消息队列 TDMQ

SDK 文档

产品文档

腾讯云

【版权声明】

【商标声明】

【服务声明】

本文档意在向您介绍腾讯云全部或部分产品、服务的当时的相关概况，部分产品、服务的内容可能不时有所调整。您所购买的腾讯云产品、服务的种类、服务标准等应由您与腾讯云之间的商业合同约定，除非双方另有约定，否则，腾讯云对本文档内容不做任何明示或默示的承诺或保证。

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100。

文档目录

SDK 文档
SDK 升级说明

最近更新时间：2020-09-17 12:52:23

注意：
TDMQ 内测版本将于2020年9月17日上午09:00~12:00进行停服升级，升级后需要用户做少量改造重新接入。
升级之后，我们会完整帮您迁移数据，包括您创建的环境和 Topic 以及 VPC 接入点等数据，您无需担心基础数据的丢失。

操作场景

本文档适用于2020年9月17日升级后进行参考和操作。
本文主要介绍新版 TDMQ SDK 升级后如何快速恢复您原先业务代码的接入。

操作步骤

步骤1：创建角色并授权

1. 登录 TDMQ 控制台，在左侧导航栏单击【角色管理】，进入角色管理页面。

2. 在角色管理页面，选择地域后，单击【新建】进行新建角色。

3. 在左侧导航栏单击【环境管理】，在所需环境上单击【配置权限】，配置刚刚创建的角色的权限。

步骤2：配置角色密钥

Java 客户端

当前由于 Pulsar 官方只有 Java 客户端提供了带有 listenerName 参数的最新版本，TDMQ 此次更新需要依赖于 listenerName 参数，所以截止目前只有 Java 客户端可以直接从官网下载。

1. 按照 Pulsar 官方文档 更新依赖。

```
<!-- in your <properties> block -->
<pulsar.version>2.6.1</pulsar.version>

<!-- in your <dependencies> block -->
<dependency>
<groupId>org.apache.pulsar</groupId>
<artifactId>pulsar-client</artifactId>
<version>${pulsar.version}</version>
</dependency>
```

2. 前往 TDMQ 控制台【角色管理】，找到刚刚创建的角色，单击复制密钥。

3. 在创建 Client 的代码中加入刚刚复制的密钥，并添加 `listenerName` 参数

```
PulsarClient client = PulsarClient.builder()
.serviceUrl("pulsar://*.*.*.*:6000/")
.listenerName("1300*****0/vpc-******/subnet-*********")
.authentication(AuthenticationFactory.token("eyJh****"))
.build();
```

> 说明：
> `listenerName` 即"custom:"拼接原先的路由 ID（NetModel），路由 ID 可以在控制台【环境管理】接入点查看并复制。

Go 客户端

Go 客户端 Pulsar 官方目前还未更新最新适配的客户端，在官方适配之前需要下载腾讯云提供的 SDK。

1. 下载 SDK。

```
$ go get -u "github.com/TencentCloud/tdmq-go-client/pulsar@v0.2.0-beta.1"
```

2. 在代码中重新导入。

```
import "github.com/TencentCloud/tdmq-go-client/pulsar"
```

3. 前往 TDMQ 控制台【角色管理】，找到刚刚创建的角色，单击复制密钥。
4. 在创建 Client 的代码中加入刚刚复制的密钥，并添加 ListenerName 参数

```
client, err := NewClient(ClientOptions{
URL: "pulsar://*.*.*.*:6000",
ListenerName: "custom:1300*****0/vpc-******/subnet-********",
Authentication: NewAuthenticationToken("eyJh****"),
})
```

说明：
listenerName 即“custom:”拼接原先的路由 ID（ NetModel ），路由 ID 可以在控制台【环境管理】接入点查看并复制。

步骤3：部署客户端

在修改完客户端代码后，您需要重新将客户端部署到原先的环境进行验证和生产。

Go SDK

最近更新时间：2020-09-17 18:11:53

操作场景

TDMQ 提供了 Go 语言的 SDK 来调用服务，进行消息队列的生产和消费。

本文主要介绍 Go SDK 的使用方式，提供 Demo 工程的环境配置、下载、代码编写及运行示例，帮助工程师快速搭建 TDMQ 测试工程。

前提条件

- 已经在本地安装 Golang 开发环境（下载地址）。
- 已经准备好 Go 1.11+ 的部署环境（云服务器或其他云资源），且该环境所在的 VPC 已接入TDMQ（参考 VPC 接入指南）。
- 已获取调用地址（URL）和路由 ID（NetModel）。
  这两个参数均可以在【环境管理】的接入点列表中获取。请根据客户端部署的云服务器或其他资源所在的私有网络选择正确的接入点来复制参数信息，否则会有无法连接的问题。

- 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

准备 Demo 环境

1. 安装 IDE
   您可以 安装 GoLand 或其它的 Go IDE 运行这个 Demo，直接通过 `go run` 执行也可以。

2. 配置 GCC 环境
因为现在的 SDK 依赖了 CGO 的库，所以需要本地配置64位 GCC，可以通过 MinGW 来安装。

3. 打开命令控制台，运行以下命令：

```
go get -u "github.com/TencentCloud/tdmq-go-client/pulsar@v0.2.0-beta.1"
```

如果国内网络环境下载比较慢，可以通过配置 Go Proxy 来解决。
如果处于无法连接外网的环境下，需要先行下载依赖文件 压缩包，将压缩包里的文件放在 %GOPATH/pkg/mod/cache/download 文件夹下即可， %GOPATH 可通过如下指令获取：

```
# linux
go env | grep GOPATH

# Windows
go env | findstr GOPATH
```

说明：
目前 Pulsar 官方尚未更新最新适配的客户端，腾讯云已将适配后的 Go 客户端提交至社区，即将在下个版本发布，在官方适配之前您需要先使用腾讯云提供的 SDK。

创建 Demo工程

1.使用 IDE 创建一个新工程，在文件夹中创建 go.mod 文件并编辑如下：

```
module example/godemo

go 1.12

require github.com/TencentCloud/tdmq-go-client v0.1.1
```

上述 v0.1.1是 GO SDK 的版本，云上资源环境中下载的依赖文件压缩包也需要是同样的版本。

2.创建 producer.go 和 consumer.go 测试 Demo 文件。

- producer.go 代码内容如下，其中 listenerName 即 custom: 拼接路由 ID（NetModel），路由 ID 可以在控制台【环境管理】接入点查看并复制， NewAuthenticationToken 即角色密钥，可以在【角色管理】页面复制。

```
package main

import (
"context"
"github.com/TencentCloud/tdmq-go-client/pulsar"
"log"
"strconv"
)

func main() {

client, err := NewClient(ClientOptions{
URL: "pulsar://*.*.*.*:6000",
ListenerName: "custom:1300*****0/vpc-******/subnet-********",
Authentication: NewAuthenticationToken("eyJh****"),
})
if err != nil {
log.Fatal(err)
}
defer client.Close()

producer, err := client.CreateProducer(pulsar.ProducerOptions{
DisableBatching: true,
Topic: "persistent://appid/namespace/topic-1",
})
if err != nil {
log.Fatal(err)
}
defer producer.Close()

ctx := context.Background()

for j := 0; j < 10; j++ {
if msgId, err := producer.Send(ctx, &pulsar.ProducerMessage{
Payload: []byte("Hello " + strconv.Itoa(j)),
}); err != nil {
log.Fatal(err)
} else {
log.Println("Published message: ", msgId)
}
}
}
```

其中 Topic 名称需要填入完整路径，即 persistent://appid/environment/Topic 的组合，其中 appid/environment/topic 的部分可以从控制台【Topic管理】页面直接复制。

- consumer.go 的代码内容如下：

```
package main

import (
"context"
"fmt"
"github.com/TencentCloud/tdmq-go-client/pulsar"
"log"
)

func main() {

client, err := pulsar.NewClient(pulsar.ClientOptions{
URL: "pulsar://10.*.*.*:6000",//更换为接入点地址
ListenerName: "custom:1300*****0/vpc-******/subnet-********",
Authentication: NewAuthenticationToken("eyJh****"),
})
if err != nil {
log.Fatal(err)
}
defer client.Close()

consumer, err := client.Subscribe(pulsar.ConsumerOptions{
Topics: []string{"persistent://appid/namespace/topic-1"},
SubscriptionName: "my-sub",
Type: pulsar.Shared,
})
if err != nil {
log.Fatal(err)
```

```
}
defer consumer.Close()

for ; ; {
msg, err := consumer.Receive(context.Background())
if err != nil {
log.Fatal(err)
}
fmt.Printf("Received message msgId: %#v -- content: '%s' -- topic : '%v'\n",
msg.ID(), string(msg.Payload()), msg.Topic())
}
}
```

测试验证

您需要先将 Demo 代码打包上传到云服务器上，且确认该云服务器所在的私有网络 VPC 和 TDMQ 中配置的接入点吻合。

接下来进入 Demo 测试，先执行 `go run` 指令启动 `consumer.go` ，命令如下：

```
go run consumer.go
```

再类似启动 producer.go，观察控制台消息：

在 producer.go 运行的控制台可以看到有10条消息发送成功：

```
2020/02/20 20:20:20 Published message: &{581 0 0 0 <nil> <nil>}
2020/02/20 20:20:20 Published message: &{581 1 0 0 <nil> <nil>}
2020/02/20 20:20:20 Published message: &{581 2 0 0 <nil> <nil>}
2020/02/20 20:20:20 Published message: &{581 3 0 0 <nil> <nil>}
2020/02/20 20:20:21 Published message: &{581 4 0 0 <nil> <nil>}
2020/02/20 20:20:21 Published message: &{581 5 0 0 <nil> <nil>}
2020/02/20 20:20:21 Published message: &{581 6 0 0 <nil> <nil>}
2020/02/20 20:20:21 Published message: &{581 7 0 0 <nil> <nil>}
2020/02/20 20:20:21 Published message: &{581 8 0 0 <nil> <nil>}
2020/02/20 20:20:22 Published message: &{581 9 0 0 <nil> <nil>}
```

在 consumer.go 运行的控制台可以看到消息被成功接收并打印出来：

```
Received message msgId: &pulsar.messageID{ledgerID:581, entryID:0, batchIdx:0, partitionIdx:0, tracke
r:(*pulsar.ackTracker)(nil), consumer:(*pulsar.partitionConsumer)(0xc000198000)} -- content: 'Hello 0'
-- topic : 'persistent://appid/namespace/topic-1'

Received message msgId: &pulsar.messageID{ledgerID:581, entryID:1, batchIdx:0, partitionIdx:0, tracke
r:(*pulsar.ackTracker)(nil), consumer:(*pulsar.partitionConsumer)(0xc000198000)} -- content: 'Hello 1'
```

```
-- topic : 'persistent://appid/namespace/topic-1'

Received message msgId: &pulsar.messageID{ledgerID:581, entryID:2, batchIdx:0, partitionIdx:0, tracke
r:(*pulsar.ackTracker)(nil), consumer:(*pulsar.partitionConsumer)(0xc000198000)} -- content: 'Hello 2'
-- topic : 'persistent://appid/namespace/topic-1'

Received message msgId: &pulsar.messageID{ledgerID:581, entryID:3, batchIdx:0, partitionIdx:0, tracke
r:(*pulsar.ackTracker)(nil), consumer:(*pulsar.partitionConsumer)(0xc000198000)} -- content: 'Hello 3'
-- topic : 'persistent://appid/namespace/topic-1'

...//后续省略
```

则 Go 版本的 SDK Demo 运行成功。

Tag 功能

为了配置 Go SDK 的 Tag 支持，需要在消费者订阅的时候配置相应的 Tag 参数，如下：

```
consumer, err := client.Subscribe(pulsar.ConsumerOptions{
Topics: []string{"persistent://appid/namespace/topic-1"},
SubscriptionName: "my-sub",
Type: pulsar.Shared,
TagMapTopicNames: map[string]string{"persistent://appid/namespace/topic-1":"a||b"},
})
if err != nil {
log.Fatal(err)
}
defer consumer.Close()
```

不同的 Tag 之间用“||”符号分隔，此时创建的 consumer 就会只消费 Tag 中包含 a 或 b 的消息，如下创建 producer 并发送消息：

```
// 创建 Producer 对象
producer, err := client.CreateProducer(pulsar.ProducerOptions{
Topic: "persistent://appid/namespace/topic-1",
})
if err != nil {
log.Fatal(err)
}
defer producer.Close()
// 发送消息
for j := 0; j < 10; j++ {
if msgId, err := producer.Send(ctx, &pulsar.ProducerMessage{
Payload: []byte("Hello " + strconv.Itoa(j)),
Tags: []string{"a","b"},
```

```
}); err != nil {
log.Fatal(err)
} else {
log.Println("Published message: ", msgId)
}
}
```

可以用 consumer 来接收消息，完成消费。

消息延迟重试

有时我们接收到一条消息时希望能在可控的延迟时间之后再次消费这个消息，这个功能现在也集成在 SDK 中，首先我们需要在创建 consumer 时配置相应的参数：

```
consumer, err := client.Subscribe(pulsar.ConsumerOptions{
Topics: []string{"persistent://appid/namespace/topic-1"},
SubscriptionName: "my-sub",
Type: pulsar.Shared,
//EnableRetry 设为 true 是必须的，否则默认关闭 Retry 功能
EnableRetry: true,
//DelayLevelUtil 不是必须配置的，系统会有缺省值
DelayLevelUtil: pulsar.NewDelayLevelUtil("1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m"),
})
if err != nil {
log.Fatal(err)
}
defer consumer.Close()
```

在消息重试的时候，我们提供了两个接口，分别是异步和同步的方式来重试，示例如下：

```
//同步的方式
err = consumer.ReconsumeLater(msg,pulsar.NewReconsumeOptionsWithLevel(2))
if err != nil{
log.Fatal(err)
}
//异步的方式，提供了一个回调方法，会在 Retry 消息发送出去后进行调用
consumer.ReconsumeLaterAsync(msg, pulsar.NewReconsumeOptionsWithLevel(2), func(id pulsar.Mes
sageID, message *pulsar.ProducerMessage, err error) {
if err != nil {
fmt.Printf("Error %v when send retry msg", err)
} else {
fmt.Printf("Retry message send success with id : %v", id)
}
})
```

Java SDK

最近更新时间：2020-09-17 18:12:18

操作场景

TDMQ 提供了 Java 语言的 SDK 来调用服务，进行消息队列的生产和消费。
本文主要介绍 Java SDK 的使用方式，提供代码编写示例，帮助工程师快速搭建 TDMQ 客户端工程。

前提条件

- 已完成 Java SDK 的下载和安装（参考 Java SDK 下载方式）。
- 已获取调用地址（URL）和路由 ID（NetModel）。
  这两个参数均可以在【环境管理】的接入点列表中获取。请根据客户端部署的云服务器或其他资源所在的私有网络选择正确的接入点来复制参数信息，否则会有无法连接的问题。

- 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

创建 Client

```
PulsarClient client = PulsarClient.builder()
.serviceUrl("pulsar://*.*.*.*:6000/")
.listenerName("1300*****0/vpc-******/subnet-********")
.authentication(AuthenticationFactory.token("eyJh****"))
.build();
```

说明：

- listenerName 即 “custom:” 拼接路由ID（ NetModel ），路由ID可以在控制台【环境管理】接入点查看并复制。
- token 即角色的密钥，角色密钥可以在【角色管理】中复制。

生产消息

创建好 Client 之后，通过创建一个 Producer，就可以生产消息到指定的 Topic 中。

```
Producer<byte[]> producer = client.newProducer().topic("persistent://1300****30/default/mytopic").c
reate();
producer.send("My message".getBytes());
```

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

这种生产方式是阻塞的方式生产消息到指定的 Topic 中，我们还可以使用异步发送的方式生产消息。

```
producer.sendAsync("my-async-message".getBytes()).thenAccept(msgId -> {
System.out.printf("Message with ID %s successfully sent", msgId);
});
```

TDMQ 的消息中除了可以保存消息体之外，还可以设置其他的消息属性。

```
producer.newMessage()
.key("my-message-key")//默认相同key路由到同一个partition中
.value("my-async-message".getBytes())
.property("my-key", "my-value")
.property("my-other-key", "my-other-value")
.send();
```

消息延迟传递：

```
producer.newMessage().deliverAfter(3L, TimeUnit.Minute).value("Hello Tdmq!").send();
```

设置消息标签（TAG）：

```
producer.newMessage()
.key("my-message-key")
.value("my-sync-message".getBytes())
//支持设置多个标签
.tags("TagA", "TagB","TagC")
.send();
```

路由模式：

模式	描述
RoundRobinPartition	如果消息没有指定 key，为了达到最大吞吐量，消息会以 round-robin 方式被路由所有分区。 请注意 round-robin 并不是作用于每条单独的消息，而是作用于延迟处理的批次边界，以确保批处理有效。 如果为消息指定了 key，发往分区的消息会被分区生产者根据 key 做 hash，然后分散到对应的分区上。 这是默认的模式。
SinglePartition	如果消息没有指定 key，生产者将会随机选择一个分区，并发送所有消息。 如果为消息指定了 key，发往分区的消息会被分区生产者根据 key 做 hash，然后分散到对应的分区上。
CustomPartition	使用自定义消息路由，可以定制消息如何进入特定的分区。 可以使用 Java client 或实现 MessageRouter 接口来实现自定义的路由模式。

顺序保证：

顺序保证	描述	路由策略与消息 Key
每个 key 分区	所有具有相同 key 的消息将按顺序排列并放置在相同的分区（Partition）中。	自同一生产者的所有消息都是有序的
同一个生产者	自同一生产者的所有消息都是有序的	自同一生产者的所有消息都是有序的

订阅消息

Consumer

通过指定 Topic 和订阅名进行消费消息

- 主动拉取

```
Consumer consumer = client.newConsumer()
.topic("persistent://1300****30/default/mytopic")
//.subscriptionType(SubscriptionType.Shared)
//.enableRetry(true)默认关闭，如果需要重试则开启
.subscriptionName("my-subscription")
.subscribe();
while (true) {
//等待接收消息
Message msg = consumer.receive();
try {
System.out.printf("Message received: %s", new String(msg.getData()));
//消息ACK
consumer.acknowledge(msg);
} catch (Exception e) {
//处理出错
// enableRetry=true 才能使用重试方法
consumer.reconsumeLater(msg, 1000L, TimeUnit.MILLISECONDS);//消息按照指定的延迟时间重试
//consumer.reconsumeLater(msg);按照延迟等级进行重试，多次重试默认自增延迟等级
//consumer.reconsumeLater(msg, 1);按照指定的延迟等级进行重试
//delayLevel =1 代表1s,
//delayLevel =2 代表5s,
//默认延迟等级"1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h"
}
}
```

- 被动接收

```
Consumer<byte[]> consumer = client.newConsumer()
.topic("persistent://1300****30/default/mytopic")
.messageListener(new MessageListener<byte[]> () {
@Override
public void received(Consumer<byte[]> consumer, Message<byte[]> msg) {
//处理消息
}
})
.subscriptionName("my-subscription")
.subscribe();
```

- 指定标签（TAG）

```
Consumer consumer = client.newConsumer()
.topicByTag(" persistent://1300****30/default/mytopic ", "TagA || TagB")
//.topic(" my-topic ", "*") 订阅所有
//.topicByTagsPattern(" my-topic ", "Tag.*")正则表达式
.subscriptionName("my-subscription")
.subscriptionType(SubscriptionType.Shared)
.subscribe();
```

异步订阅

```
CompletableFuture<Message> msg = consumer.receiveAsync();
// 处理消息
System.out.printf("Message received: %s", new String(msg.get().getData()));
```

批量订阅

```
Consumer consumer = client.newConsumer()
.topic("persistent://1300****30/default/mytopic")
.subscriptionName("my-subscription")
.batchReceivePolicy(BatchReceivePolicy.builder()
.maxNumMessages(100)
.maxNumBytes(1024 * 1024)
.timeout(200, TimeUnit.MILLISECONDS)
.build())
.subscribe();
Messages messages = consumer.batchReceive();
for (message in messages) {
// 处理消息
}
consumer.acknowledge(messages)
```

多主题订阅

- 订阅指定的 Topic 列表：

```
List<String> topics = Arrays.asList(
"persistent://1300****30/default/mytopic1",
"persistent://1300****30/default/mytopic2",
"persistent://1300****30/default/mytopic3"
);
Consumer multiTopicConsumer = consumerBuilder
```

```
.topics(topics)
.subscribe();
```

- 订阅一个 namaspace (即控制台中的环境)下的所有 Topic：

```
Pattern allTopicsInNamespace = Pattern.compile("persistent://1300****30/default/.*");
Consumer allTopicsConsumer = consumerBuilder
.topicsPattern(allTopicsInNamespace)
.subscribe();
```

- 通过一个正则表达式订阅一个 namaspace (即控制台中的环境)下匹配的 Topic：

```
Pattern someTopicsInNamespace = Pattern.compile("persistent://1300****30/default/foo.*");
Consumer allTopicsConsumer = consumerBuilder
.topicsPattern(someTopicsInNamespace)
.subscribe();
```

> 注意：
> 这种方式只支持匹配同一个环境（Namespace）下的 Topic，Namespace不能做正则匹配。

Reader

通过 Reader 的订阅模式，可以从指定的消息开始读取消息。

```
ReaderConfiguration conf = new ReaderConfiguration();
byte[] msgIdBytes = // 消息ID 的字节数组
MessageId id = MessageId.fromByteArray(msgIdBytes);
Reader reader = pulsarClient.newReader()
.topic(topic)
.startMessageId(id)
.create();

while (true) {
Message message = reader.readNext();
// 处理消息
}
```

关闭链接

生产和消费数据完成之后，注意需要关闭链接，包括 Consumer 和 Producer 的链接，以及 Client 的链接。

```
producer.close();
consumer.close();
client.close();
```

Java SDK 下载方式

最近更新时间：2020-09-17 18:12:30

开发前准备

在执行安装脚本之前，请确保您的机器上已经安装了 Java（≥1.8版本）和 Maven。

1. 安装 Java

1.1 检查 Java 安装

打开终端，执行如下命令：

```
java -version
```

如果输出 Java 版本号，说明 Java 安装成功；如果没有安装 Java，请 下载安装 Java 软件开发套件（JDK）。

1.2 设置 Java 环境

设置 JAVA_HOME 环境变量，并指向您机器上的 Java 安装目录。
以 Java jdk1.8.0_20 版本为例，操作系统的输出如下：

操作系统	输出
Windows	Set the environment variable JAVA_HOME to C:\Program Files\Java\jdkjdk1.8.0_20
Linux	export JAVA_HOME=/usr/local/java-current
Mac OSX	export JAVA_HOME=/Library/Java/Home

将 Java 编译器地址添加到系统路径中：

操作系统	输出
Windows	将字符串";C:\Program Files\Java\jdk1.8.0_20\bin"添加到系统变量"Path"的末尾
Linux	export PATH=$PATH:$JAVA_HOME/bin/
Mac OSX	not required

使用上面提到的 java -version 命令验证 Java 安装。

2. 安装 Maven

2.1 下载 安装 Maven

参考 Maven 下载。

2.2 设置 MAVEN_HOME 和 PATH 环境变量

- Windows 系统下：

  新建系统变量 **MAVEN_HOME** 变量值：**E:\Maven\apache-maven-3**.3.9
  编辑系统变量 **Path** 添加变量值： ;%**MAVEN_HOME**%**bin**

- Linux / macOS 系统下：

```
export MAVEN_HOME=/usr/local/maven/apache-maven-3.3.9
export PATH=$MAVEN_HOME/bin:$PATH
```

2.3 验证 Maven 安装

当 Maven 安装完成后， 通过执行如下命令验证 Maven 是否安装成功。

```
mvn --version
```

若出现正常的版本号信息后，说明 Maven 安装成功。

安装 SDK

1. 在您 Java 工程的 `pom.xml` 中添加以下依赖：

```
<!-- in your <properties> block -->
<pulsar.version>2.6.1</pulsar.version>
<!-- in your <dependencies> block -->
<dependency>
<groupId>org.apache.pulsar</groupId>
<artifactId>pulsar-client</artifactId>
<version>2.6.0</version>
</dependency>
```

2. 在 `pom.xml` 所在目录执行 `mvn clean package` 即可下载 Java SDK。

说明：
TDMQ 现已兼容 Pulsar 官方 Java 客户端，您可以直接使用2.6.0以上社区版本的客户端进行开发。