```

注意：

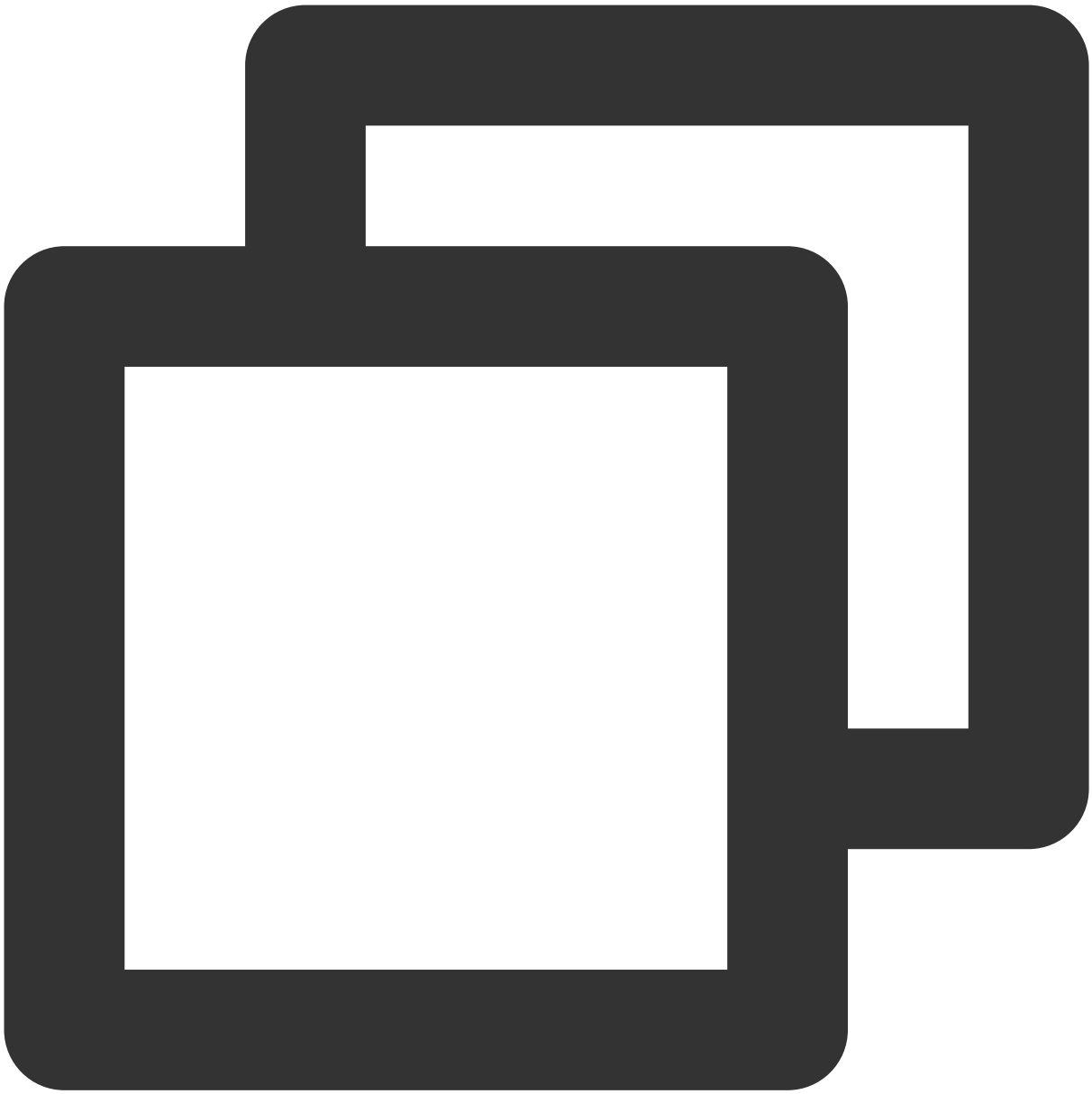
发送消息的 Topic 是在生产者配置中已经声明的 Topic。

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

消费者配置。

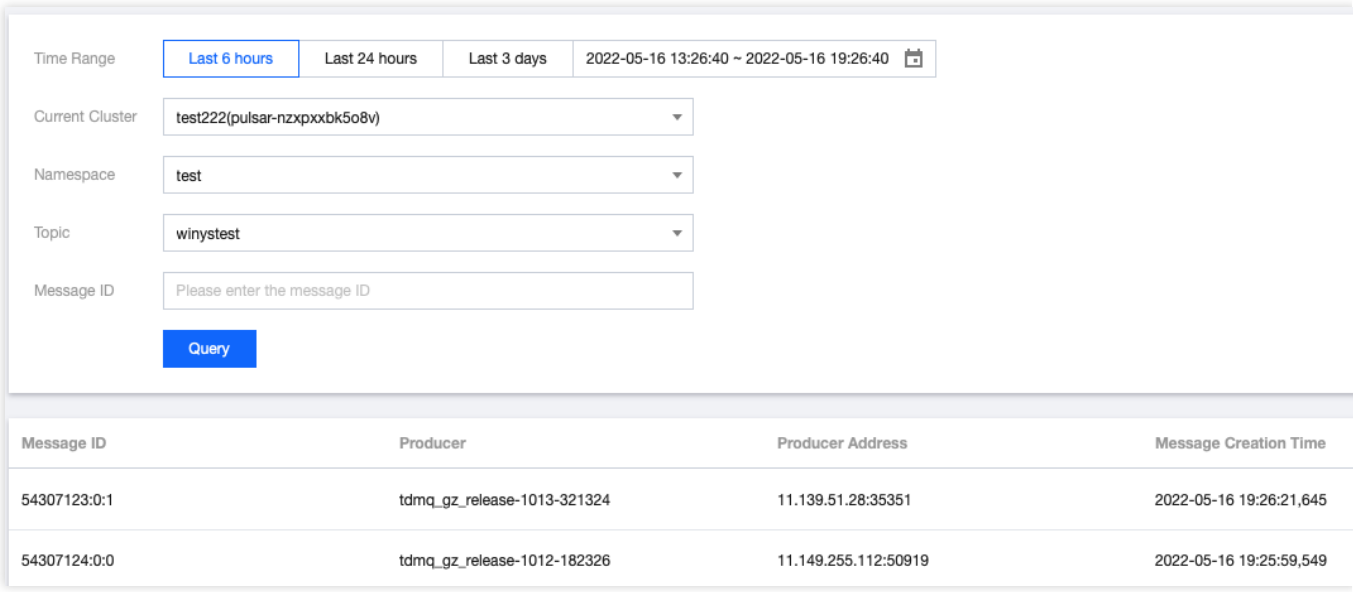


```
@PulsarConsumer(topic = "topic1",  // 订阅topic名称
                subscriptionName = "sub_topic1", // 订阅名称
                serialization = Serialization.JSON, // 序列化方式
                subscriptionType = SubscriptionType.Shared, // 订阅模式，默认为独占模式
                consumerName = "firstTopicConsumer", // 消费者名称
                maxRedeliverCount = 3, // 最大重试次数
                deadLetterTopic = "sub_topic1-DLQ" // 死信topic名称
               )
public void topicConsume(byte[] msg) {
```

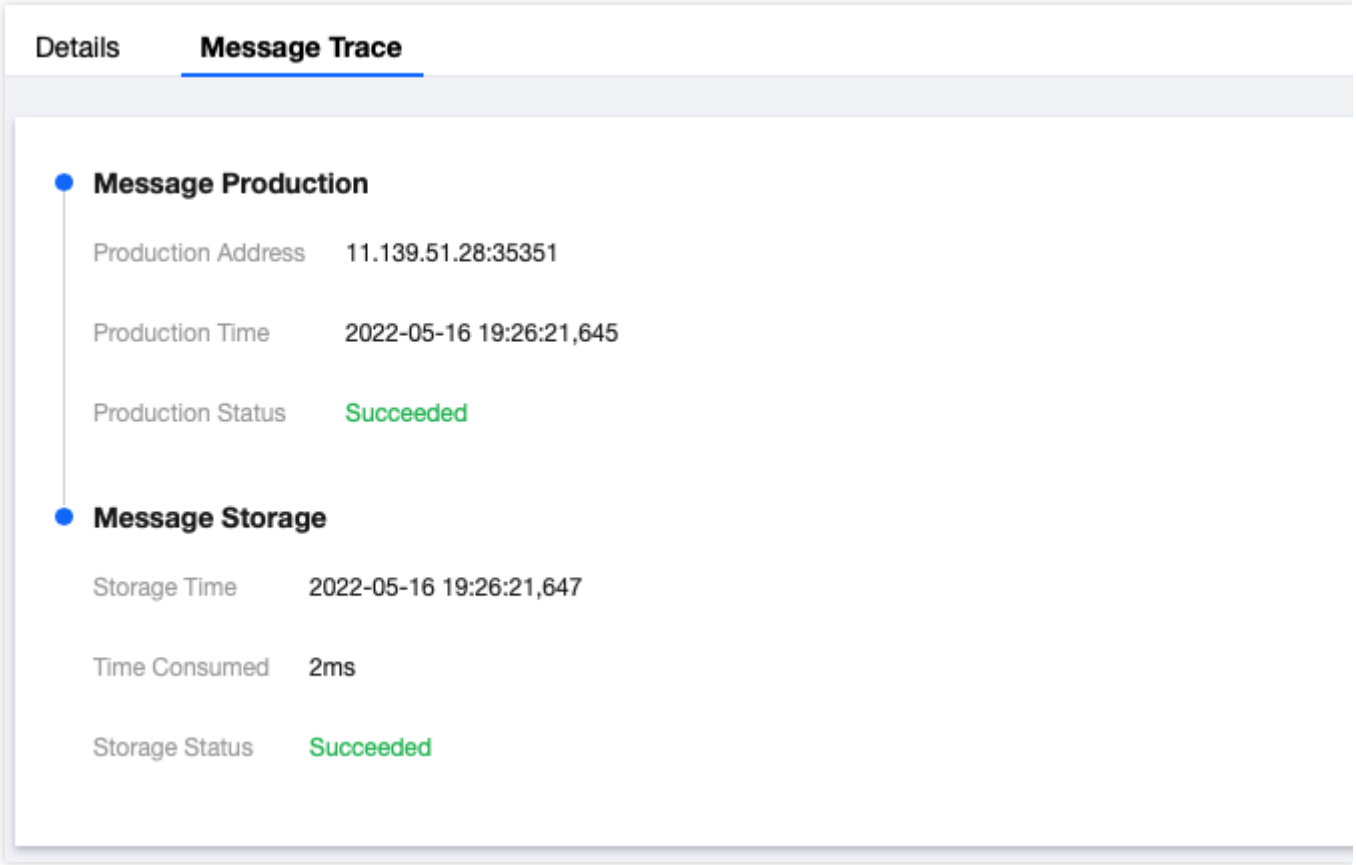
```
        // TODO process your message
        System.out.println("Received a new message. content: [" + new String(msg) + "]"
        // 如果消费失败，请抛出异常，这样消息会进入重试队列，之后可以重新消费，直到达到最大重试次数之后
    }
```

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。

Java SDK

最近更新时间：2024-01-03 14:27:38

操作场景

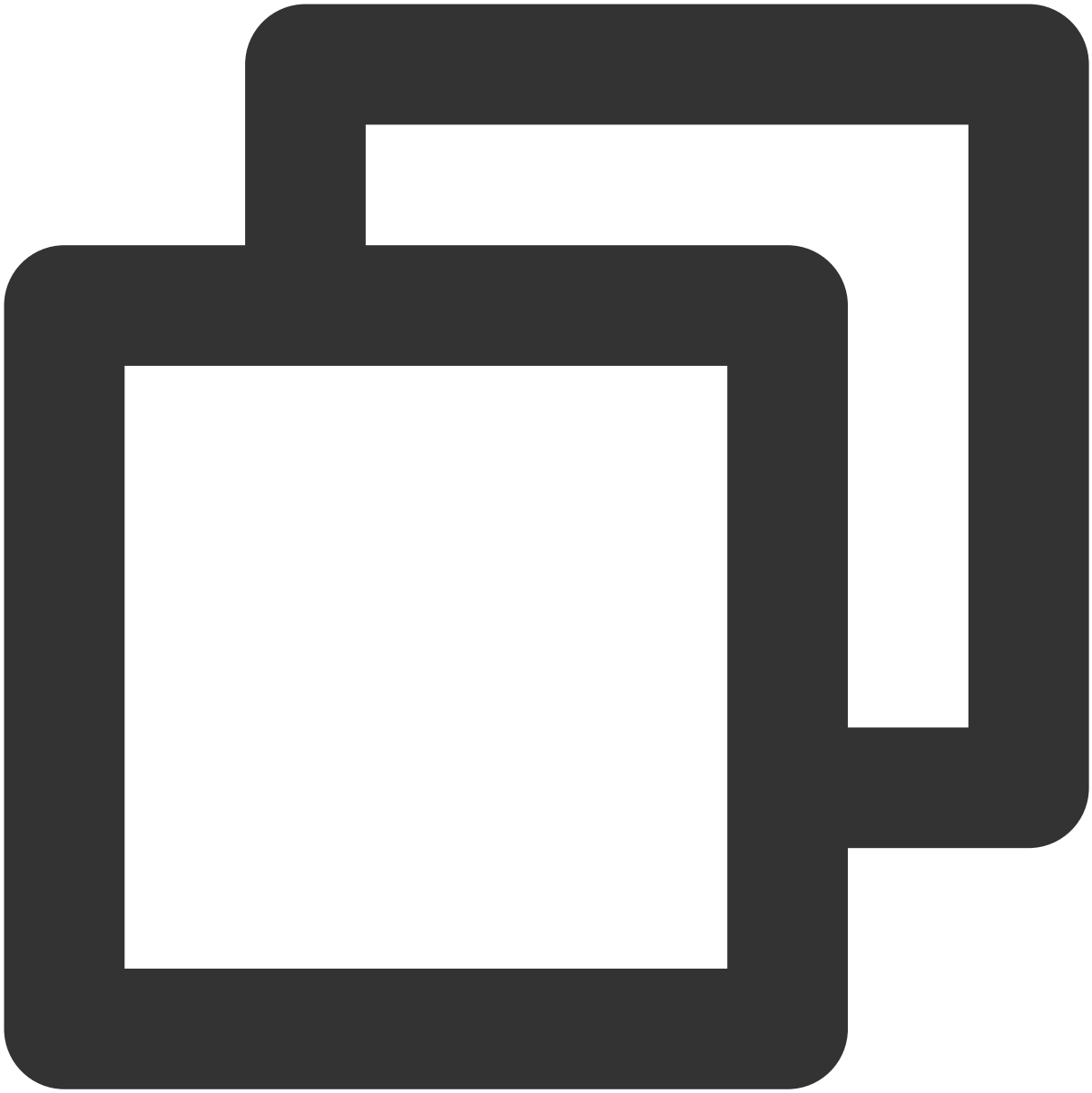
本文以调用 Java SDK 为例介绍通过开源 SDK 实现消息收发的操作过程，帮助您更好地理解消息收发的完整过程。

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

1. 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：



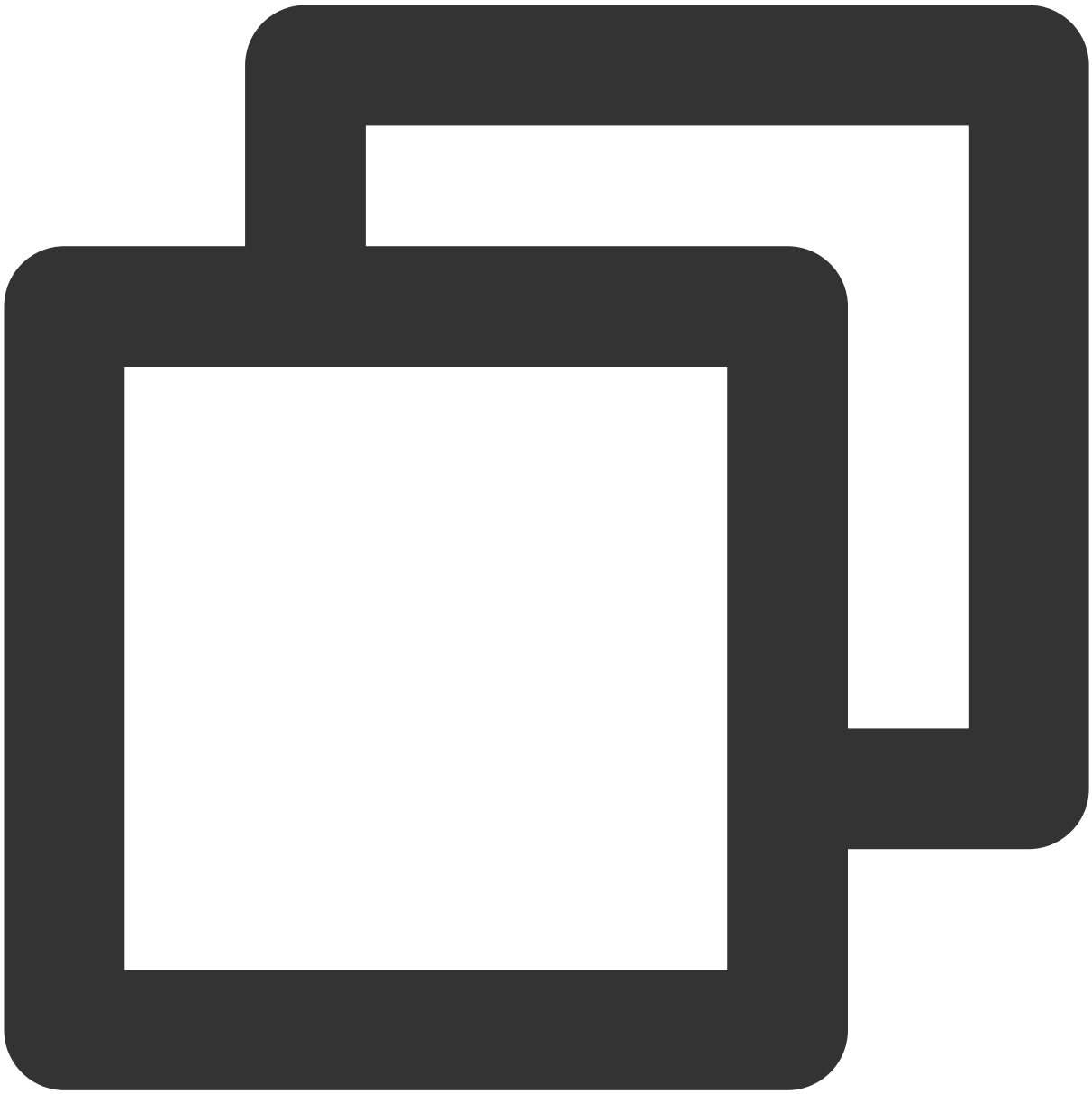
```
<dependency>
        <groupId>org.apache.pulsar</groupId>
        <artifactId>pulsar-client</artifactId>
        <version>2.7.2</version>
</dependency>
```

说明：

建议使用 2.7.2 及以上版本。

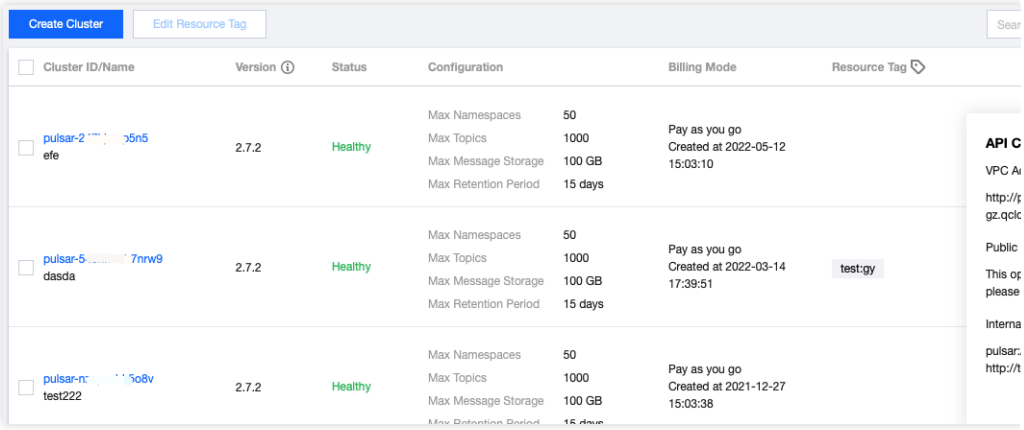
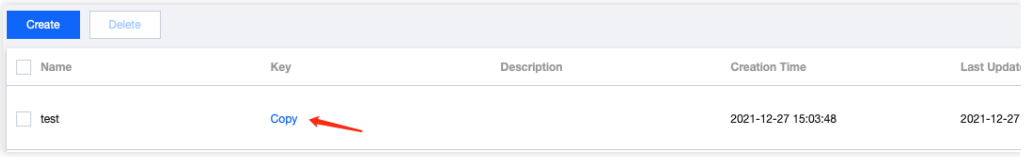
如果在客户端中使用批量收发消息功能（BatchReceive），则使用 2.7.4 及以上版本的 SDK。

2. 创建 Pulsar 客户端。

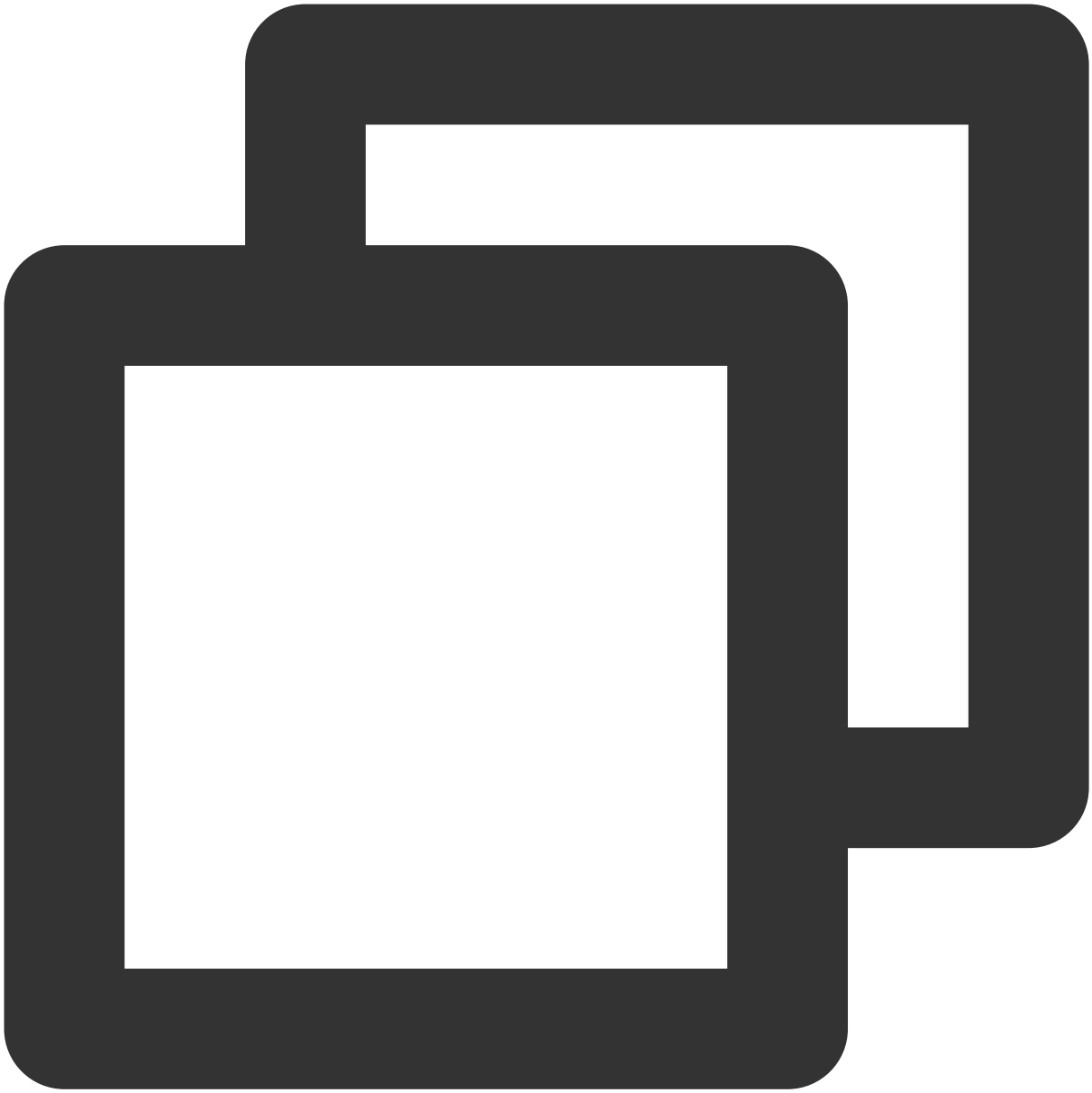


```
PulsarClient pulsarClient = PulsarClient.builder()
                // 服务接入地址
                .serviceUrl(SERVICE_URL)
                // 授权角色密钥
                .authentication(AuthenticationFactory.token(AUTHENTICATION)).bui
```

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。

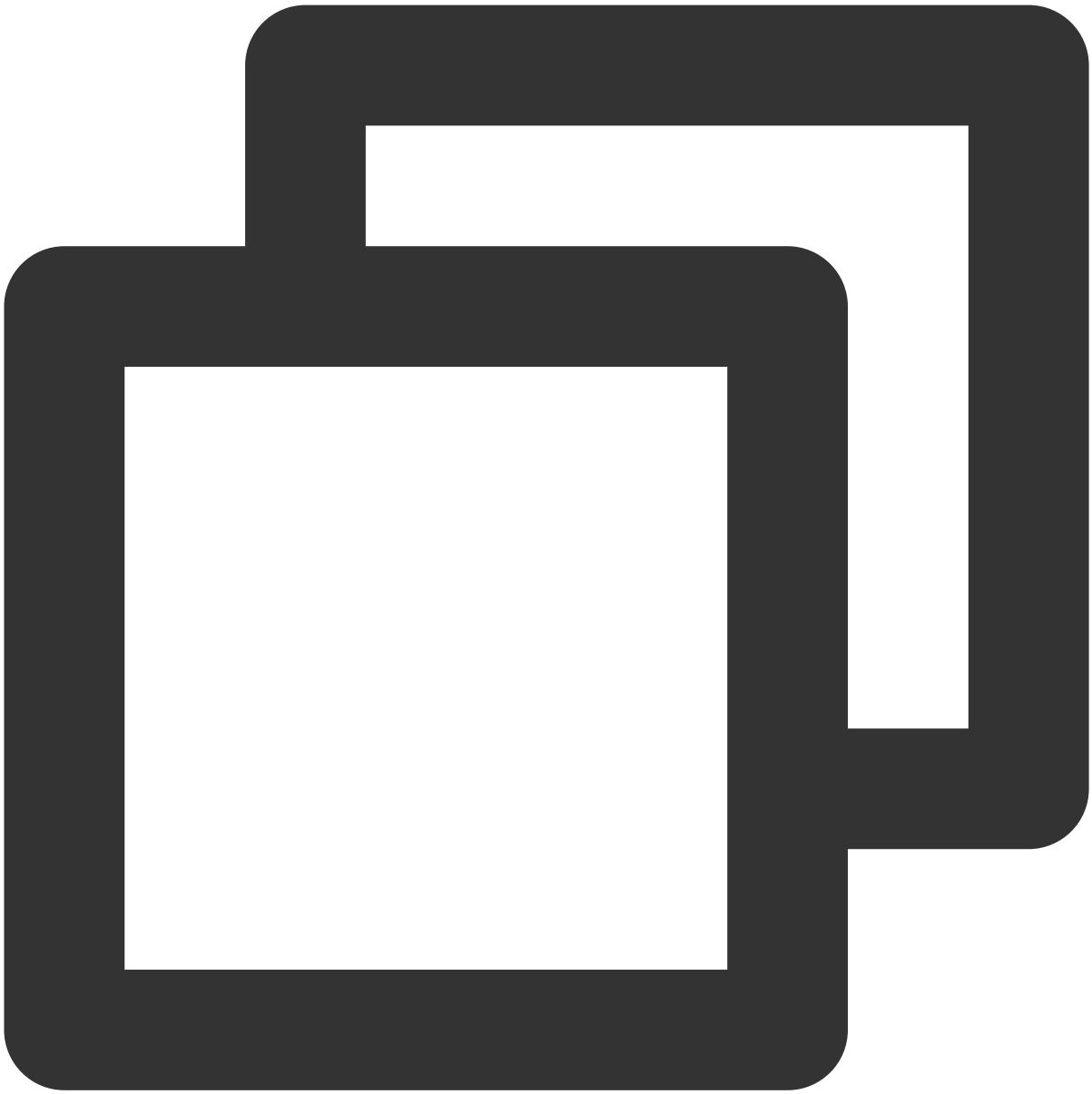


```
// 构建byte[]类型的生产者
Producer<byte[]> producer = pulsarClient.newProducer()
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称
    .topic("persistent://pulsar-xxx/sdk_java/topic1").create();
```

说明：

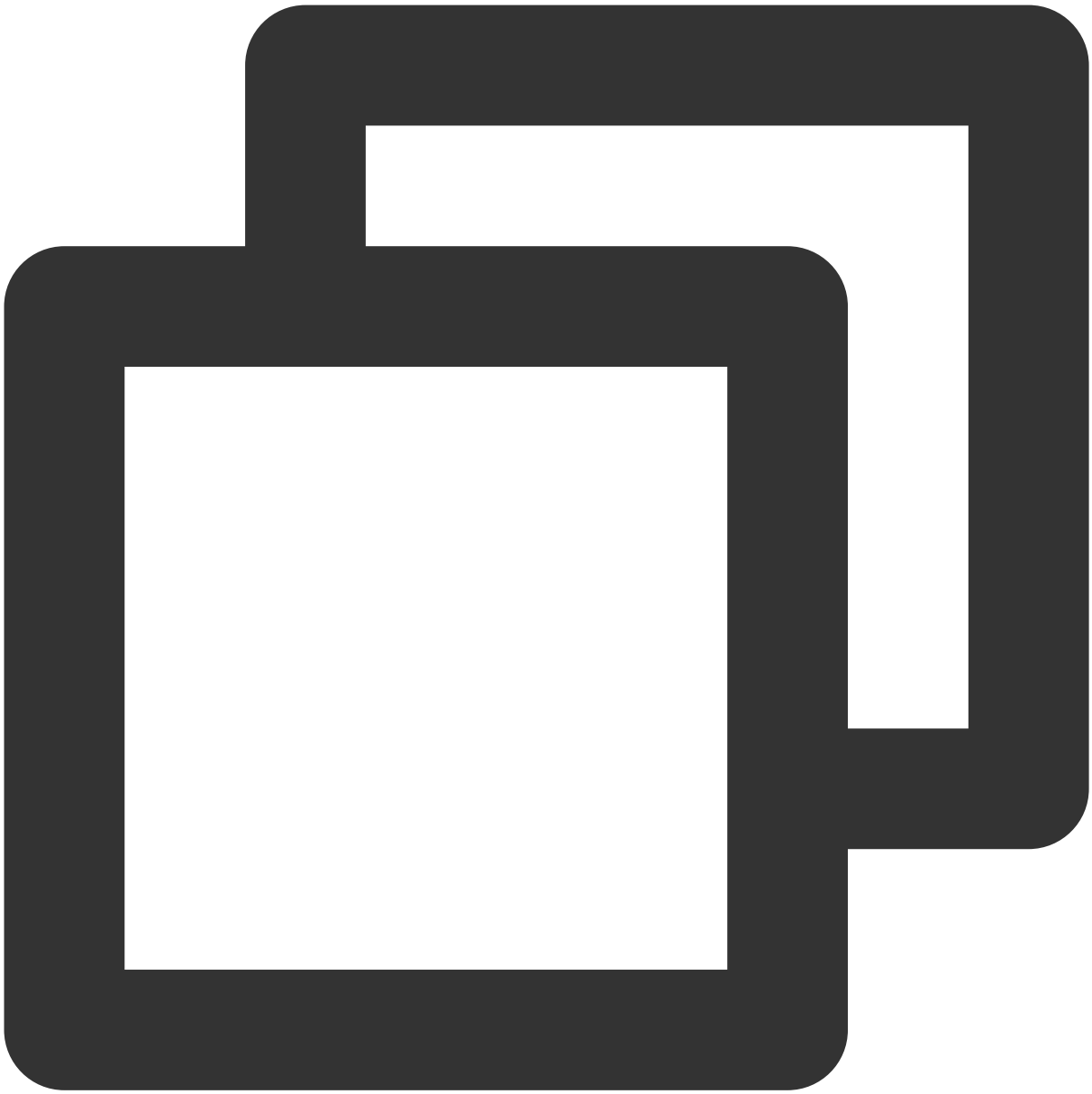
Topic 名称需要填入完整路径，即 `persistent://clusterid/namespace/Topic` ， `clusterid/namespace/topic` 的部分可以从控制台上 **Topic管理** 页面直接复制。

4. 发送消息。



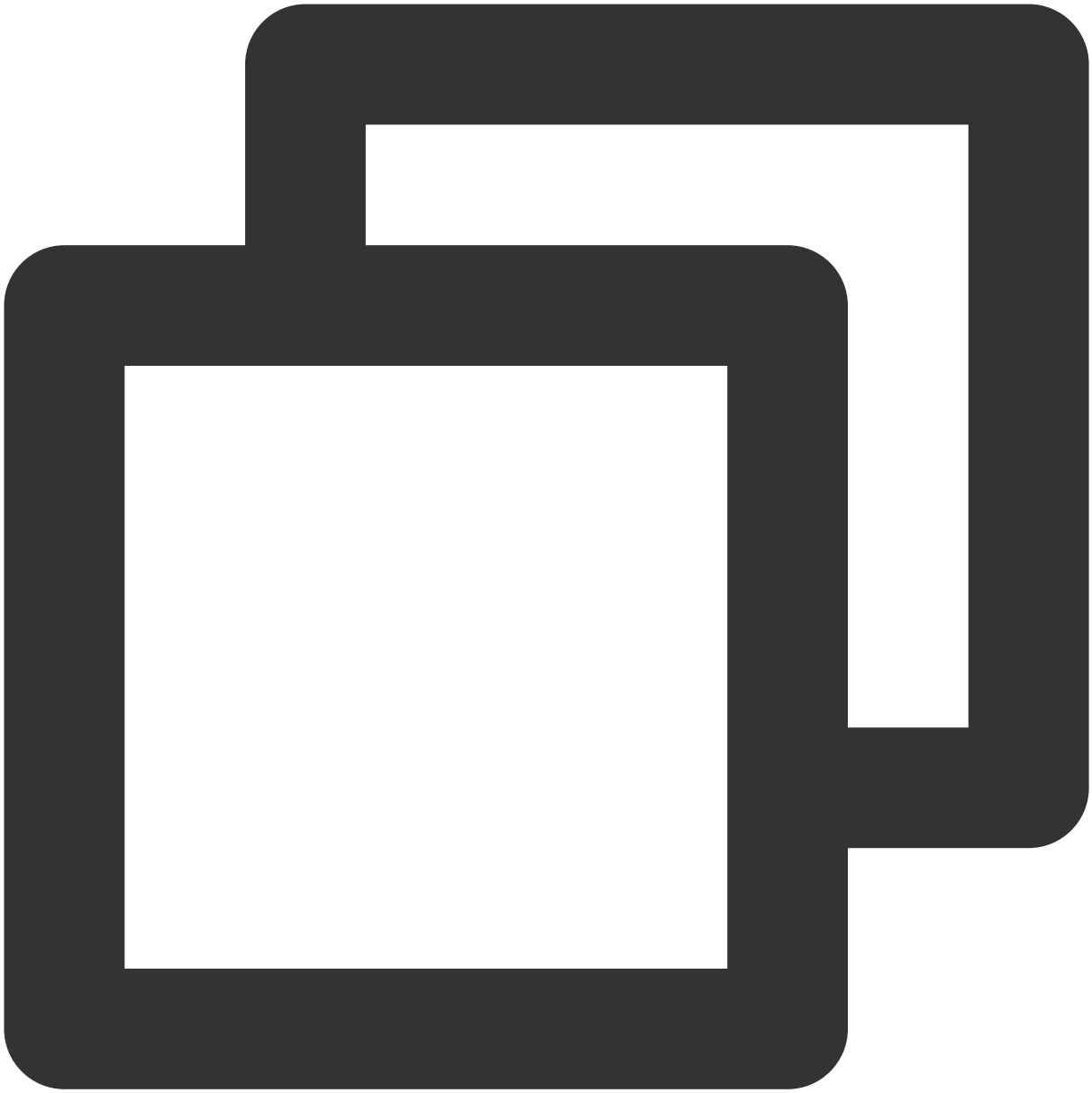
```
//发送消息
MessageId msgId = producer.newMessage()
    // 消息内容
    .value("this is a new message.".getBytes(StandardCharsets.UTF_8))
    // 业务key
    .key("youKey")
    // 业务相关参数
    .property("mykey", "myvalue").send();
```

5. 资源释放。



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

6. 创建消费者。

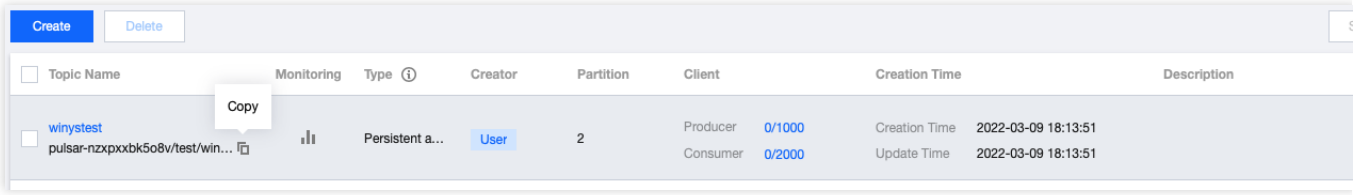


```
// 构建byte[]类型（默认类型）的消费者
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管理
    .topic("persistent://pulsar-xxx/sdk_java/topic1")
    // 需要在控制台Topic详情页创建好一个订阅，此处填写订阅名
    .subscriptionName("sub_topic1")
    // 声明消费模式为exclusive（独占）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)
    // 订阅
```

```
        .subscribe();
```

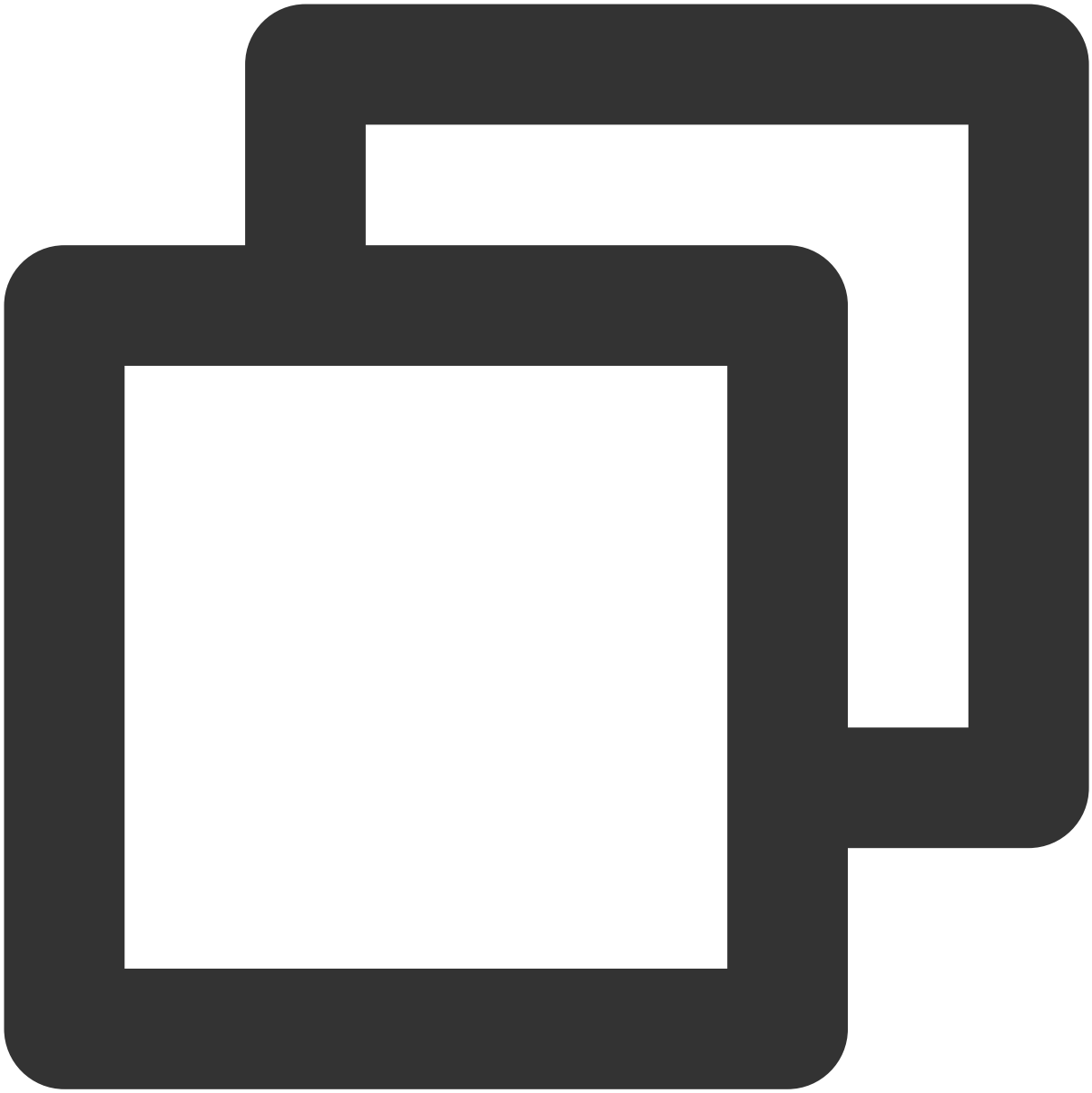
说明：

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。



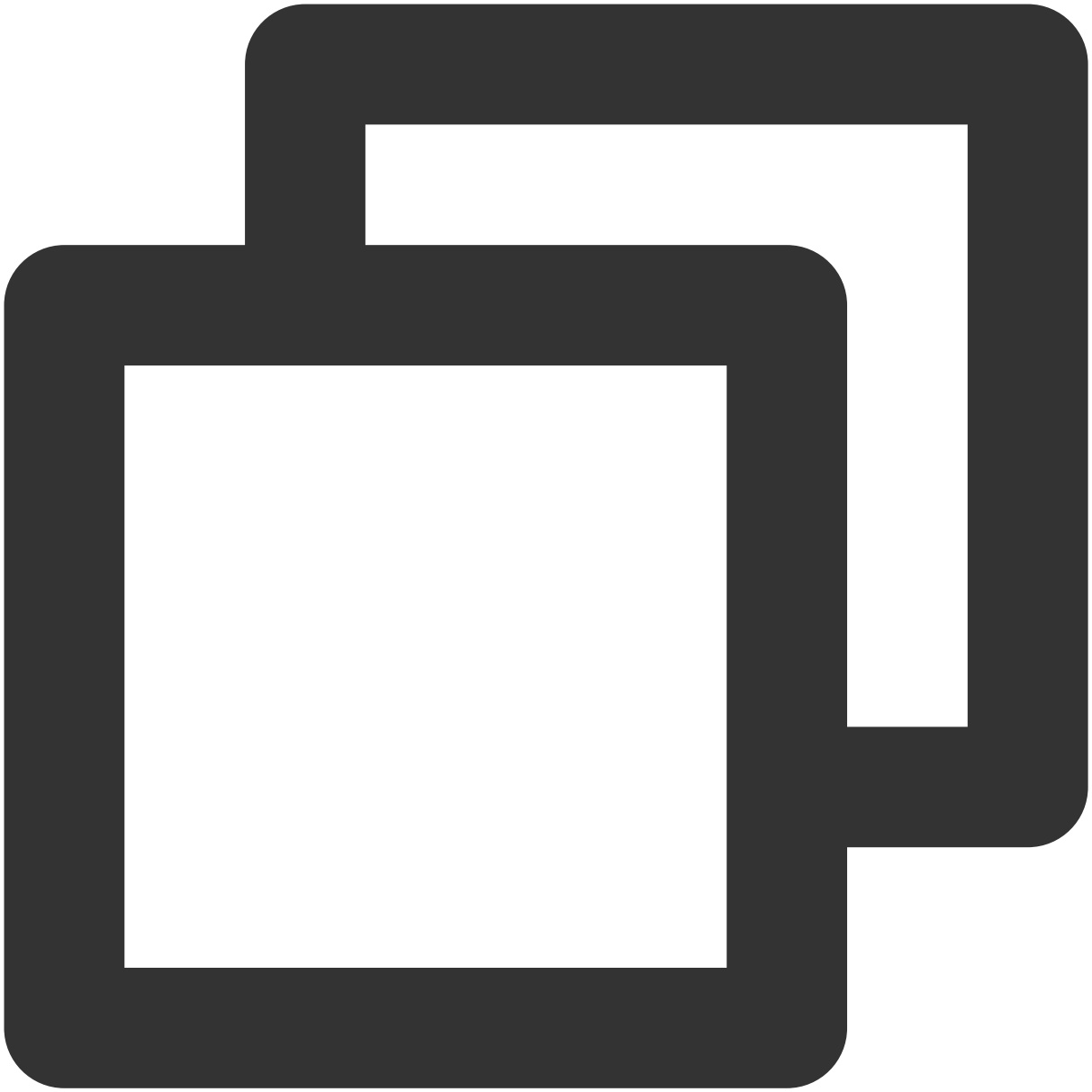
subscriptionName 需要写入订阅名，可在**消费管理**界面查看。

7. 消费消息。



```
// 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);
```

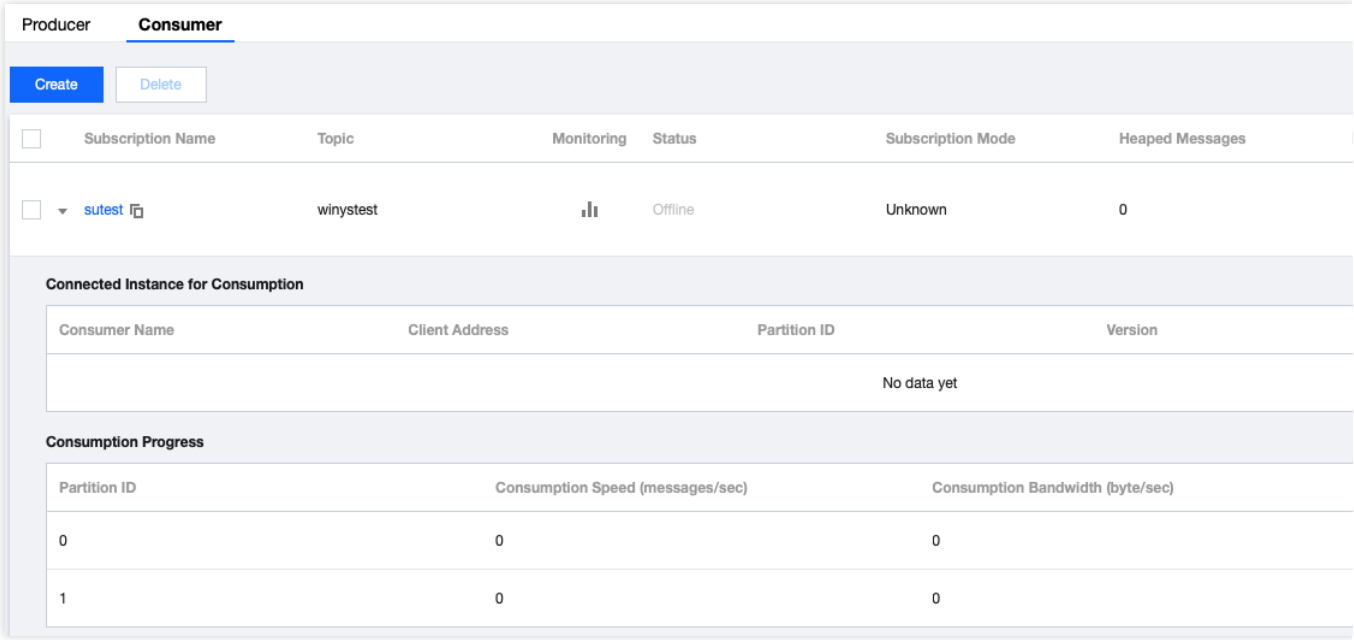
8. 使用监听器进行消费。



```
// 消息监听器
MessageListener<byte[]> myMessageListener = (consumer, msg) -> {
    try {
        System.out.println("Message received: " + new String(msg.getData()));
        // 回复ack
        consumer.acknowledge(msg);
    } catch (Exception e) {
        // 消费失败，回复nack
        consumer.negativeAcknowledge(msg);
    }
};
```

```
    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
        .topic("persistent://pulsar-mmqwr5xx9n7g/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("sub_topic1")
        // 声明消费模式为exclusive（独占）模式
        .subscriptionType(SubscriptionType.Exclusive)
        // 设置监听器
        .messageListener(myMessageListener)
        // 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();
```

9. 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。

Go SDK

最近更新时间：2024-01-03 14:27:38

操作场景

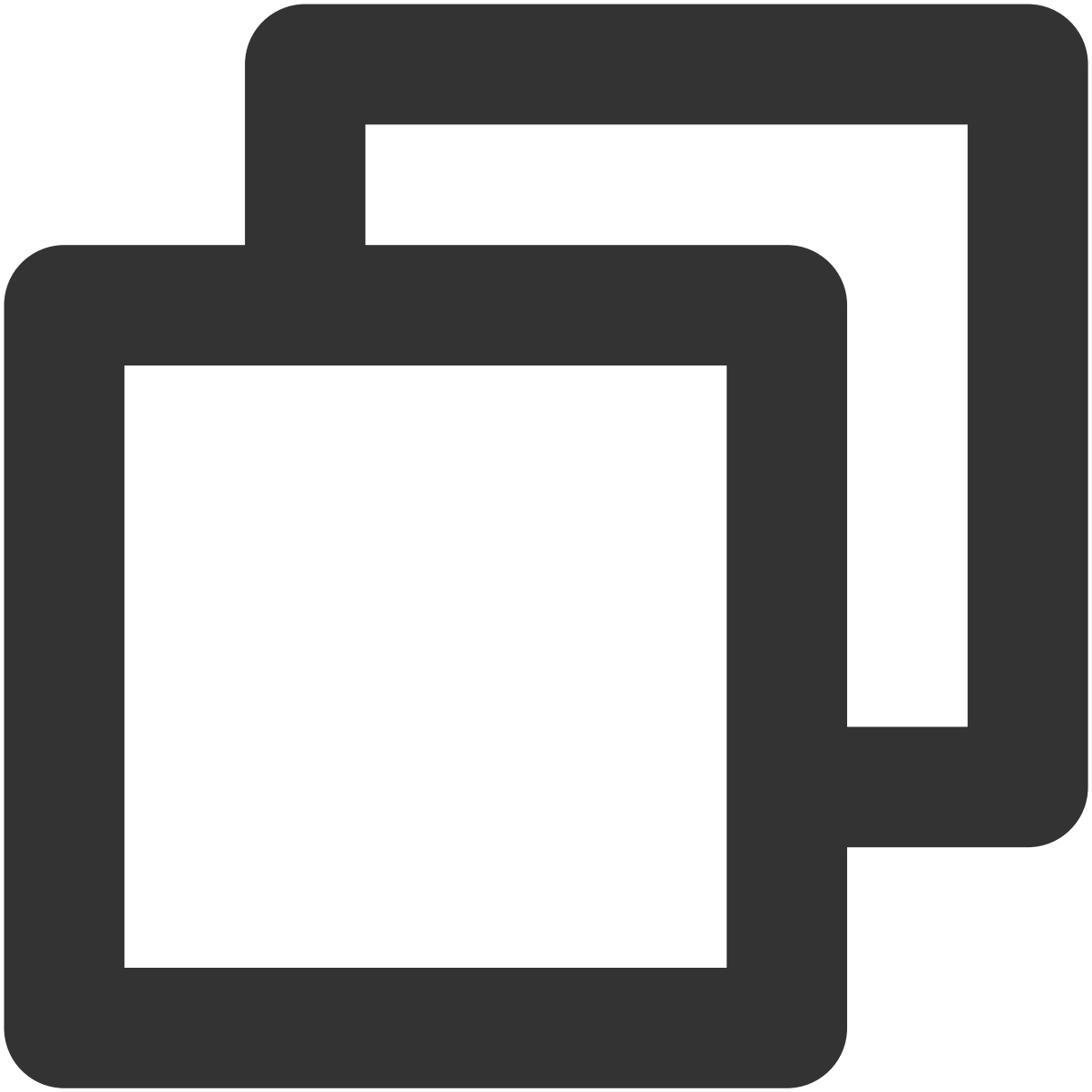
本文以调用 Go SDK 为例介绍通过开源 SDK 实现消息收发的操作过程，帮助您更好地理解消息收发的完整过程。

前提条件

完成资源创建与准备
安装 Go
下载 Demo

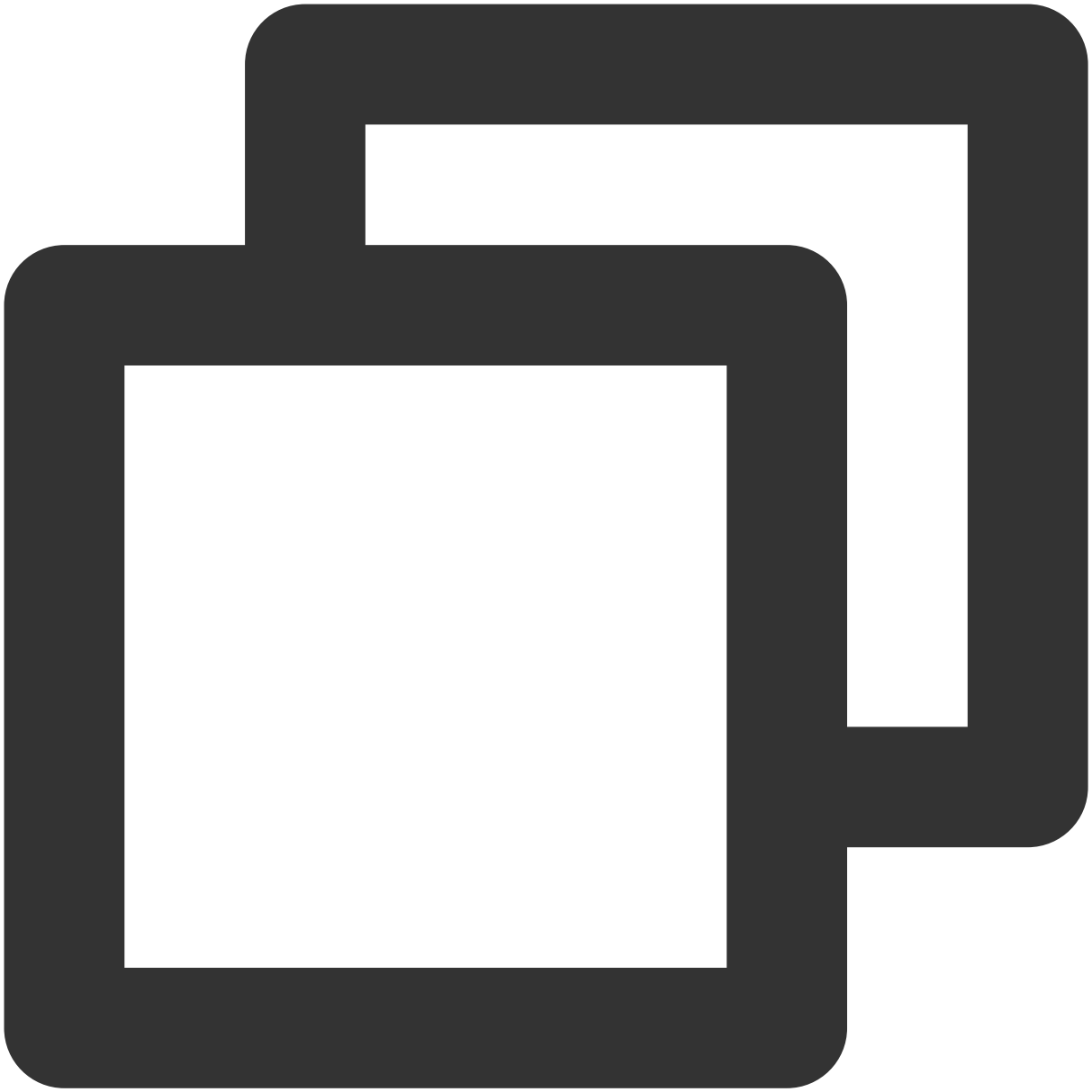
操作步骤

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。



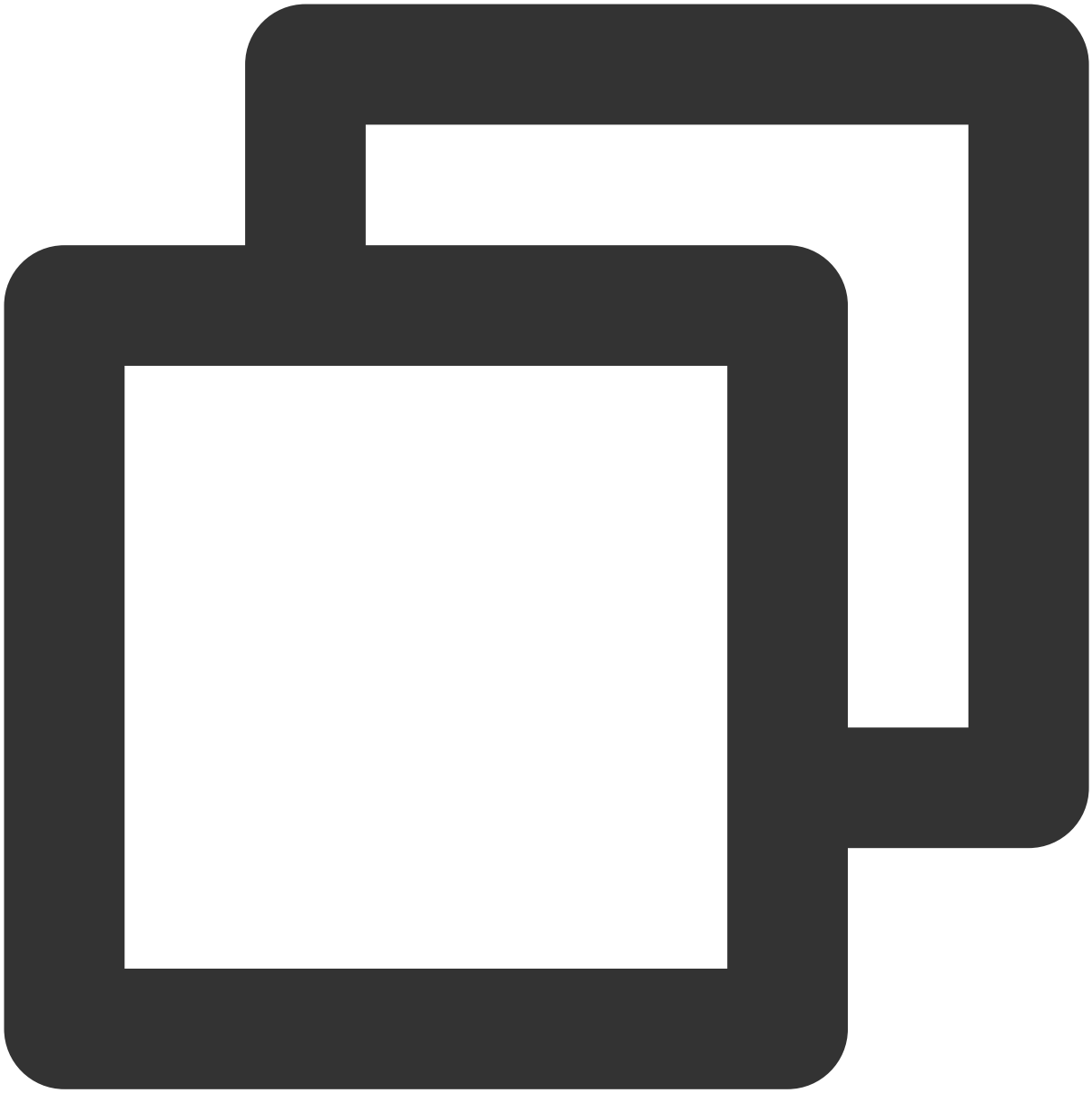
```
go get -u "github.com/apache/pulsar-client-go/pulsar"
```

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。



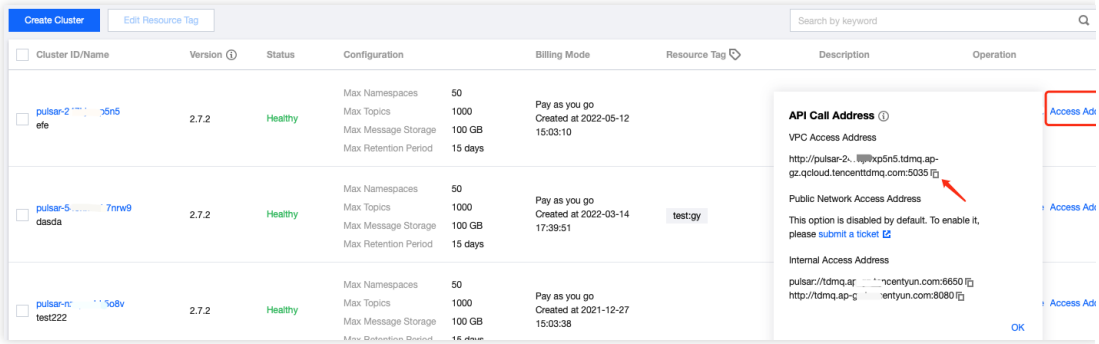
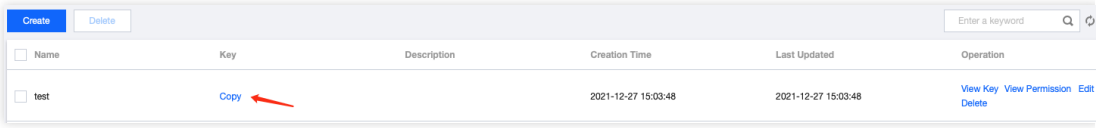
```
import "github.com/apache/pulsar-client-go/pulsar"
```

2. 创建 Pulsar Client。

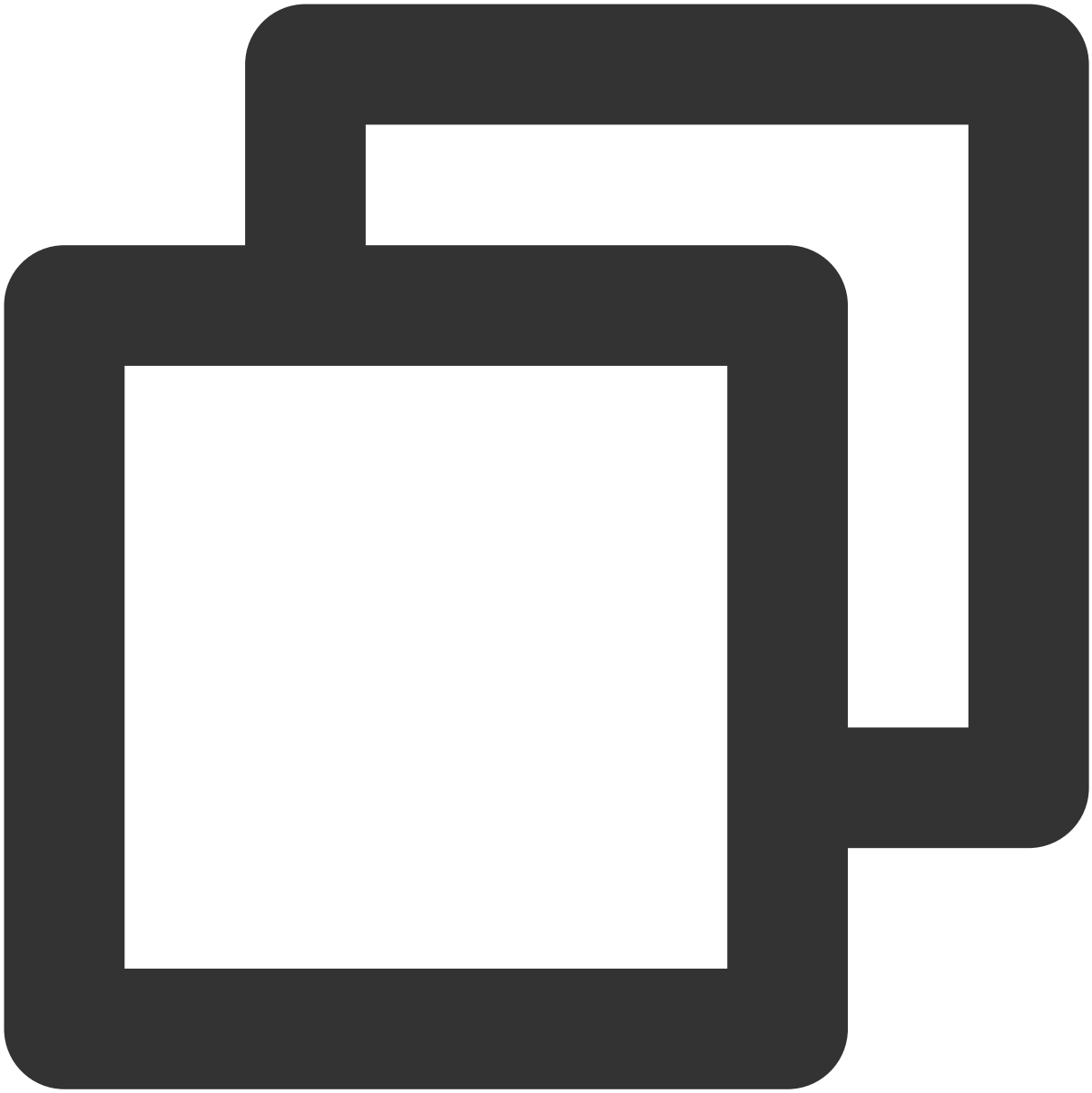


```
// 创建pulsar客户端
client, err := pulsar.NewClient(pulsar.ClientOptions{
    // 服务接入地址
    URL: serviceUrl,
    // 授权角色密钥
    Authentication:    pulsar.NewAuthenticationToken(authentication),
    OperationTimeout:  30 * time.Second,
    ConnectionTimeout: 30 * time.Second,
})

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。

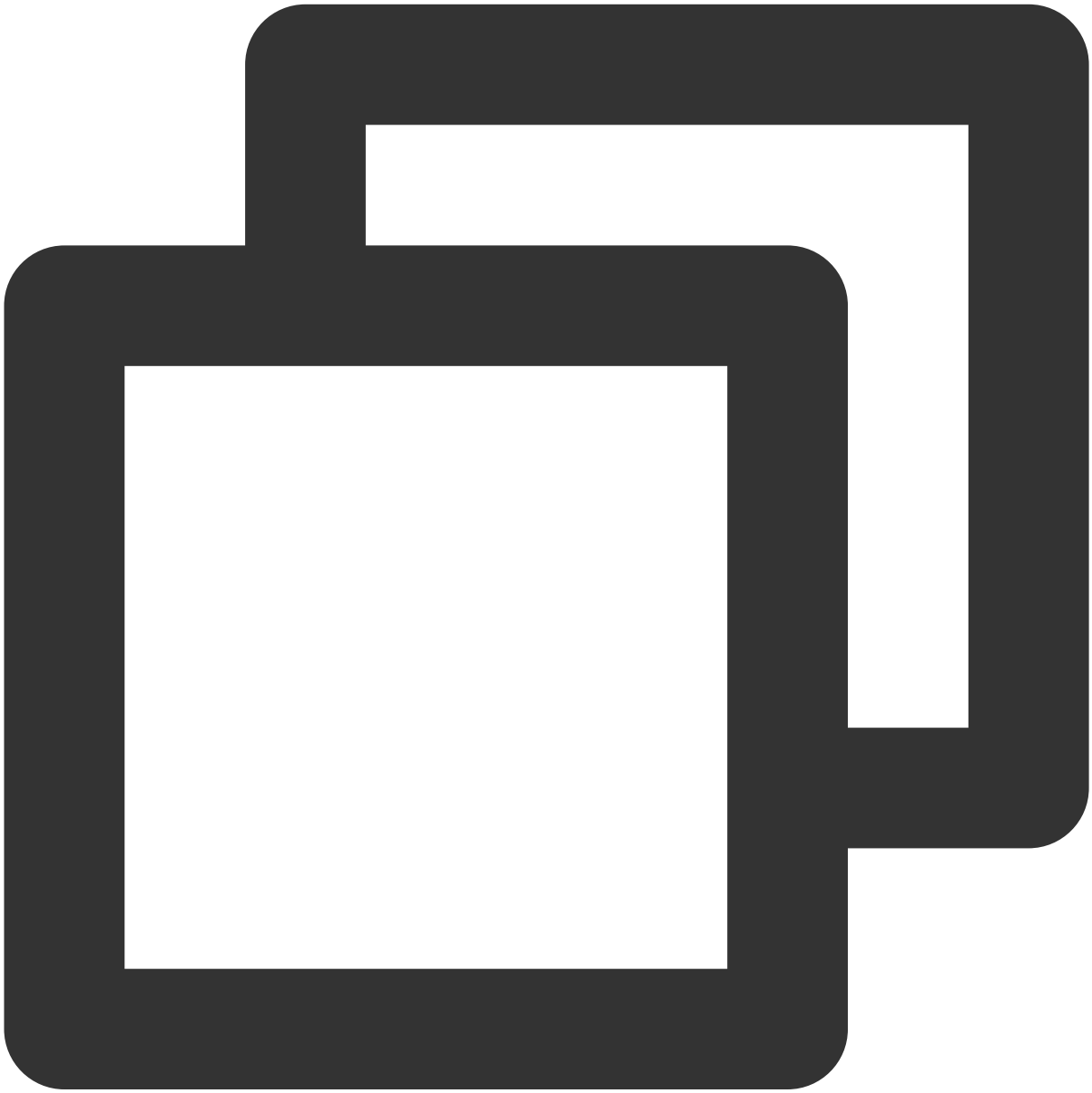


```
// 使用客户端创建生产者
producer, err := client.CreateProducer(pulsar.ProducerOptions{
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称
    Topic: "persistent://pulsar-mmqwr5xx9n7g/sdk_go/topic1",
})

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。

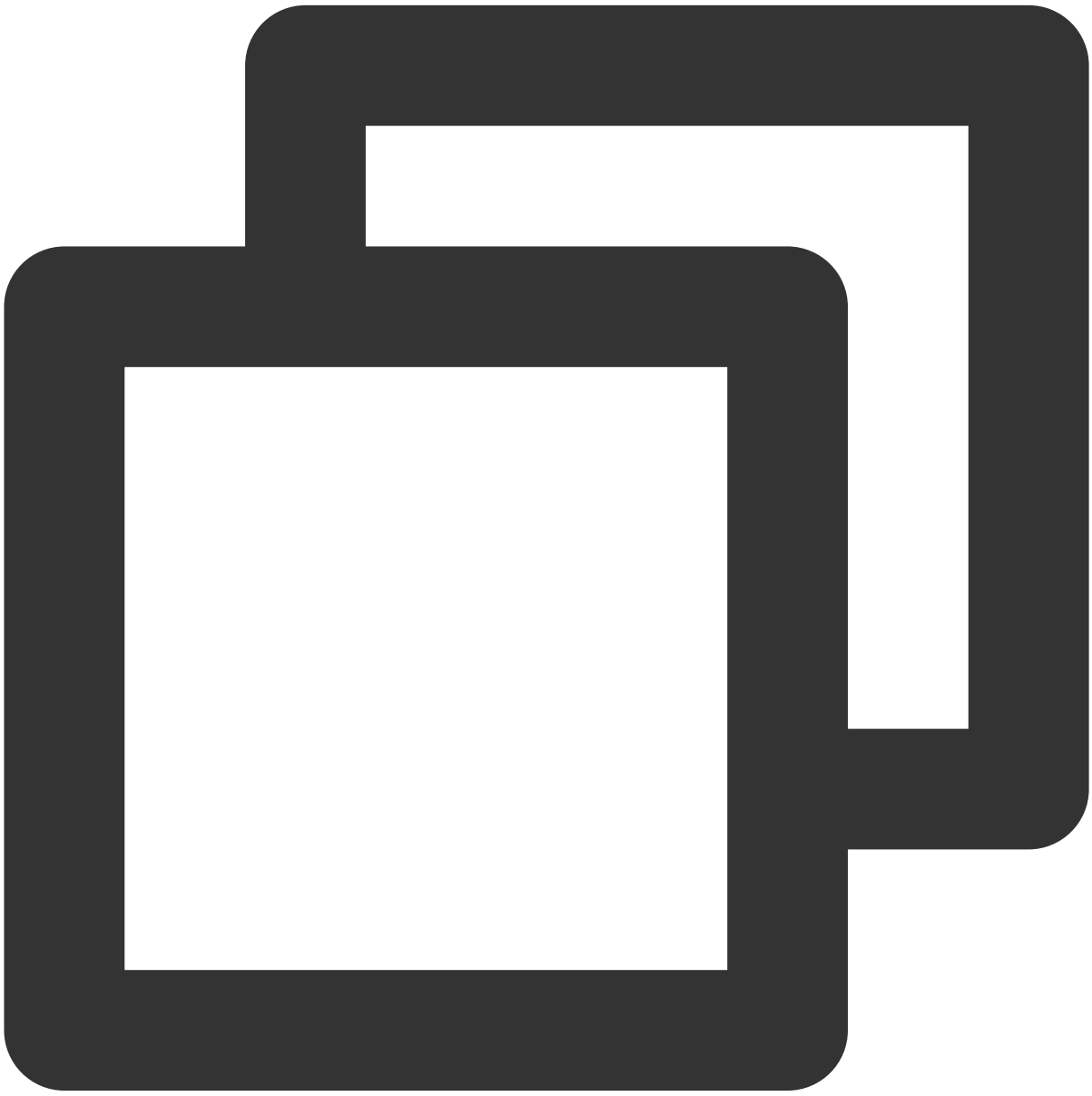
4. 发送消息。



```
// 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。

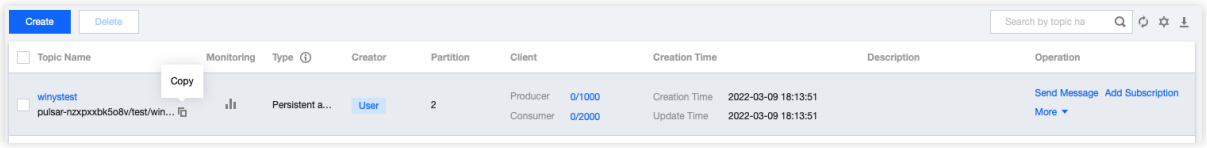


```
// 使用客户端创建消费者
consumer, err := client.Subscribe(pulsar.ConsumerOptions{
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称
    Topic:            "persistent://pulsar-mmqwr5xx9n7g/sdk_go/topic1",
```

```
        // 订阅名称
        SubscriptionName: "topic1_sub",
        // 订阅模式
        Type:             pulsar.Shared,
    })
    if err != nil {
        log.Fatal(err)
    }
    defer consumer.Close()
```

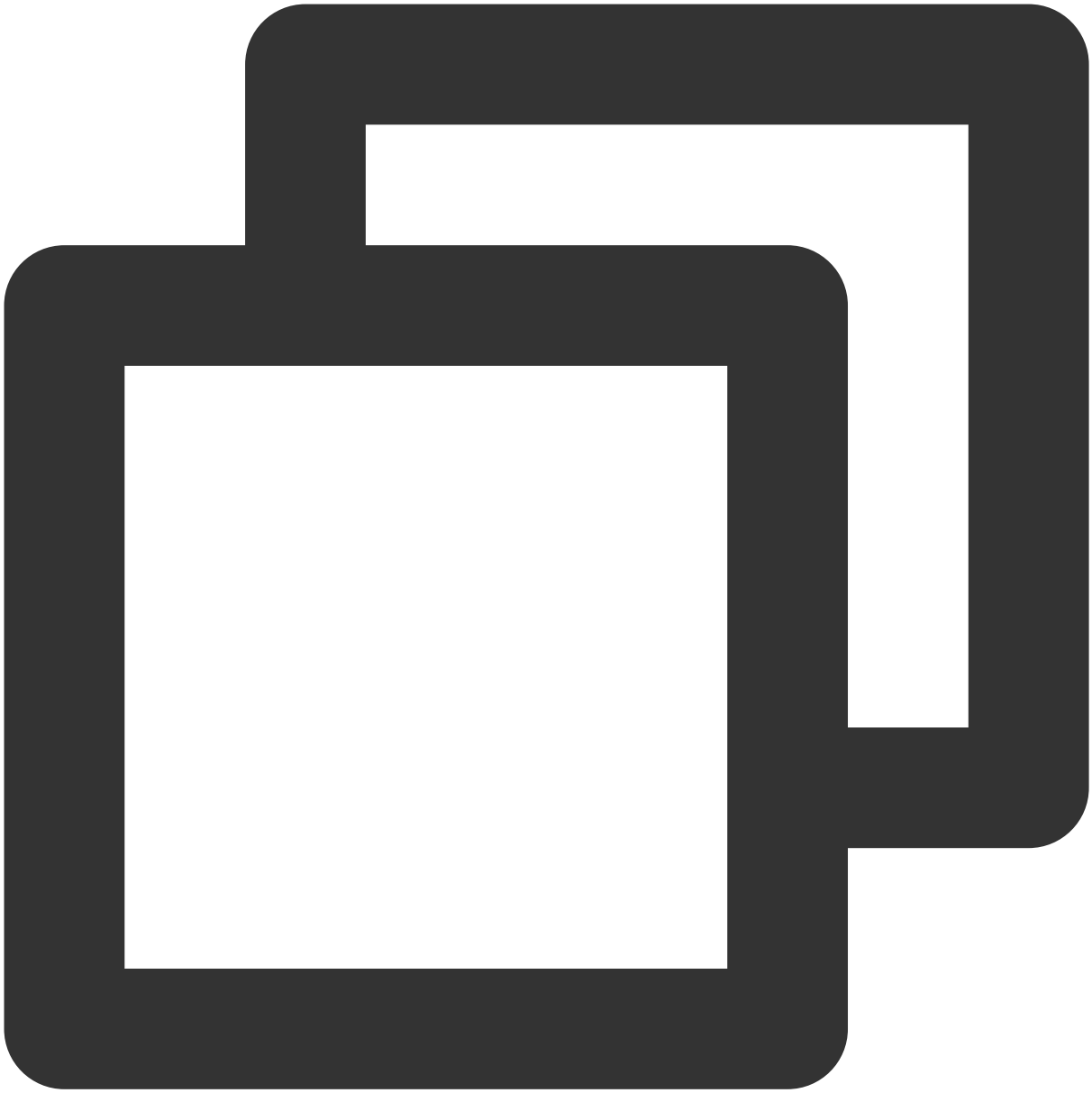
说明：

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。



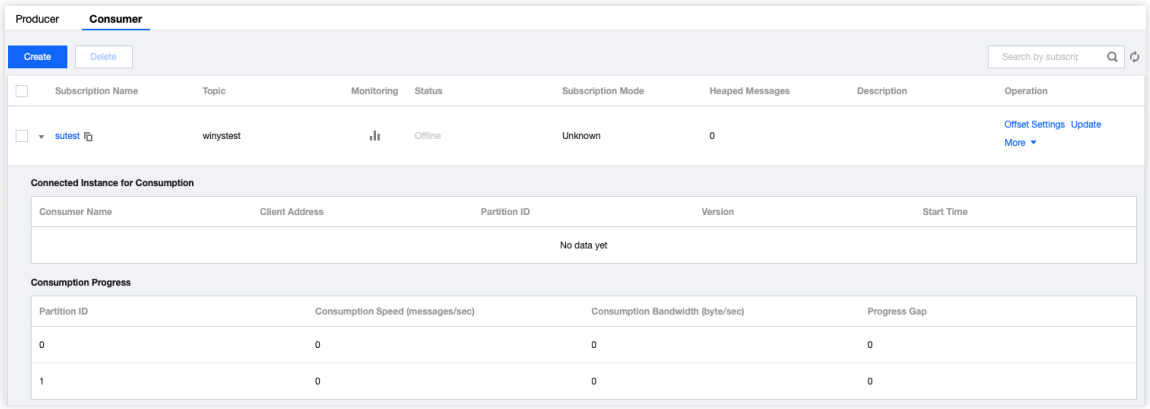
subscriptionName 需要写入订阅名，可在**消费管理**界面查看。

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。



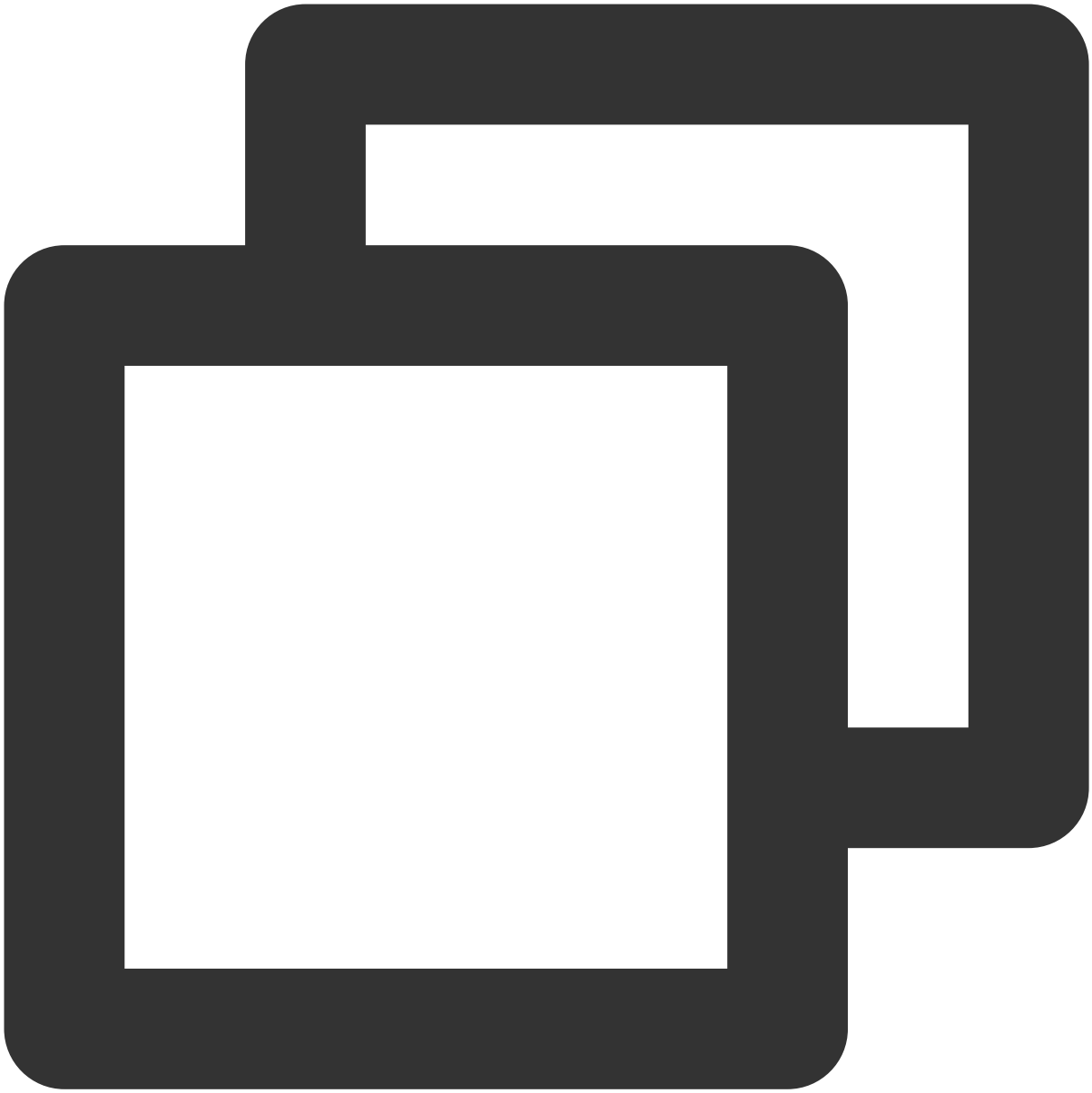
说明：

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：



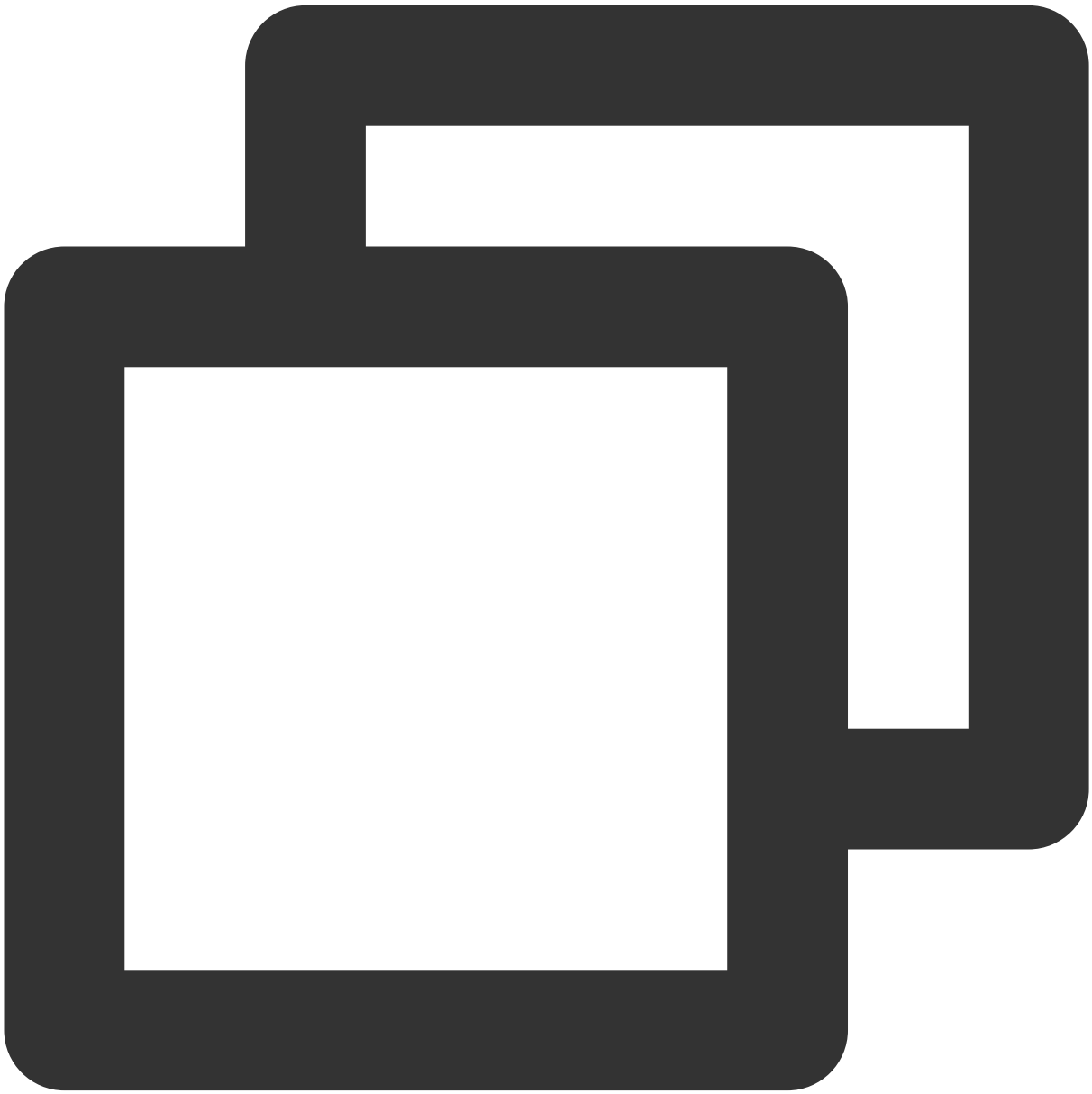
```
// It's recommended to make this a global instance called `log`.
func New() *Logger {
    return &Logger{
        Out:          os.Stderr, // 默认输出
        Formatter:    new(TextFormatter),
        Hooks:        make(LevelHooks),
        Level:        InfoLevel,
        ExitFunc:     os.Exit,
   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

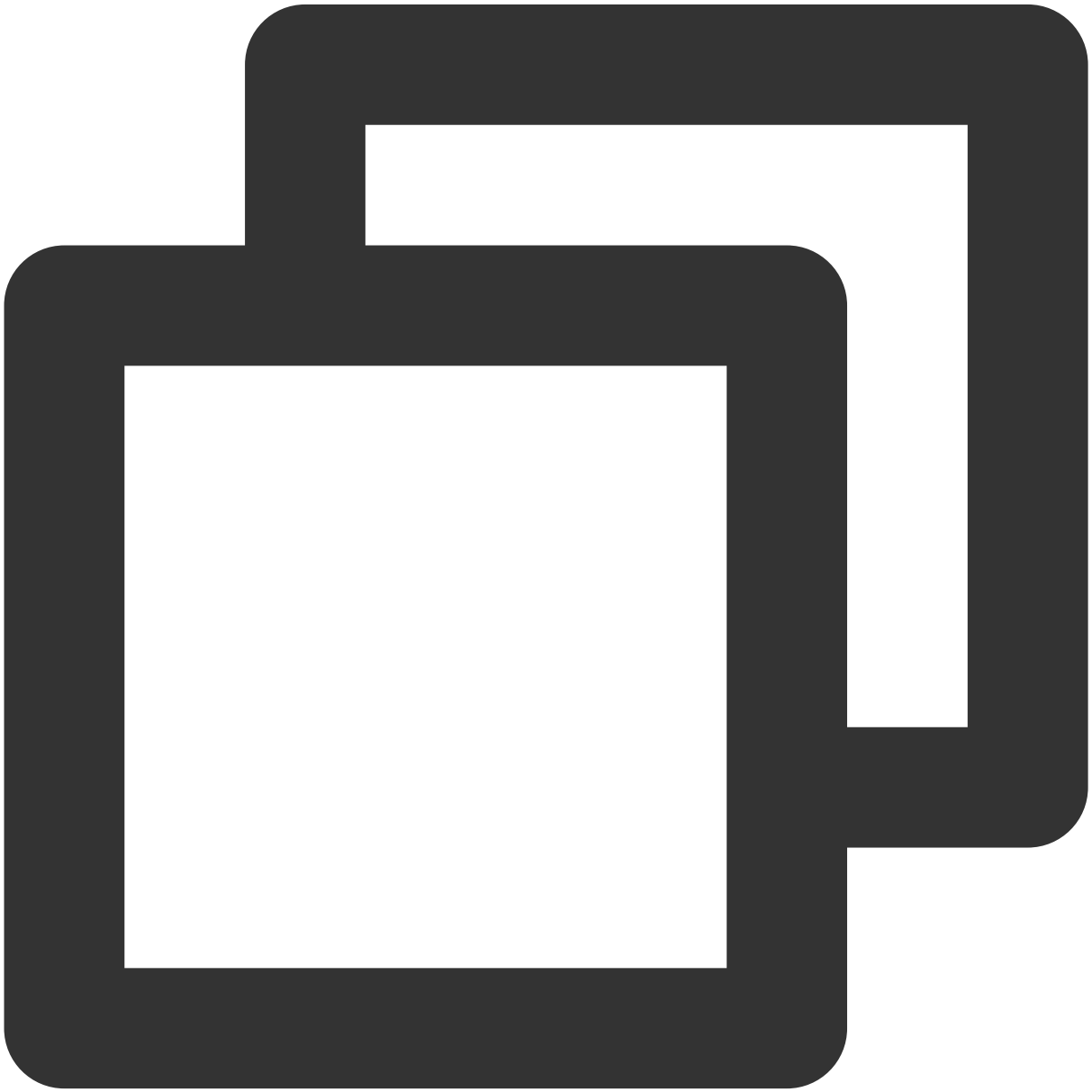
1. 自定义 log lib 实现 Pulsar Go SDK 提供的 log.Logger 的接口：



```
// 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 idi
    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：

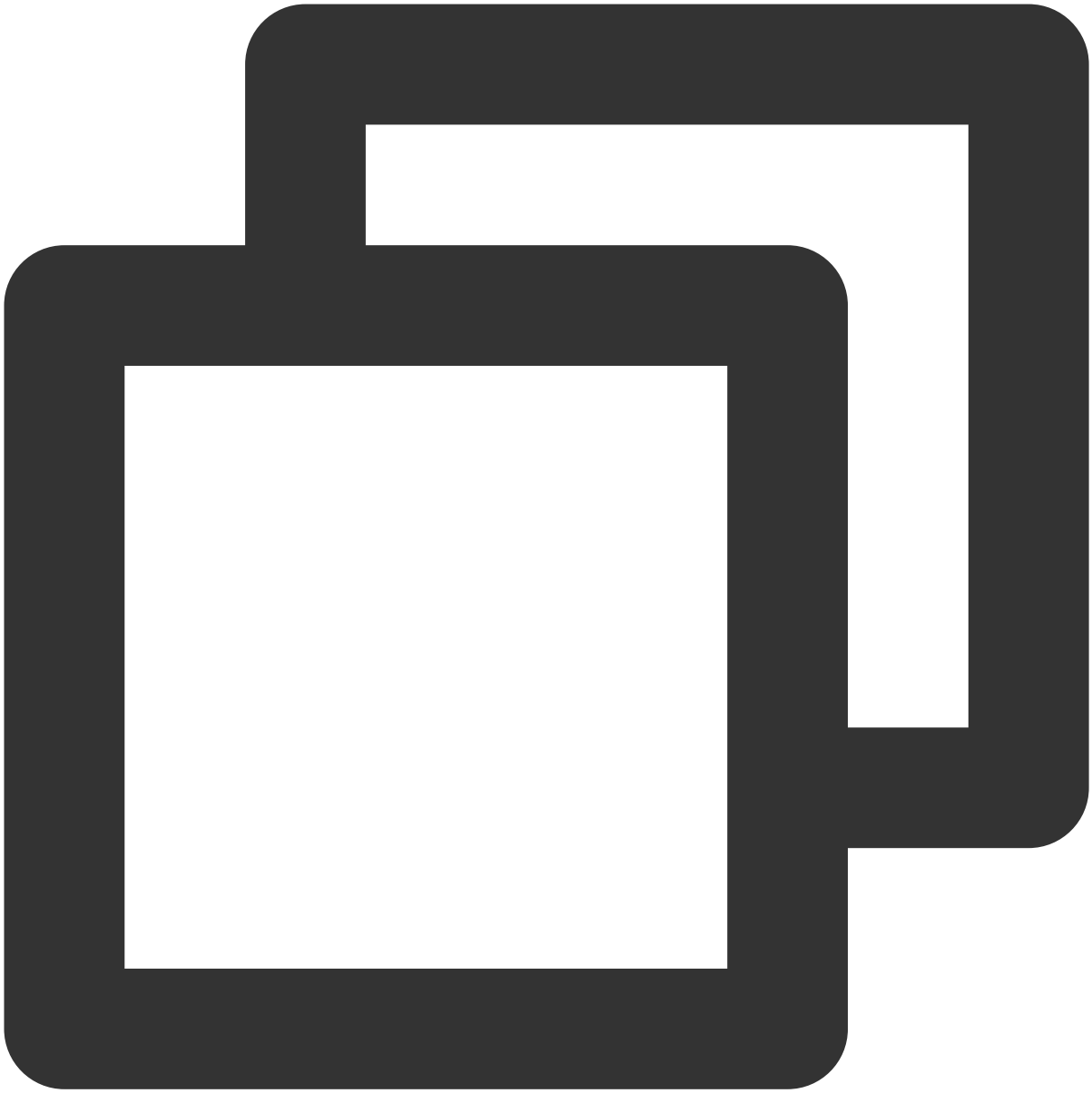


```
package main

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。

C++ SDK

最近更新时间：2024-01-03 14:27:38

操作场景

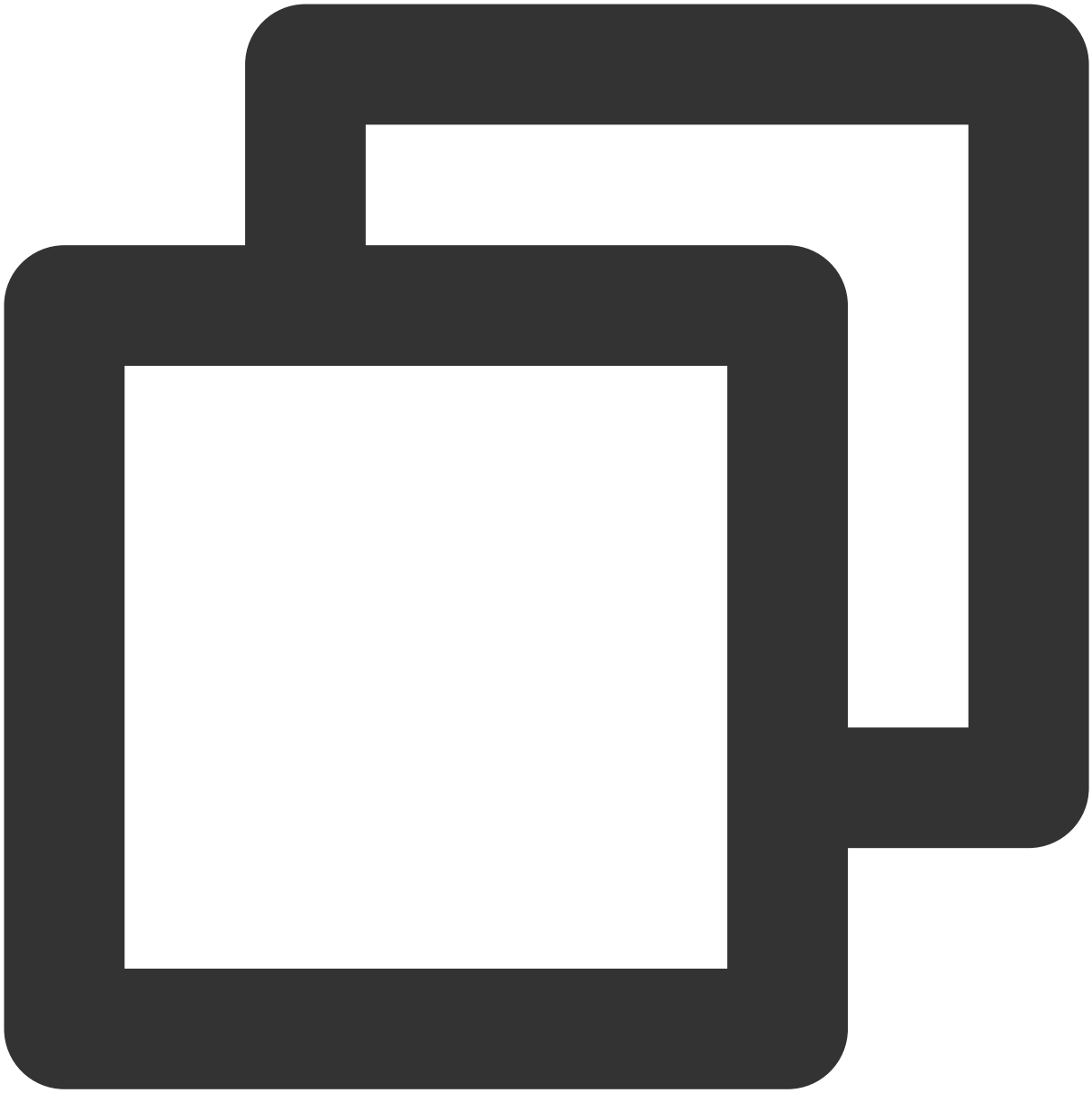
本文以调用 C++ SDK 为例介绍通过开源 SDK 实现消息收发的操作过程，帮助您更好地理解消息收发的完整过程。

前提条件

完成资源创建与准备
安装 GCC
下载 Demo

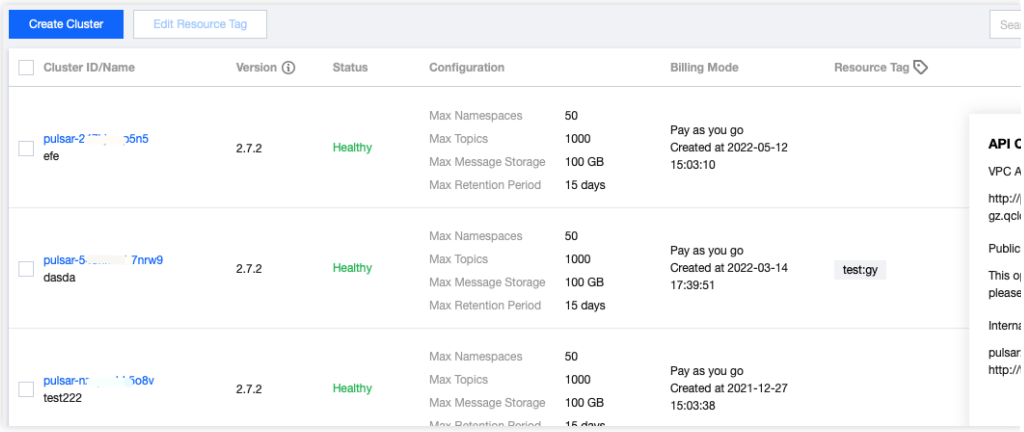
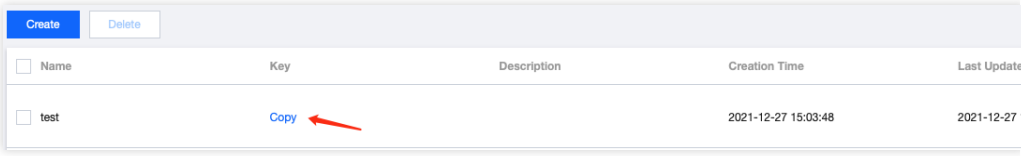
操作步骤

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。

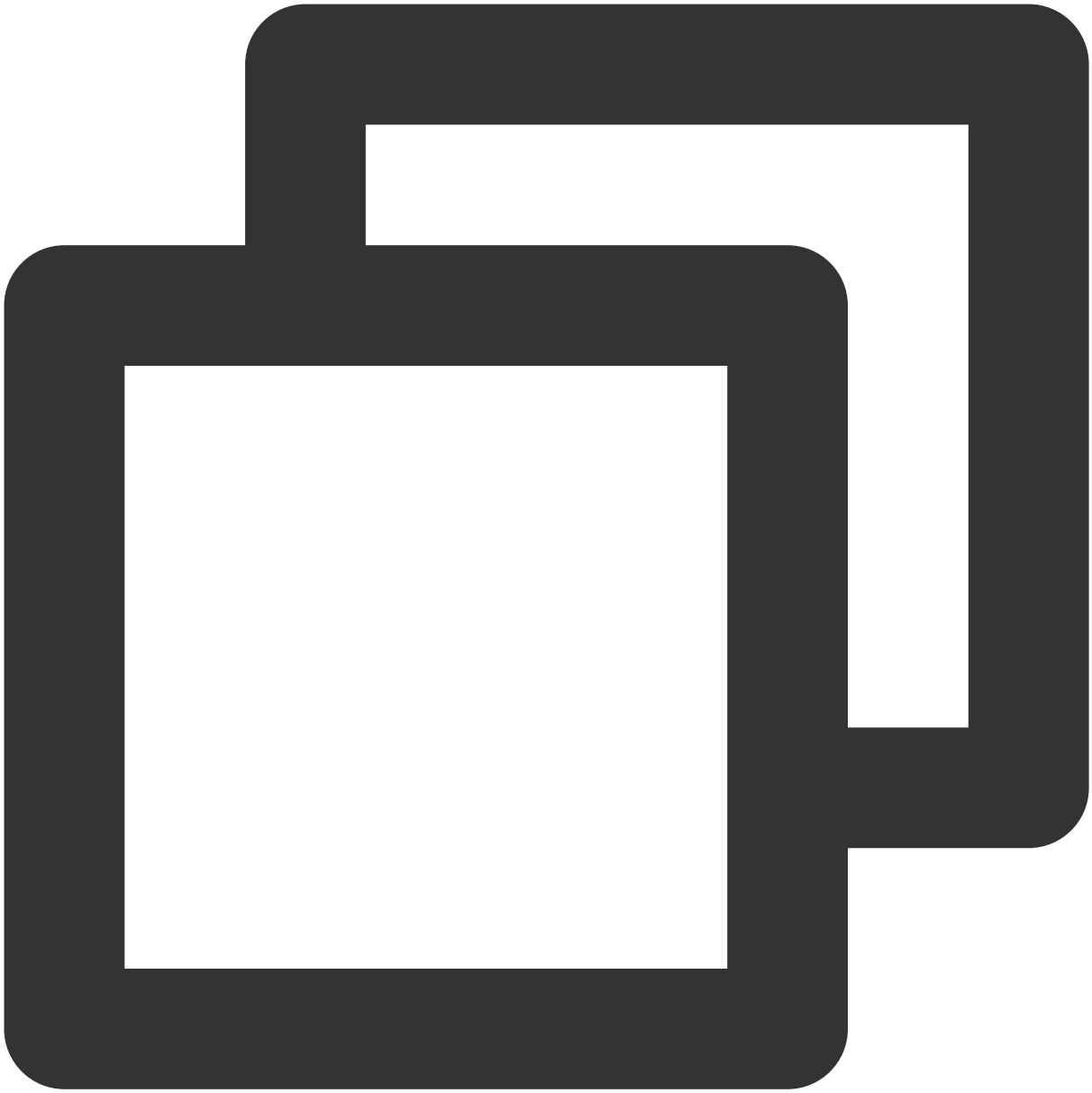


```
// 客户端配置信息
ClientConfiguration config;
// 设置授权角色密钥
AuthenticationPtr auth = pulsar::AuthToken::createWithToken(AUTHENTICATION);
config.setAuth(auth);
// 创建客户端
Client client(SERVICE_URL, config);
```

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。



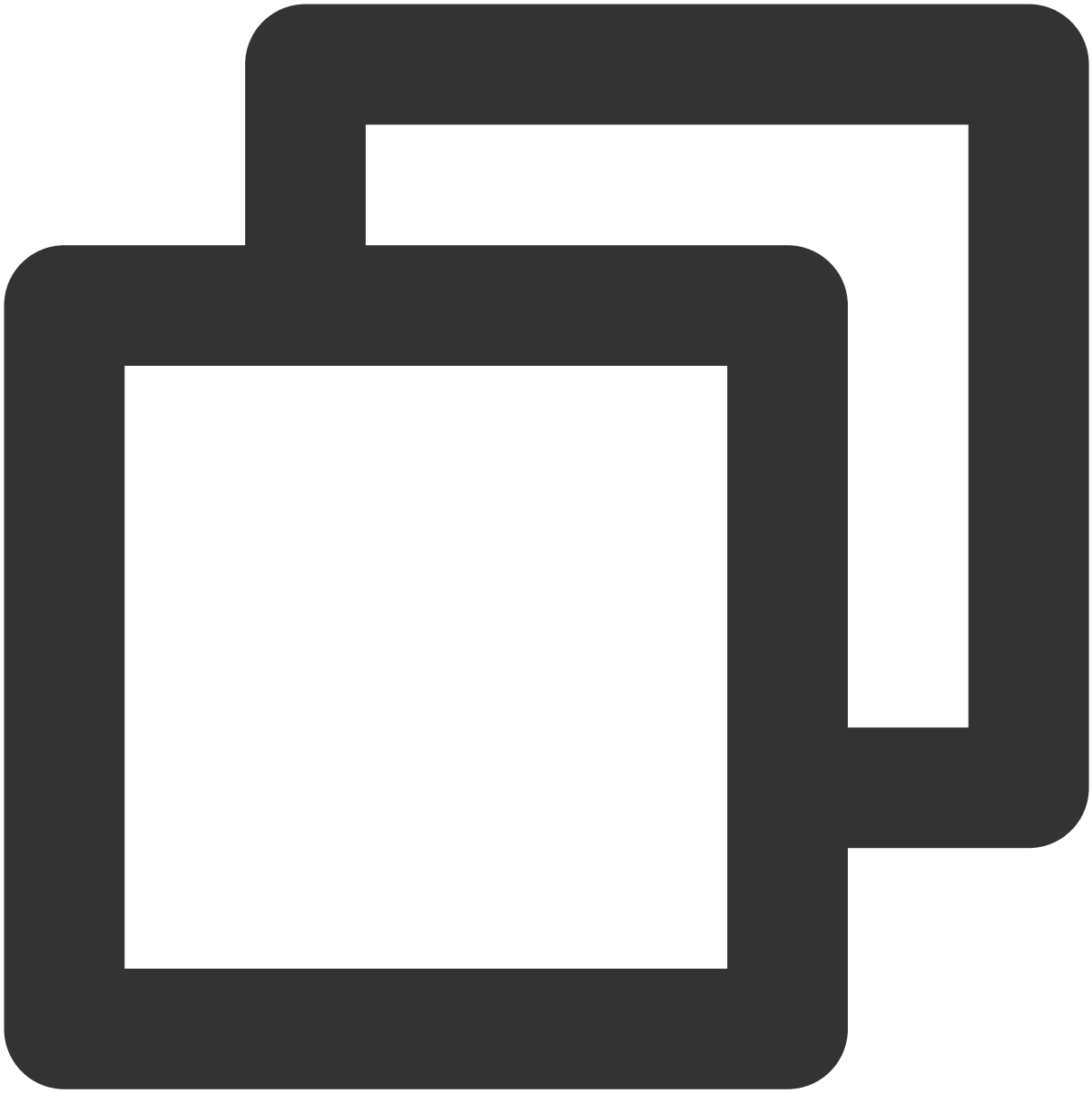
```
// 生产者配置
ProducerConfiguration producerConf;
producerConf.setBlockIfQueueFull(true);
producerConf.setSendTimeout(5000);
// 生产者
Producer producer;
// 创建生产者
Result result = client.createProducer(
    // topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称
    "persistent://pulsar-xxx/sdk_cpp/topic1",
    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。

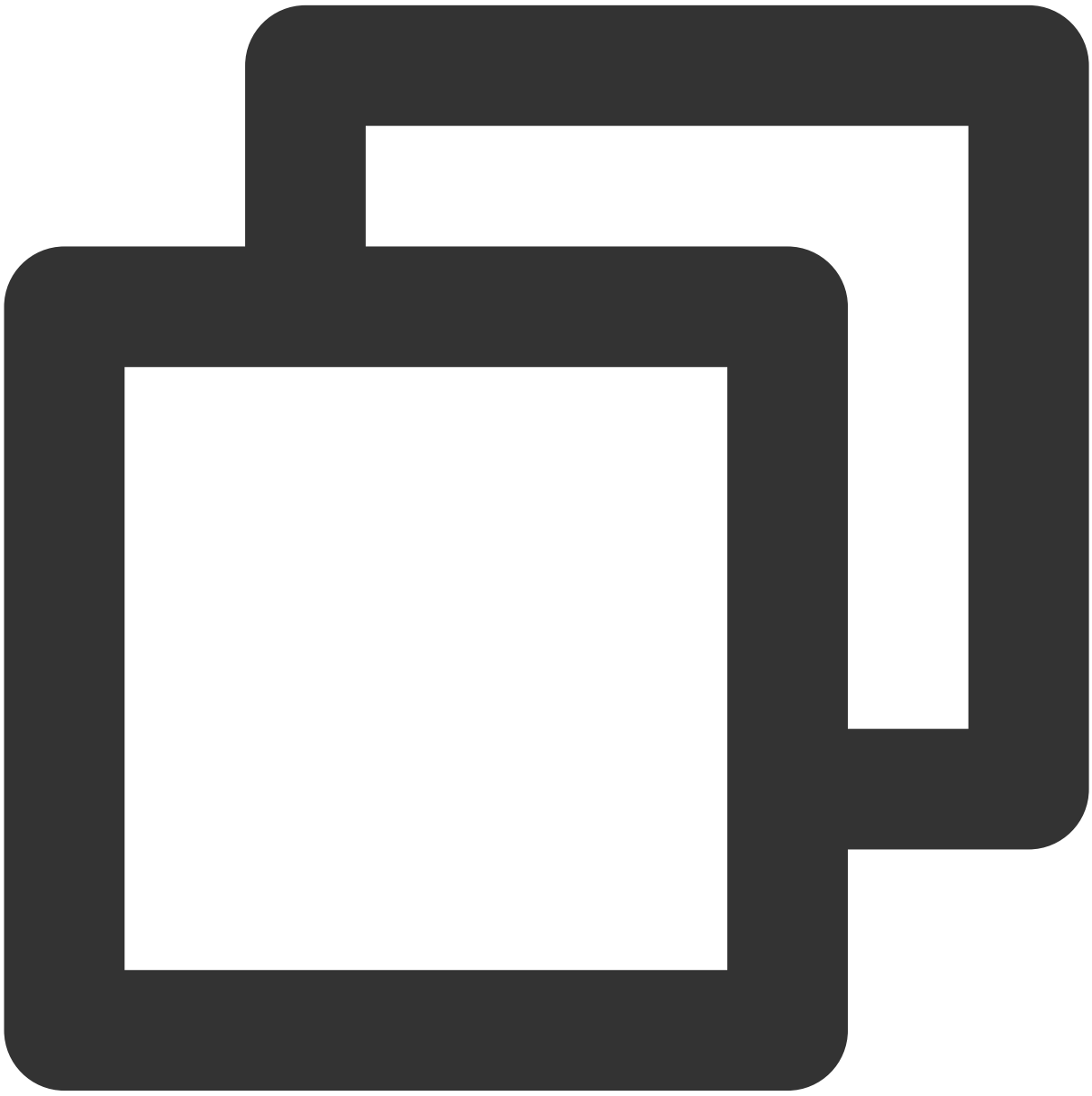
3. 发送消息。



```
// 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
    } else {
        // 发送成功
        std::cout << "The message " << content << " sent successfully" << std::endl;
    }
```

4. 创建消费者。

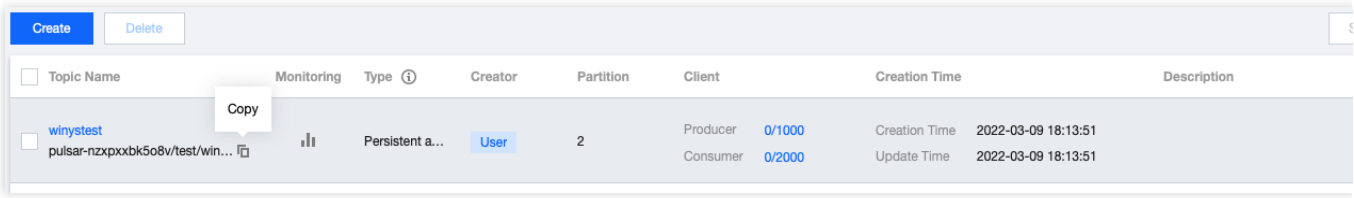


```
    // 消费者配置信息
    ConsumerConfiguration consumerConfiguration;
    consumerConfiguration.setSubscriptionInitialPosition(pulsar::InitialPositionEarl
```

```
    // 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;
    }
```

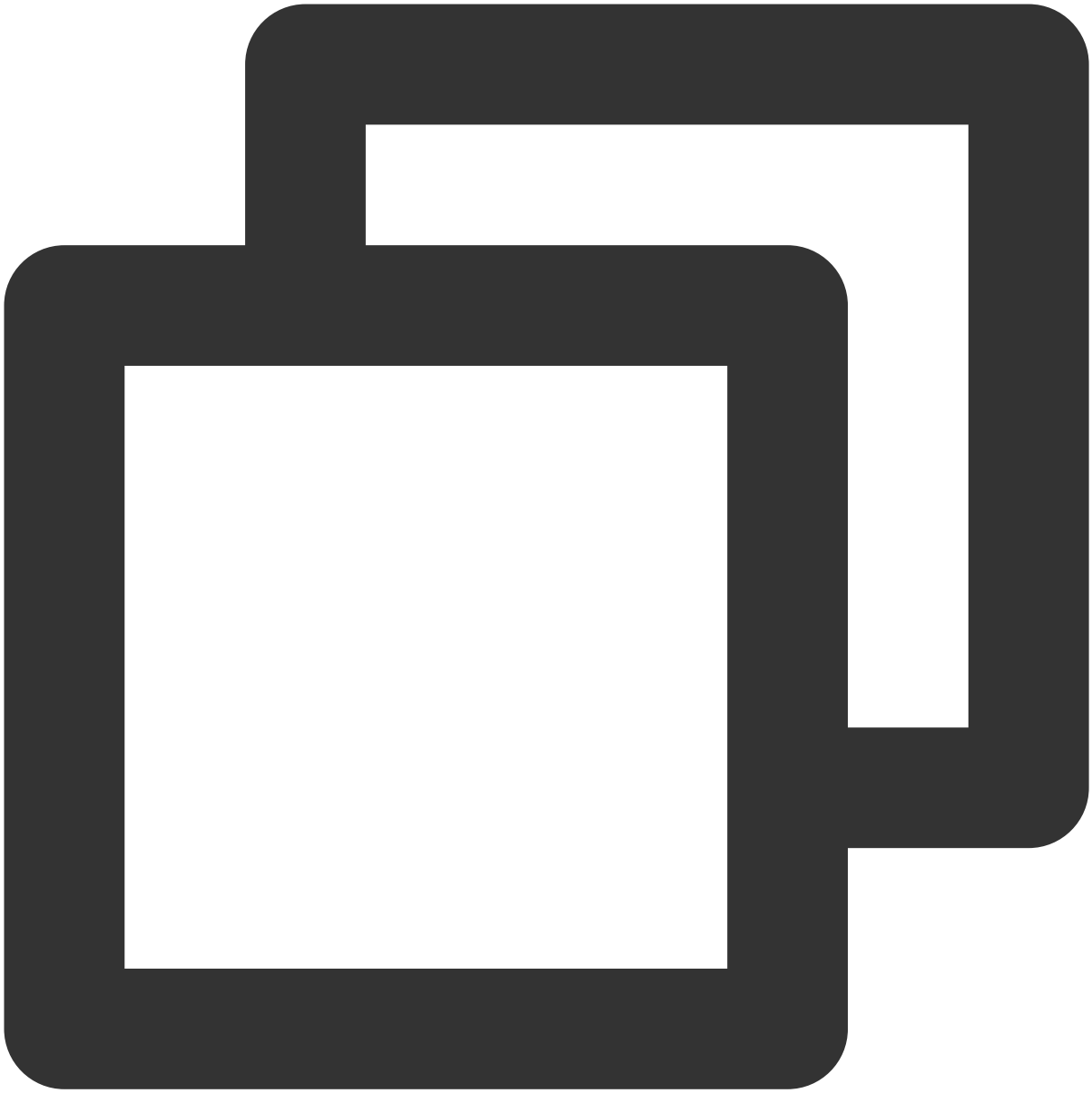
说明：

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。



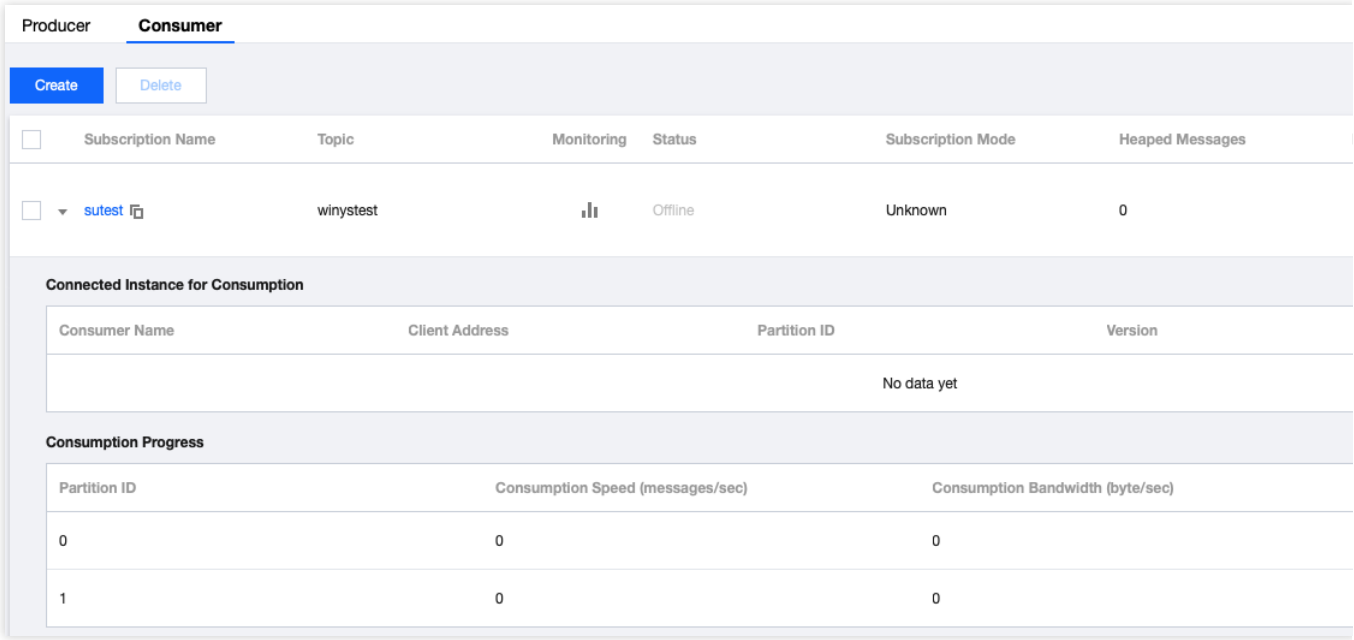
subscriptionName 需要写入订阅名，可在**消费管理**界面查看。

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()
// 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：2024-01-03 14:27:38

操作场景

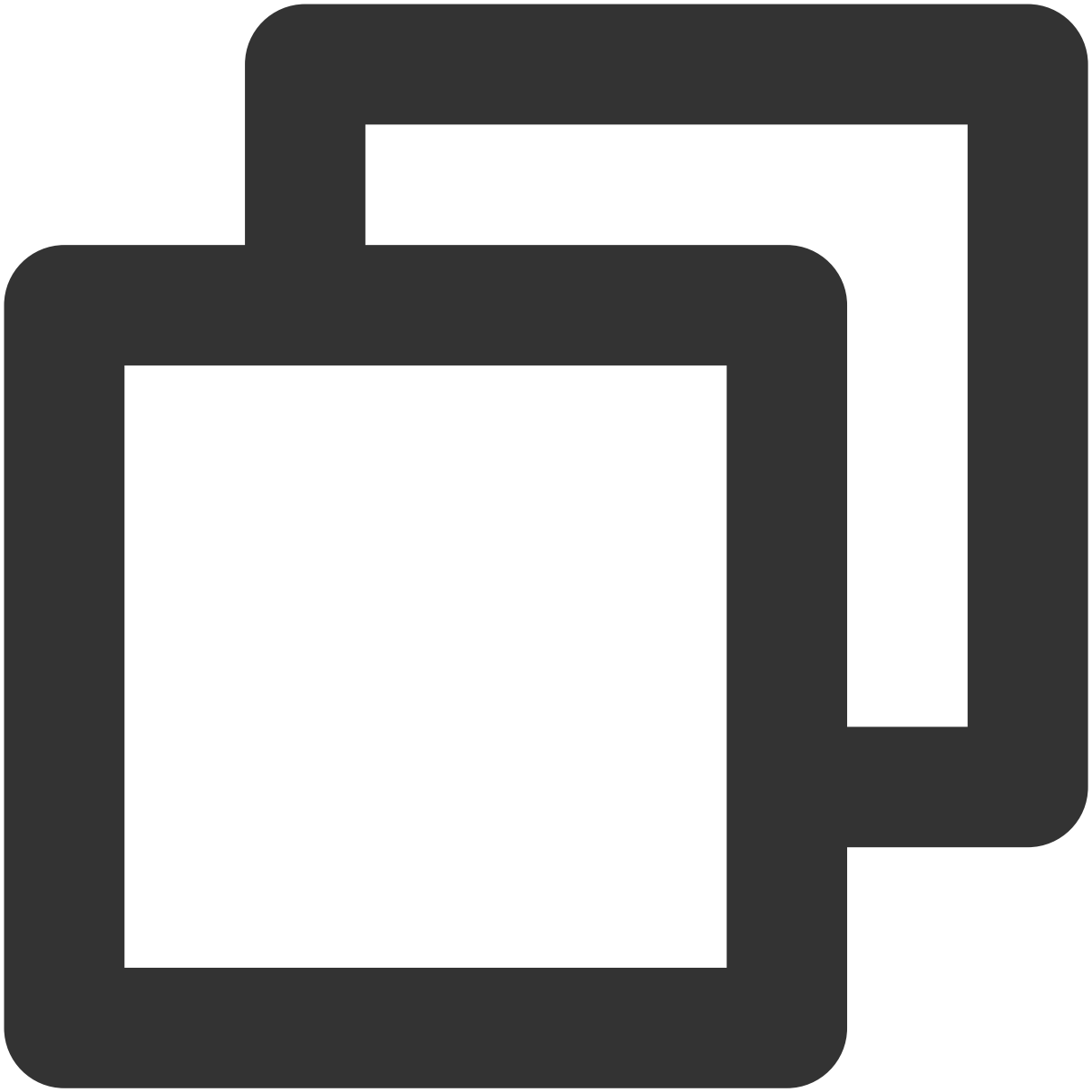
本文以调用 Python SDK 为例介绍通过开源 SDK 实现消息收发的操作过程，帮助您更好地理解消息收发的完整过程。

前提条件

完成资源创建与准备
安装 Python
安装 pip
下载 Demo

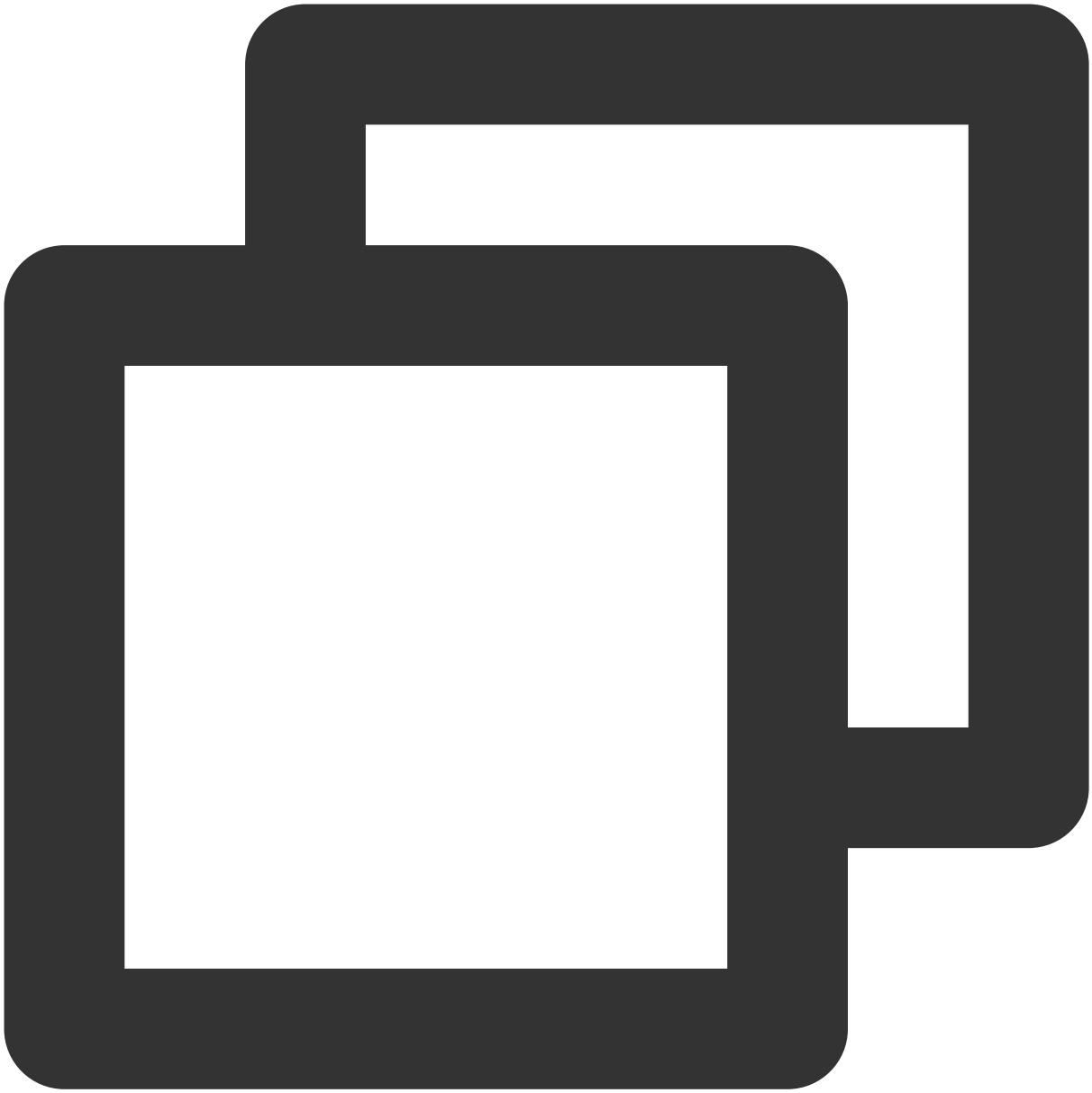
操作步骤

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。



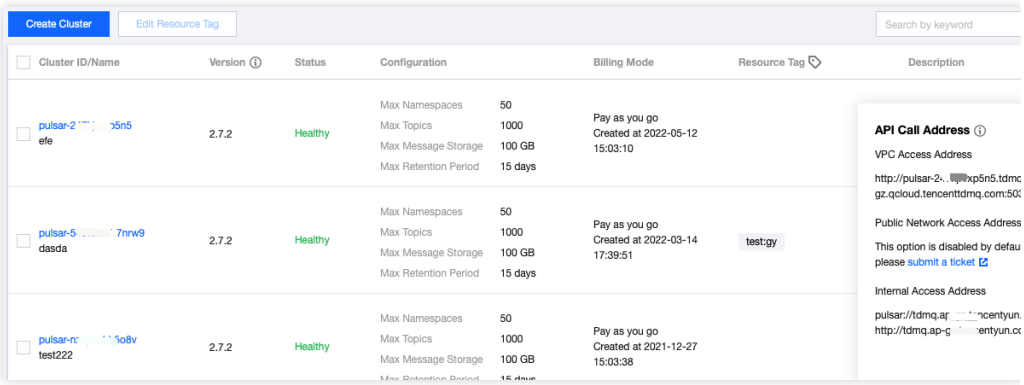
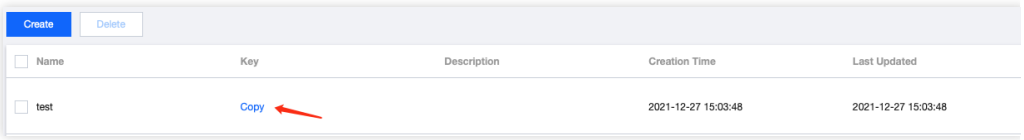
```
pip install 'pulsar-client==3.1.0'
```

2. 创建客户端。

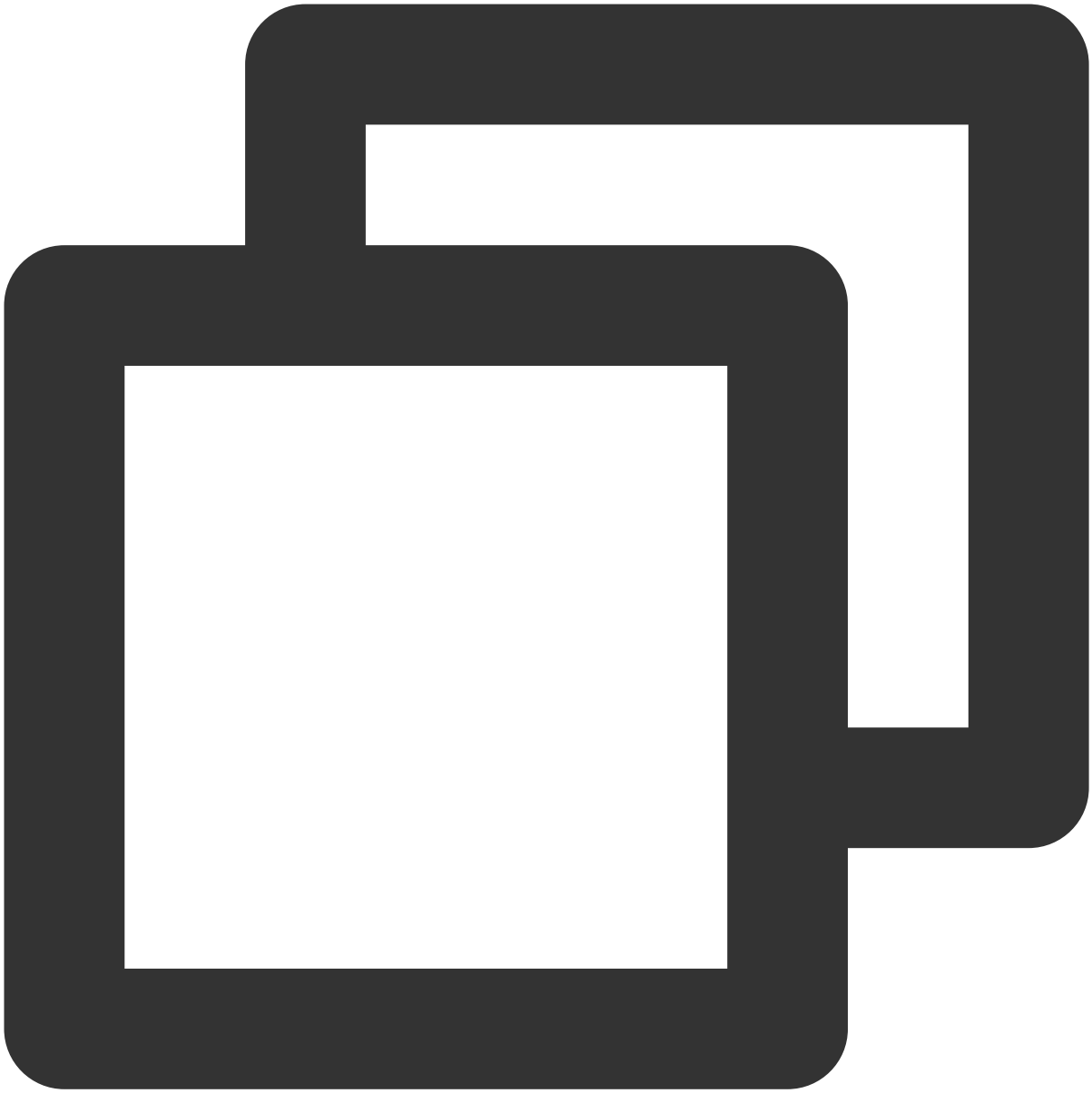


```
# 创建客户端
client = pulsar.Client(
    authentication=pulsar.AuthenticationToken(
        # 已授权角色密钥
        AUTHENTICATION),
    # 服务接入地址
    service_url=SERVICE_URL)
```

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。

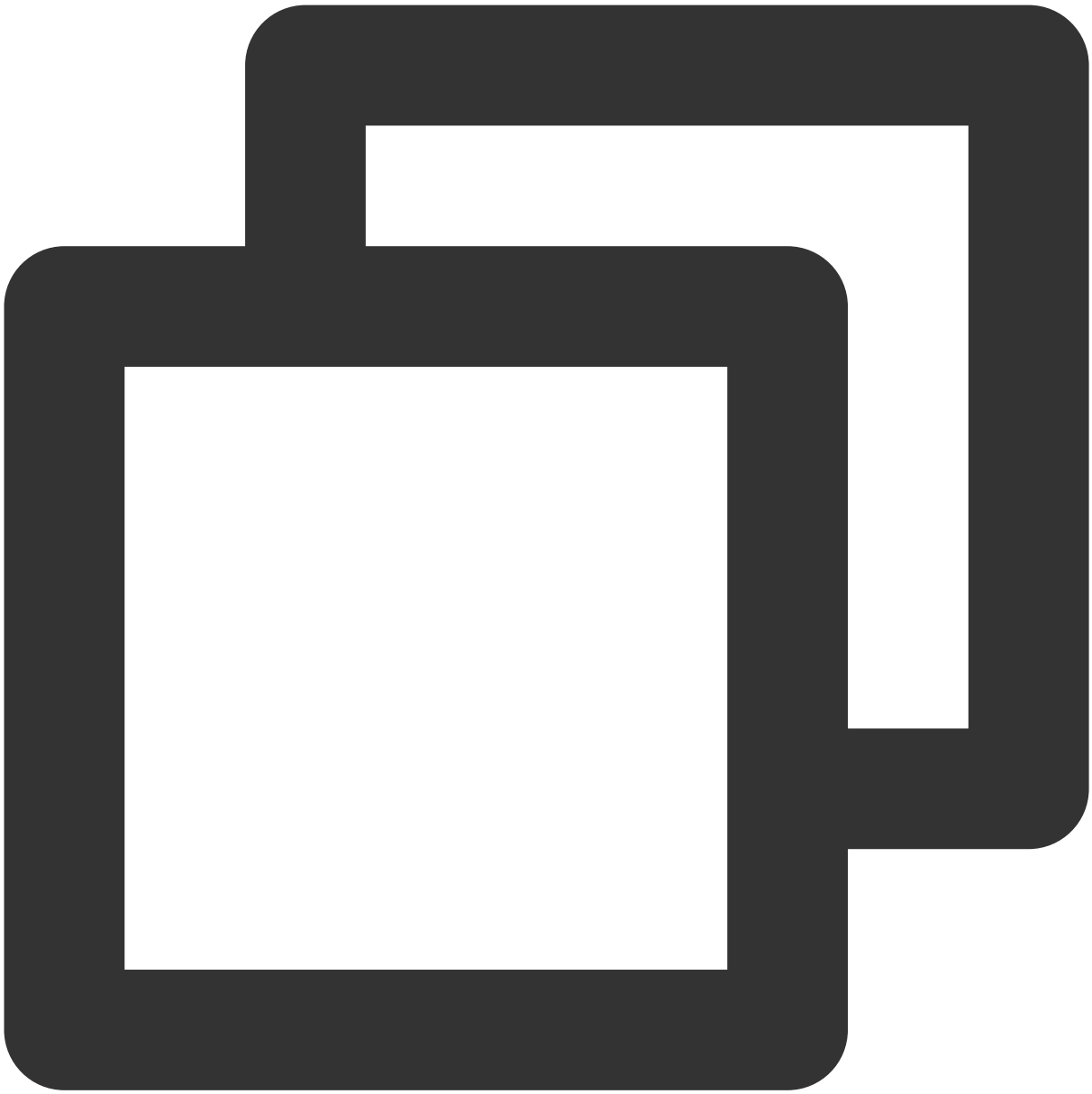


```
# 创建生产者
producer = client.create_producer(
    # topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管理】处
    topic='pulsar-xxx/sdk_python/topic1'
)
```

说明：

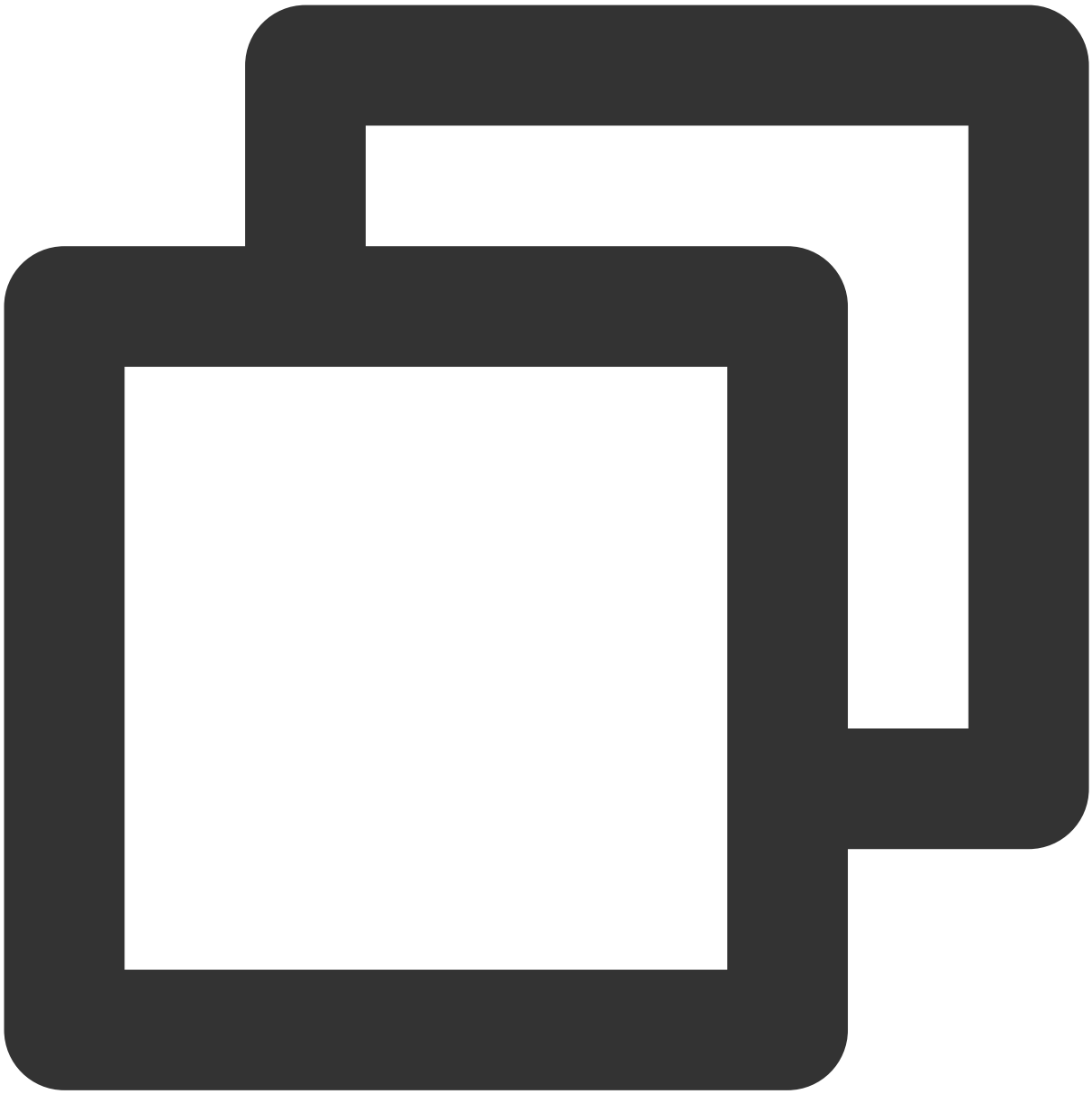
Topic 名称需要填入完整路径，即 `persistent://clusterid/namespace/Topic` ， `clusterid/namespace/topic` 的部分可以从控制台上 Topic管理 页面直接复制。

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'
)
```

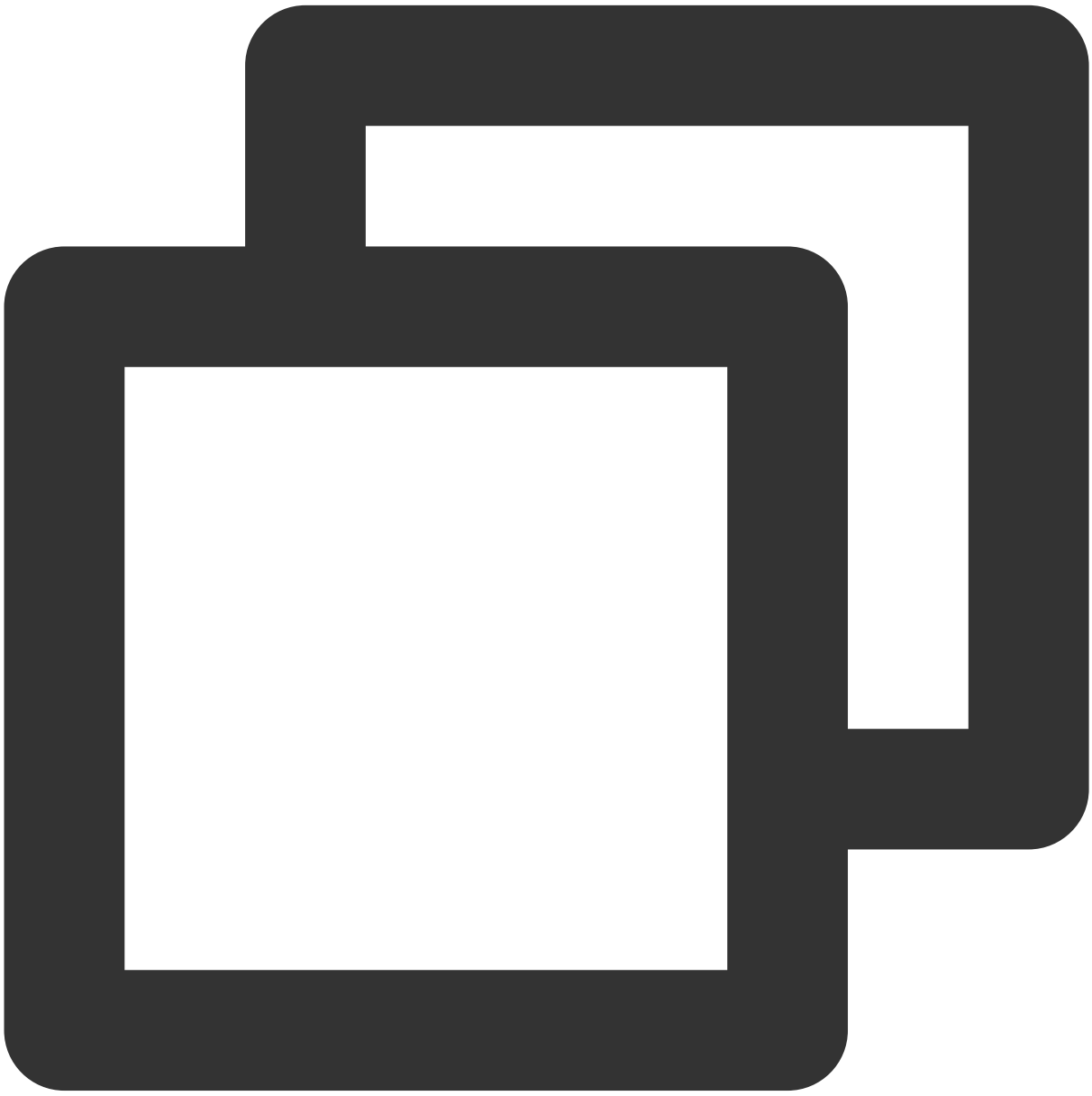
还可以使用异步方式发送消息。



```
# 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。

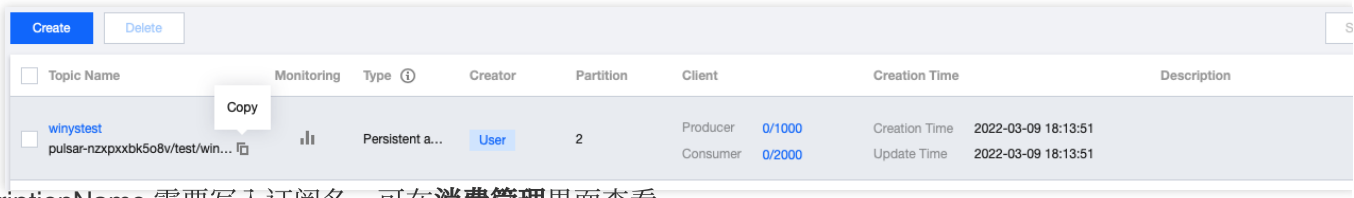


```
    # 订阅消息
    consumer = client.subscribe(
        # topic完整路径，格式为persistent://集群（租户）ID/命名空间/Topic名称，从【Topic管理】处
```

```
        topic='pulsar-xxx/sdk_python/topic1',
        # 订阅名称
        subscription_name='sub_topic1'
    )
```

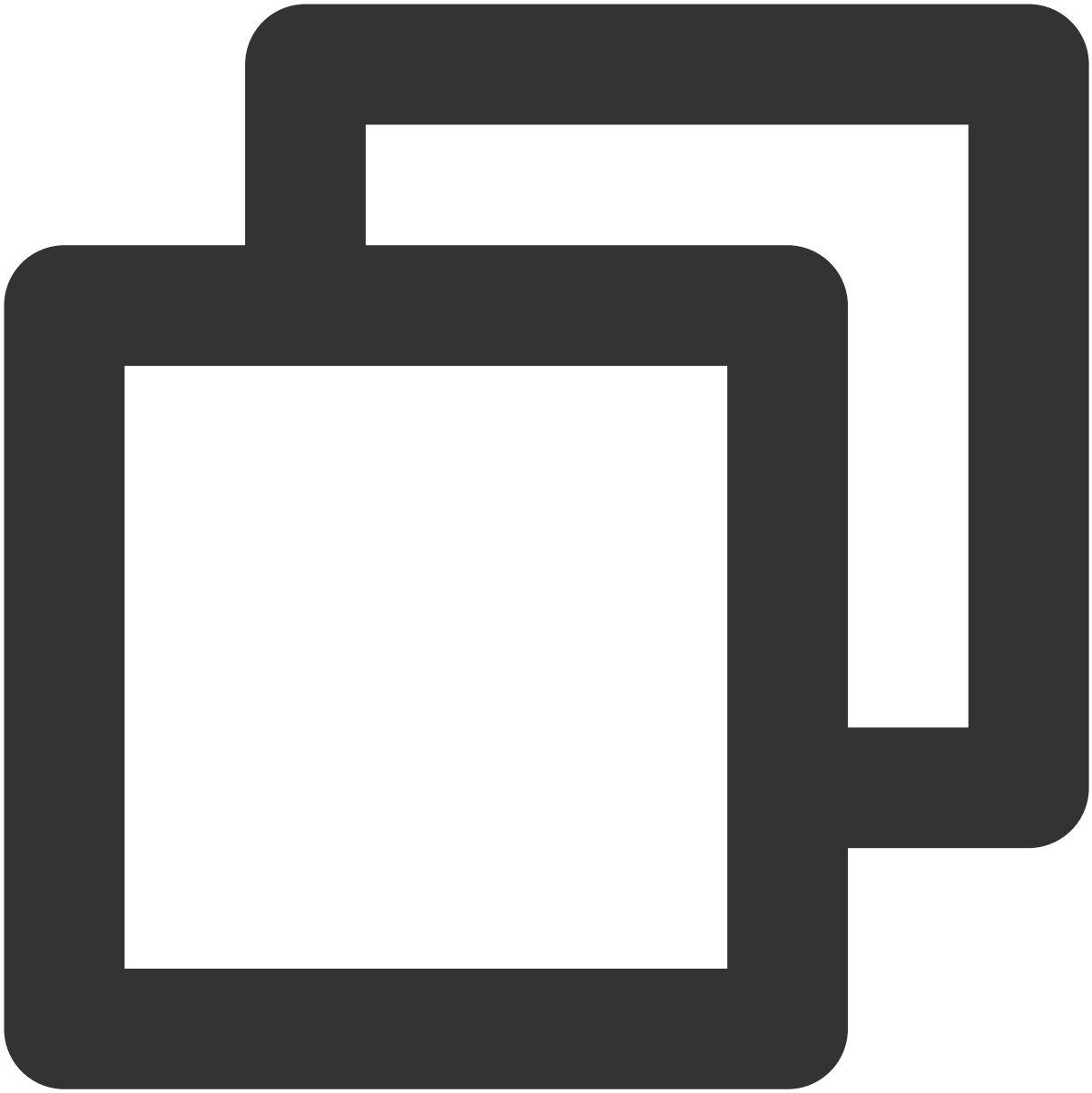
说明：

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。



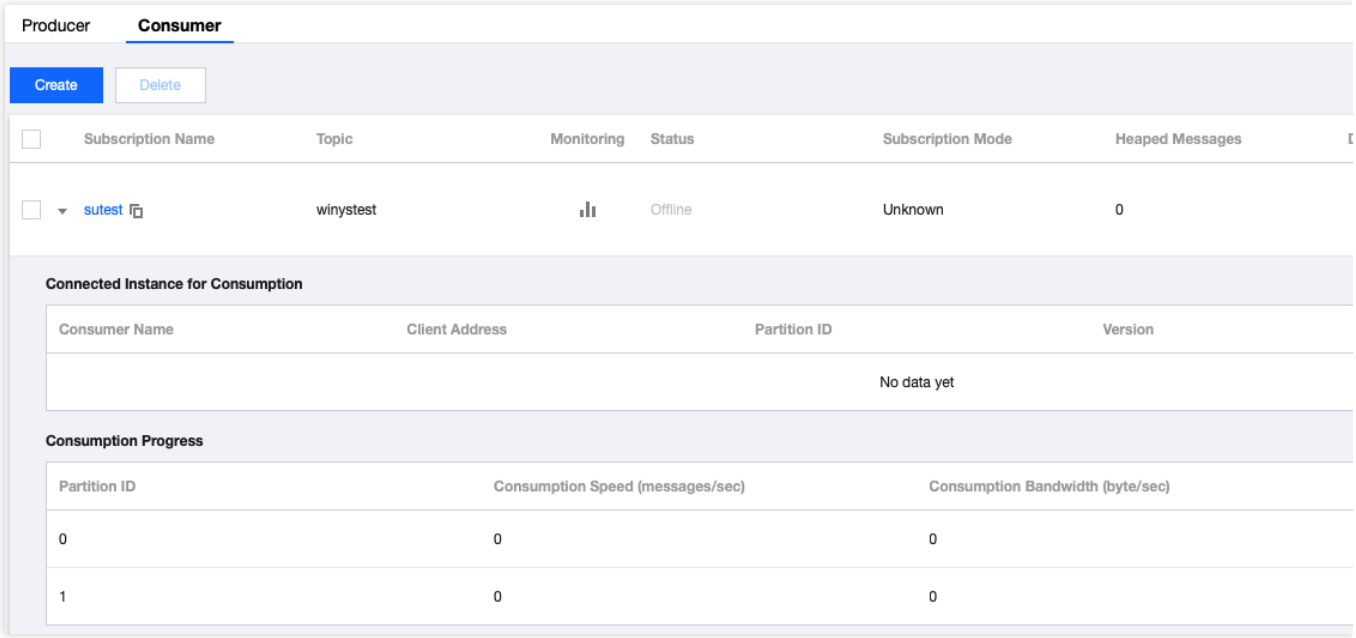
subscriptionName 需要写入订阅名，可在**消费管理**界面查看。

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（社区版）

最近更新时间：2024-01-03 14:27:38

操作场景

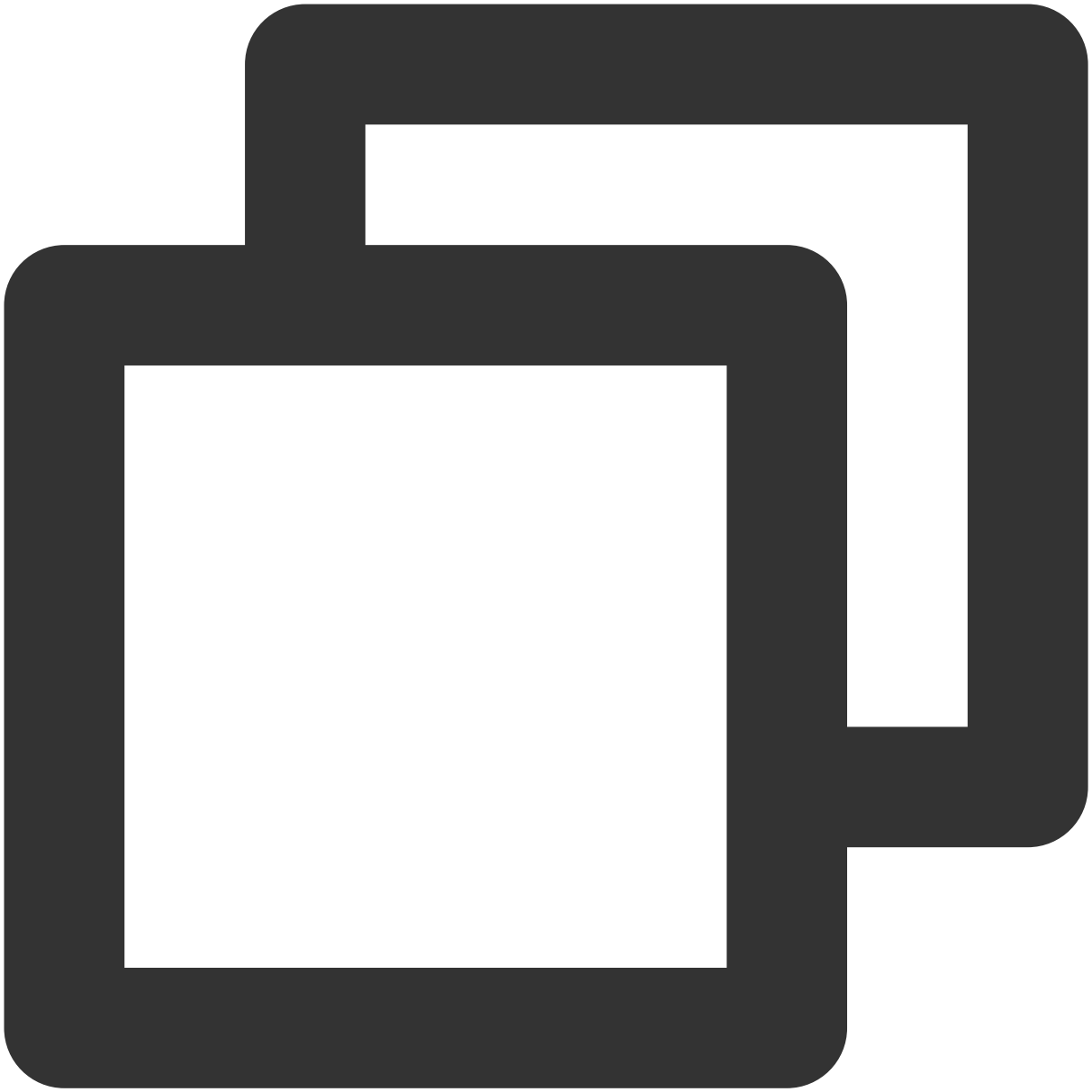
TDMQ Pulsar 版2.7.1及以上版本的集群已支持 Pulsar 社区版 Node.js SDK。本文介绍如何使用 Pulsar 社区版 Node.js SDK 完成接入。

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）。

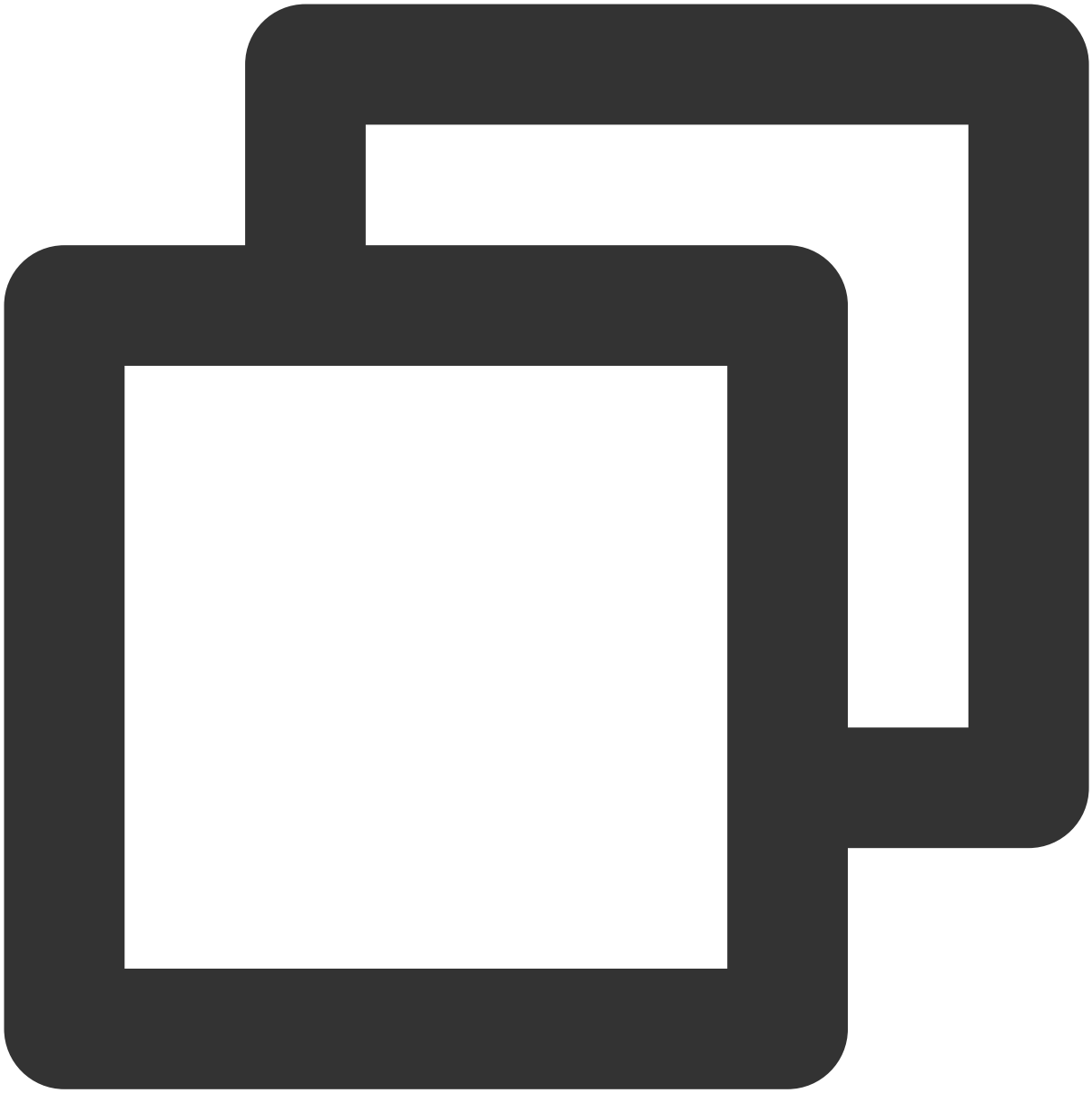
操作步骤

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:    Pulsar.NewAuthenticationToken("eyJh**"), //更换为密钥
  });

  await client.close();
})();
```

关于 Pulsar 社区版 Node.js SDK 各种功能的使用方式，请参考 Pulsar 官方文档。