Distributed Message Service for Kafka

Developer Guide

Issue	01
Date	2024-05-15

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Copyright © Huawei Cloud Computing Technologies Co., Ltd. 2024. All rights reserved.

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Cloud Computing Technologies Co., Ltd.

Trademarks and Permissions

HUAWEI and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

1 Overview

Kafka instances are compatible with Apache Kafka and can be accessed using **open-source Kafka clients**. To access an instance in SASL mode, you will also need certificates.

This document describes how to collect instance connection information, such as the instance connection address, certificate required for SASL connection, and information required for public access. It also provides examples of accessing an instance in Java, Python, and Go.

The examples only demonstrate how to invoke Kafka APIs for producing and consuming messages. For more information about the APIs provided by Kafka, visit the **Kafka official website**.

Client Network Environment

A client can access a Kafka instance in any of the following modes:

- If the client runs an Elastic Cloud Server (ECS) and is in the same region and VPC as the Kafka instance, the client can access the instance using a public network IP address.
- If the client runs an ECS and is in the same region but not the same VPC as the Kafka instance, the client can access the instance using one of the following methods:
  - Establish a VPC peering connection to allow two VPCs to communicate with each other. For details, see **VPC Peering Connection**.
  - Use VPC Endpoint (VPCEP) to establish a cross-VPC connection.
- If the client is not in the same network environment or region as the Kafka instance, the client can access the instance using a public network IP address.

  For public access, modify the inbound rules of the security group configured for the Kafka instance, allowing access over port 9094 (SASL disabled) or 9095 (SASL enabled).

NOTE

The three modes differ only in the connection address for the client to access the instance. This document takes intra-VPC access as an example to describe how to set up the development environment.

If the connection times out or fails, check the network connectivity. You can use telnet to test the connection address and port of the instance.

2 Collecting Connection Information

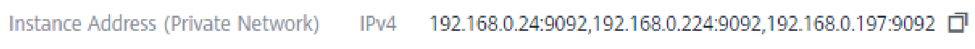
Required Kafka Instance Information

- Instance connection address and port

  Obtain them from the **Basic Information** page of the Kafka instance console. Each Kafka instance is deployed as a cluster and has at least three connection addresses. Configure all of them on the client.

  For public network access, you can use the public network addresses displayed on the **Basic Information** page.

  Figure 2-1 Viewing the connection addresses and ports of brokers of a Kafka instance

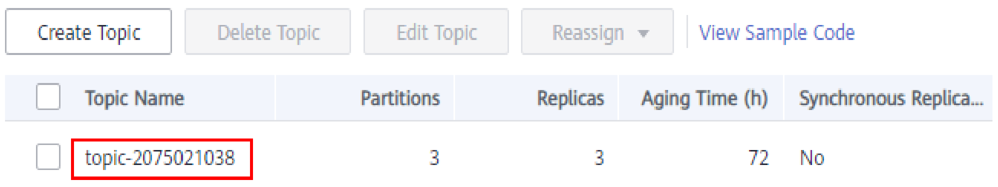
  

  Instance Address (Private Network) IPv4 192.168.0.24:9092,192.168.0.224:9092,192.168.0.197:9092

- Topic name

  Obtain the topic name from the **Topics** page of the Kafka instance console.

  Figure 2-2 Viewing the topic name

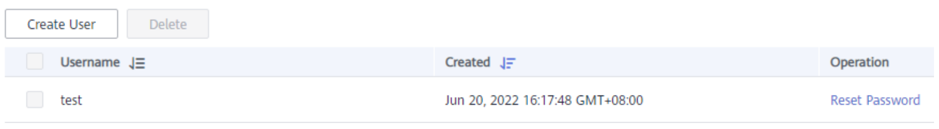
Topic Name	Partitions	Replicas	Aging Time (h)	Synchronous Replica...
topic-2075021038	3	3	72	No

- SASL information

  If SASL_SSL is enabled for the instance during instance creation, obtain the SASL_SSL username, password, certificate, and mechanism.

  – Obtain the username on the **Users** page of the Kafka instance console. If the password is lost, you can .

  Figure 2-3 Obtaining the SASL_SSL username

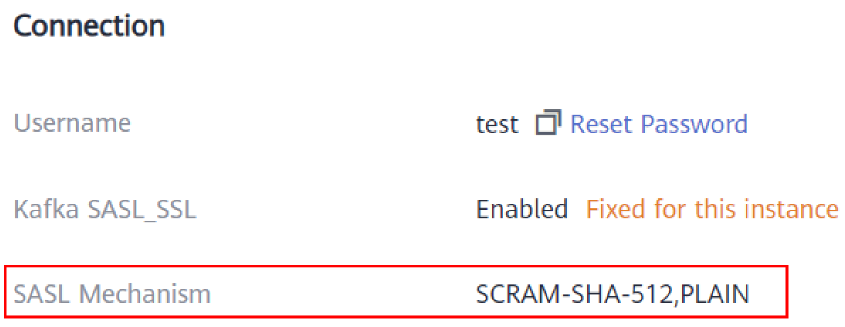
  

Username	Created	Operation
test	Jun 20, 2022 16:17:48 GMT+08:00	Reset Password

- Download the SSL certificate on the **Basic Information** page of the Kafka instance console.

  JKS certificates are used for connecting to instances in Java and CRT certificates are used for connecting to instances in Python.
- View the SASL mechanism on the **Basic Information** page of the Kafka instance console. If both SCRAM-SHA-512 and PLAIN are enabled, configure either of them for connections. If **SASL Mechanism** is not displayed, PLAIN is used by default.

Figure 2-4 SASL mechanism in use



Connection

Username	test Reset Password
Kafka SASL_SSL	Enabled Fixed for this instance
SASL Mechanism	SCRAM-SHA-512,PLAIN

3 Java

3.1 Configuring Kafka Clients in Java

This section describes how to add Kafka clients in Maven, and use the clients to access Kafka instances and produce and consume messages. To check how the demo project runs in IDEA, see **Setting Up the Java Development Environment**.

The Kafka instance connection addresses, topic name, and user information used in the following examples are available in **Collecting Connection Information**.

Adding Kafka Clients in Maven

```
//Kafka instances are based on Kafka 1.1.0/2.3.0/2.7. Use the same version of clients.
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>1.1.0/2.3.0/2.7.2</version>
        </dependency>
```

Preparing Kafka Configuration Files

The following describes example producer and consumer configuration files. If SASL is not enabled for the Kafka instance, comment out lines regarding SASL. If SASL has been enabled, set SASL configurations for encrypted access.

- Producer configuration file (the **dms.sdk.producer.properties** file in the demo project)

  The information in bold is specific to different Kafka instances and must be modified. Other parameters can also be added.

  ```
  #The topic name is in the specific production and consumption code.
  #######################
  #Information about Kafka brokers. ip:port are the connection addresses and ports used by the
  instance. The values can be obtained by referring to the "Collecting Connection Information" section.
  Example: bootstrap.servers=100.xxx.xxx.87:909x,100.xxx.xxx.69:909x,100.xxx.xxx.155:909x
  bootstrap.servers=ip1:port1,ip2:port2,ip3:port3
  #Producer acknowledgement
  acks=all
  #Method of turning the key into bytes
  key.serializer=org.apache.kafka.common.serialization.StringSerializer
  #Method of turning the value into bytes
  value.serializer=org.apache.kafka.common.serialization.StringSerializer
  ```

```
#Memory available to the producer for buffering
buffer.memory=33554432
#Number of retries
retries=0
#######################
#Comment out the following parameters if SASL access is not enabled.
#######################
# Set the SASL authentication mechanism, username, and password.
# sasl.mechanism indicates the SASL authentication mechanism. username and password indicate the
username and password of SASL_SSL. Obtain them by referring to "Collecting Connection
Information."
# If the SASL mechanism is PLAIN, the configuration is as follows:
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
    username="username" \
    password="password";
# If the SASL mechanism is SCRAM-SHA-512, the configuration is as follows:
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required \
    username="username" \
    password="password";

# Set security.protocol.
# If the security protocol is SASL_SSL, the configuration is as follows:
security.protocol=SASL_SSL
# ssl truststore.location is the path for storing the SSL certificate. The following code uses the path
format in Windows as an example. Change the path format based on the actual running environment.
ssl.truststore.location=E:\\temp\\client.truststore.jks
# ssl truststore.password is the password of the server certificate. This password is used for accessing
the JKS file generated by Java.
ssl.truststore.password=dms@kafka
# ssl.endpoint.identification.algorithm indicates whether to verify the certificate domain name. This
parameter must be left blank, which indicates disabling domain name verification.
ssl.endpoint.identification.algorithm=
```

- Consumer configuration file (the **dms.sdk.consumer.properties** file in the demo project)

  The information in bold is specific to different Kafka instances and must be modified. Other parameters can also be added.

```
#The topic name is in the specific production and consumption code.
#######################
#Information about Kafka brokers. ip:port are the connection addresses and ports used by the
instance. The values can be obtained by referring to the "Collecting Connection Information" section.
Example: bootstrap.servers=100.xxx.xxx.87:909x,100.xxx.xxx.69:909x,100.xxx.xxx.155:909x
bootstrap.servers=ip1:port1,ip2:port2,ip3:port3
#Unique string to identify the group of consumer processes to which the consumer belongs.
Configuring the same group.id for different processes indicates that the processes belong to the same
consumer group.
group.id=1
#Method of turning the key into bytes
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
#Method of turning the value into bytes
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer
#Offset reset policy
auto.offset.reset=earliest
#######################
#Comment out the following parameters if SASL access is not enabled.
#######################
# Set the SASL authentication mechanism, username, and password.
# sasl.mechanism indicates the SASL authentication mechanism. username and password indicate the
username and password of SASL_SSL. Obtain them by referring to "Collecting Connection
Information."
# If the SASL mechanism is PLAIN, the configuration is as follows:
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
    username="username" \
    password="password";
# If the SASL mechanism is SCRAM-SHA-512, the configuration is as follows:
```

```
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required \
    username="username" \
    password="password";

# Set security.protocol.
# If the security protocol is SASL_SSL, the configuration is as follows:
security.protocol=SASL_SSL
# ssl truststore.location is the path for storing the SSL certificate. The following code uses the path
format in Windows as an example. Change the path format based on the actual running environment.
ssl.truststore.location=E:\\temp\\client.truststore.jks
# ssl truststore.password is the password of the server certificate. This password is used for accessing
the JKS file generated by Java.
ssl.truststore.password=dms@kafka
# ssl.endpoint.identification.algorithm indicates whether to verify the certificate domain name. This
parameter must be left blank, which indicates disabling domain name verification.
ssl.endpoint.identification.algorithm=
```

Producing Messages

- Test code

```
package com.dms.producer;

import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.junit.Test;

public class DmsProducerTest {
    @Test
    public void testProducer() throws Exception {
        DmsProducer<String, String> producer = new DmsProducer<String, String>();
        int partition = 0;
        try {
            for (int i = 0; i < 10; i++) {
                String key = null;
                String data = "The msg is " + i;
                //Enter the name of the topic you created. There are multiple APIs for producing messages.
For details, see the Kafka official website or the following code.
                producer.produce("topic-0", partition, key, data, new Callback() {
                    public void onCompletion(RecordMetadata metadata,
                        Exception exception) {
                        if (exception != null) {
                            exception.printStackTrace();
                            return;
                        }
                        System.out.println("produce msg completed");
                    }
                });
                System.out.println("produce msg:" + data);
            }
        } catch (Exception e) {
            // TODO: Exception handling
            e.printStackTrace();
        } finally {
            producer.close();
        }
    }
}
```

- Message production code

```
package com.dms.producer;

import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;
import java.util.Properties;

import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;

public class DmsProducer<K, V> {
     //Add the producer configurations that have been specified earlier.
    public static final String CONFIG_PRODUCER_FILE_NAME = "dms.sdk.producer.properties";

    private Producer<K, V> producer;

    DmsProducer(String path)
    {
        Properties props = new Properties();
        try {
            InputStream in = new BufferedInputStream(new FileInputStream(path));
            props.load(in);
        }catch (IOException e)
        {
            e.printStackTrace();
            return;
        }
        producer = new KafkaProducer<K,V>(props);
    }
    DmsProducer()
    {
        Properties props = new Properties();
        try {
            props = loadFromClasspath(CONFIG_PRODUCER_FILE_NAME);
        }catch (IOException e)
        {
            e.printStackTrace();
            return;
        }
        producer = new KafkaProducer<K,V>(props);
    }

    /**
     * Producing messages
     *
     * @param topic        Topic
     * @param partition    partition
     * @param key          Message key
     * @param data         Message data
     */
    public void produce(String topic, Integer partition, K key, V data)
    {
        produce(topic, partition, key, data, null, (Callback)null);
    }

    /**
     * Producing messages
     *
     * @param topic        Topic
     * @param partition    partition
     * @param key          Message key
     * @param data         Message data
     * @param timestamp    timestamp
     */
    public void produce(String topic, Integer partition, K key, V data, Long timestamp)
    {
        produce(topic, partition, key, data, timestamp, (Callback)null);
    }
    /**
     * Producing messages
     *
     * @param topic        Topic
     * @param partition    partition
     * @param key          Message key
     * @param data         Message data
     * @param callback     callback
     */
    public void produce(String topic, Integer partition, K key, V data, Callback callback)
    {
        produce(topic, partition, key, data, null, callback);
    }

    public void produce(String topic, V data)
    {
        produce(topic, null, null, data, null, (Callback)null);
    }

    /**
     * Producing messages
     *
     * @param topic        Topic
     * @param partition    partition
     * @param key          Message key
     * @param data         Message data
     * @param timestamp    timestamp
     * @param callback     callback
     */
    public void produce(String topic, Integer partition, K key, V data, Long timestamp, Callback
callback)
    {
        ProducerRecord<K, V> kafkaRecord =
                timestamp == null ? new ProducerRecord<K, V>(topic, partition, key, data)
                        : new ProducerRecord<K, V>(topic, partition, timestamp, key, data);
        produce(kafkaRecord, callback);
    }

    public void produce(ProducerRecord<K, V> kafkaRecord)
    {
        produce(kafkaRecord, (Callback)null);
    }

    public void produce(ProducerRecord<K, V> kafkaRecord, Callback callback)
    {
        producer.send(kafkaRecord, callback);
    }

    public void close()
    {
        producer.close();
    }

    /**
     * get classloader from thread context if no classloader found in thread
     * context return the classloader which has loaded this class
     *
     * @return classloader
     */
    public static ClassLoader getCurrentClassLoader()
    {
        ClassLoader classLoader = Thread.currentThread()
                .getContextClassLoader();
        if (classLoader == null)
        {
            classLoader = DmsProducer.class.getClassLoader();
        }
        return classLoader;
    }

    /**
```

```
     * Load configuration information from classpath.
     *
     * @param configFileName Configuration file name
     * @return Configuration information
     * @throws IOException
     */
    public static Properties loadFromClasspath(String configFileName) throws IOException
    {
        ClassLoader classLoader = getCurrentClassLoader();
        Properties config = new Properties();

        List<URL> properties = new ArrayList<URL>();
        Enumeration<URL> propertyResources = classLoader
                .getResources(configFileName);
        while (propertyResources.hasMoreElements())
        {
            properties.add(propertyResources.nextElement());
        }

        for (URL url : properties)
        {
            InputStream is = null;
            try
            {
                is = url.openStream();
                config.load(is);
            }
            finally
            {
                if (is != null)
                {
                    is.close();
                    is = null;
                }
            }
        }

        return config;
    }
}
```

Consuming Messages

- Test code

```
package com.dms.consumer;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.junit.Test;
import java.util.Arrays;

public class DmsConsumerTest {
    @Test
    public void testConsumer() throws Exception {
        DmsConsumer consumer = new DmsConsumer();
        consumer.consume(Arrays.asList("topic-0"));
        try {
            for (int i = 0; i < 10; i++){
                ConsumerRecords<Object, Object> records = consumer.poll(1000);
                System.out.println("the numbers of topic:" + records.count());
                for (ConsumerRecord<Object, Object> record : records)
                {
                    System.out.println(record.toString());
                }
            }
        }catch (Exception e)
        {
            // TODO: Exception handling
            e.printStackTrace();
        }finally {
            consumer.close();
        }
    }
}
```

- Message consumption code

```
package com.dms.consumer;

import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import java.util.*;

public class DmsConsumer {

    public static final String CONFIG_CONSUMER_FILE_NAME = "dms.sdk.consumer.properties";

    private KafkaConsumer<Object, Object> consumer;

    DmsConsumer(String path)
    {
        Properties props = new Properties();
        try {
            InputStream in = new BufferedInputStream(new FileInputStream(path));
            props.load(in);
        }catch (IOException e)
        {
            e.printStackTrace();
            return;
        }
        consumer = new KafkaConsumer<Object, Object>(props);
    }

    DmsConsumer()
    {
        Properties props = new Properties();
        try {
            props = loadFromClasspath(CONFIG_CONSUMER_FILE_NAME);
        }catch (IOException e)
        {
            e.printStackTrace();
            return;
        }
        consumer = new KafkaConsumer<Object, Object>(props);
    }
    public void consume(List topics)
    {
        consumer.subscribe(topics);
    }

    public ConsumerRecords<Object, Object> poll(long timeout)
    {
        return consumer.poll(timeout);
    }

    public void close()
    {
        consumer.close();
    }

    /**
     * get classloader from thread context if no classloader found in thread
     * context return the classloader which has loaded this class
     *
     * @return classloader
     */
    public static ClassLoader getCurrentClassLoader()
    {
        ClassLoader classLoader = Thread.currentThread()
                .getContextClassLoader();
        if (classLoader == null)
        {
            classLoader = DmsConsumer.class.getClassLoader();
        }
        return classLoader;
    }

    /**
     * Load configuration information from classpath.
     *
     * @param configFileName Configuration file name
     * @return Configuration information
     * @throws IOException
     */
    public static Properties loadFromClasspath(String configFileName) throws IOException
    {
        ClassLoader classLoader = getCurrentClassLoader();
        Properties config = new Properties();

        List<URL> properties = new ArrayList<URL>();
        Enumeration<URL> propertyResources = classLoader
                .getResources(configFileName);
        while (propertyResources.hasMoreElements())
        {
            properties.add(propertyResources.nextElement());
        }

        for (URL url : properties)
        {
            InputStream is = null;
            try
            {
                is = url.openStream();
                config.load(is);
            }
            finally
            {
                if (is != null)
                {
                    is.close();
                    is = null;
                }
            }
        }

        return config;
    }
}
```

3.2 Setting Up the Java Development Environment

With the information collected in **Collecting Connection Information** and the network environment prepared for Kafka clients, you can proceed to configuring Kafka clients. This section describes how to configure Kafka clients to produce and consume messages.

Preparing Tools

- Maven

Apache Maven 3.0.3 or later can be downloaded from the **Maven official website**.

- JDK

  Java Development Kit1.8.111 or later can be downloaded from the **Oracle official website**.

  After the installation, configure the Java environment variables.

- IntelliJ IDEA

  IntelliJ IDEA can be downloaded from the **IntelliJ IDEA official website** and be installed.

Procedure

Step 1 Download the **demo package**.

Decompress the package to obtain the following files.

Table 3-1 Files in the demo package

File	Directory	Description
DmsConsumer.java	.\src\main\java\com \dms\consumer	API for consuming messages
DmsProducer.java	.\src\main\java\com \dms\producer	API for producing messages
dms.sdk.consumer. properties	.\src\main\resources	Configuration information for consuming messages
dms.sdk.producer.p roperties	.\src\main\resources	Configuration information for producing messages
client.truststore.jks	.\src\main\resources	SSL certificate, used for SASL connection
DmsConsumerTest.j ava	.\src\test\java\com\dms \consumer	Test code of consuming messages
DmsProducerTest.ja va	.\src\test\java\com\dms \producer	Test code of producing messages
pom.xml	.\	Maven configuration file, containing the Kafka client dependencies

Step 2 In IntelliJ IDEA, import the demo project.

The demo project is a Java project built in Maven. Therefore, you need the JDK and the Maven plugin in IDEA.

Figure 3-1 Click **Import Project**.

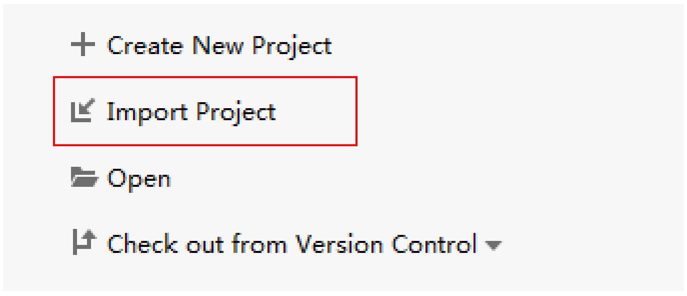
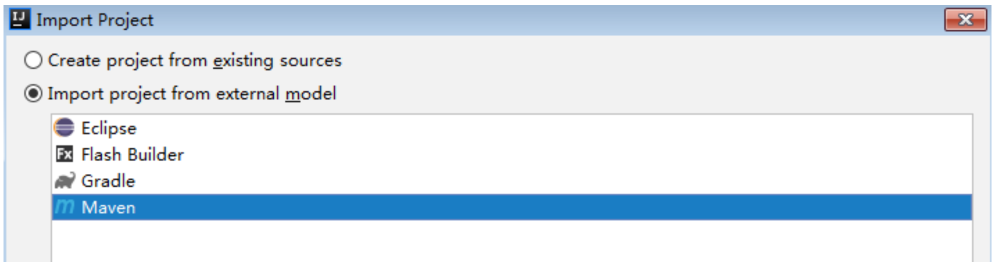


Figure 3-2 Choose **Maven**.

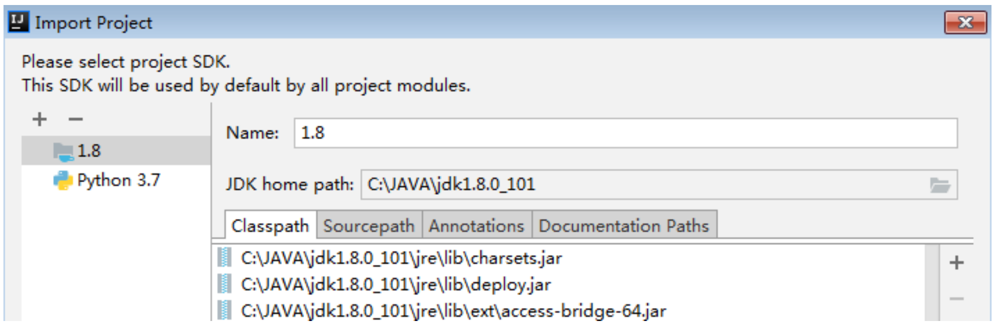

Import Project

○ Create project from existing sources

◉ Import project from external model

- Eclipse
- Flash Builder
- Gradle
- Maven

Figure 3-3 Select the JDK.


Import Project

Please select project SDK.
This SDK will be used by default by all project modules.

- 1.8
- Python 3.7

Name: 1.8

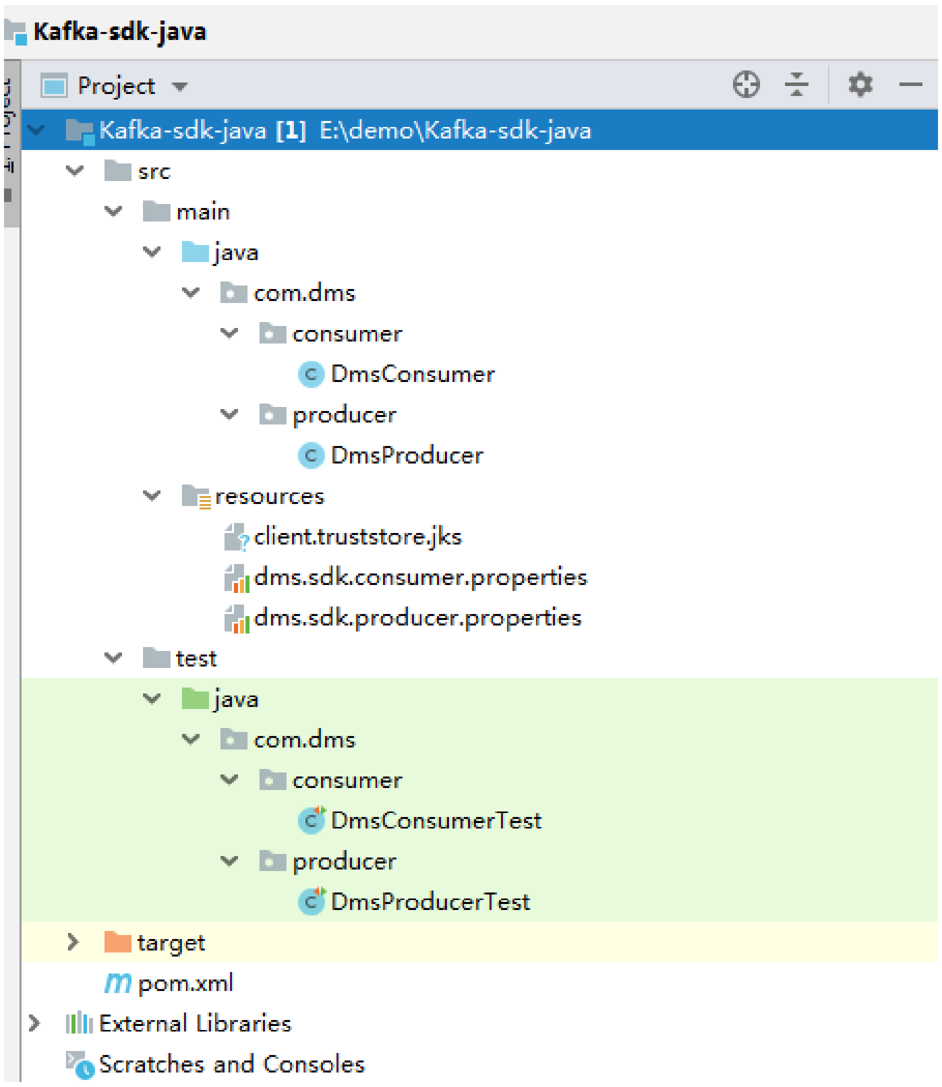
JDK home path: C:\JAVA\jdk1.8.0_101

Classpath | Sourcepath | Annotations | Documentation Paths

```
C:\JAVA\jdk1.8.0_101\jre\lib\charsets.jar
C:\JAVA\jdk1.8.0_101\jre\lib\deploy.jar
C:\JAVA\jdk1.8.0_101\jre\lib\ext\access-bridge-64.jar
```

You can select other options or retain the default settings. Click **Finish**.

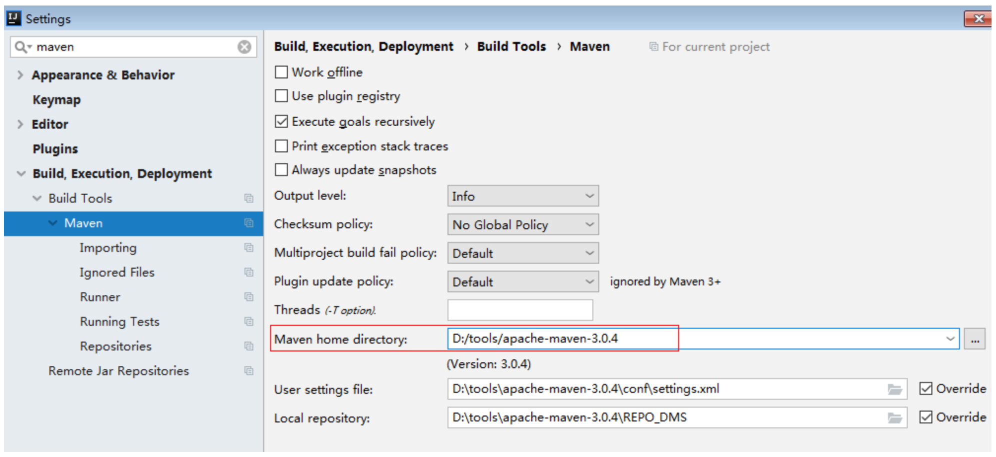
The demo project has been imported.



```
Kafka-sdk-java
Project
Kafka-sdk-java [1]  E:\demo\Kafka-sdk-java
  src
    main
      java
        com.dms
          consumer
            DmsConsumer
          producer
            DmsProducer
      resources
        client.truststore.jks
        dms.sdk.consumer.properties
        dms.sdk.producer.properties
    test
      java
        com.dms
          consumer
            DmsConsumerTest
          producer
            DmsProducerTest
  target
  pom.xml
External Libraries
Scratches and Consoles
```

Step 3 Configure Maven.

Choose **Files** > **Settings**, set **Maven home directory** correctly, and select the required **settings.xml** file.



Settings

Build, Execution, Deployment › Build Tools › Maven   For current project

- [ ] Work offline
- [ ] Use plugin registry
- [x] Execute goals recursively
- [ ] Print exception stack traces
- [ ] Always update snapshots

Output level:	Info
Checksum policy:	No Global Policy
Multiproject build fail policy:	Default
Plugin update policy:	Default ignored by Maven 3+
Threads (-T option):	
Maven home directory:	D:/tools/apache-maven-3.0.4 (Version: 3.0.4)
User settings file:	D:\tools\apache-maven-3.0.4\conf\settings.xml [x] Override
Local repository:	D:\tools\apache-maven-3.0.4\REPO_DMS [x] Override

Step 4 Specify Kafka configurations.

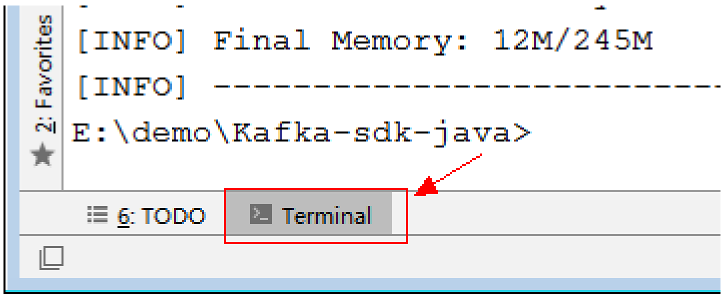
The following is a configuration example for producing messages. Replace the information in bold with the actual values.

```
#The information in bold is specific to different Kafka instances and must be modified. Other parameters can also be added.
#The topic name is in the specific production and consumption code.
#######################
#Information about Kafka brokers. ip:port are the connection addresses and ports used by the instance. The values can be obtained by referring to the "Collecting Connection Information" section. Example: bootstrap.servers=100.xxx.xxx.87:909x,100.xxx.xxx.69:909x,100.xxx.xxx.155:909x
bootstrap.servers=ip1:port1,ip2:port2,ip3:port3
#Producer acknowledgement
acks=all
#Method of turning the key into bytes
key.serializer=org.apache.kafka.common.serialization.StringSerializer
#Method of turning the value into bytes
value.serializer=org.apache.kafka.common.serialization.StringSerializer
#Memory available to the producer for buffering
buffer.memory=33554432
#Number of retries
retries=0
#######################
#Comment out the following parameters if SASL access is not enabled.
#######################
# Set the SASL authentication mechanism, username, and password.
# sasl.mechanism indicates the SASL authentication mechanism. username and password indicate the username and password of SASL_SSL. Obtain them by referring to "Collecting Connection Information."
# If the SASL mechanism is PLAIN, the configuration is as follows:
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
    username="username" \
    password="password";
# If the SASL mechanism is SCRAM-SHA-512, the configuration is as follows:
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required \
    username="username" \
    password="password";

# Set security.protocol.
# If the security protocol is SASL_SSL, the configuration is as follows:
security.protocol=SASL_SSL
# ssl truststore.location is the path for storing the SSL certificate. The following code uses the path format in Windows as an example. Change the path format based on the actual running environment.
ssl.truststore.location=E:\\temp\\client.truststore.jks
# ssl truststore.password is the password of the server certificate. This password is used for accessing the JKS file generated by Java.
ssl.truststore.password=dms@kafka
# ssl.endpoint.identification.algorithm indicates whether to verify the certificate domain name. This parameter must be left blank, which indicates disabling domain name verification.
ssl.endpoint.identification.algorithm=
```

Step 5 In the down left corner of IDEA, click **Terminal**. In terminal, run the **mvn test** command to see how the demo project goes.

Figure 3-4 Opening terminal in IDEA



```
[INFO] Final Memory: 12M/245M
[INFO] --------------------------
E:\demo\Kafka-sdk-java>
```

The following information is displayed for the producer:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.dms.producer.DmsProducerTest
produce msg:The msg is 0
produce msg:The msg is 1
produce msg:The msg is 2
produce msg:The msg is 3
produce msg:The msg is 4
produce msg:The msg is 5
produce msg:The msg is 6
produce msg:The msg is 7
produce msg:The msg is 8
produce msg:The msg is 9
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 138.877 sec
```

The following information is displayed for the consumer:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.dms.consumer.DmsConsumerTest
the numbers of topic:0
the numbers of topic:0
the numbers of topic:6
ConsumerRecord(topic = topic-0, partition = 2, offset = 0, CreateTime = 1557059377179, serialized key size
= -1, serialized value size = 12, headers = RecordHeaders(headers = [], isReadOnly = false), key = null, value
= The msg is 2)
ConsumerRecord(topic = topic-0, partition = 2, offset = 1, CreateTime = 1557059377195, serialized key size
= -1, serialized value size = 12, headers = RecordHeaders(headers = [], isReadOnly = false), key = null, value
= The msg is 5)
```

----End

4 Python

This section describes how to access a Kafka premium instance using a Kafka client in Python on the Linux CentOS, including how to install the client, and produce and consume messages.

Before getting started, ensure that you have collected the information listed in **Collecting Connection Information**.

Preparing the Environment

- Python

  Generally, Python is pre-installed in the system. Enter **python** in a CLI. If the following information is displayed, Python has already been installed.

  ```
  [root@ecs-test python-kafka]# python3
  Python 3.7.1 (default, Jul  5 2020, 14:37:24)
  [GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
  Type "help", "copyright", "credits" or "license" for more information.
  >>>
  ```

  If Python is not installed, run the following command:

  yum install python

- Kafka clients in Python

  Run the following command to install a Python client of the recommended version:

  pip install kafka-python==2.0.1

Producing Messages

NOTE

Replace the following information in bold with the actual values.

- With SASL

  ```
  from kafka import KafkaProducer
  import ssl
  ##Connection information
  conf = {
      'bootstrap_servers': ["ip1:port1","ip2:port2","ip3:port3"],
      'topic_name': 'topic_name',
      'sasl_username': 'username',
      'sasl_password': 'password'
  }
  ```

```
context = ssl.create_default_context()
context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
context.verify_mode = ssl.CERT_REQUIRED
## Certificate file. For details about how to obtain an SSL certificate, see section "Collecting
Connection Information."
context.load_verify_locations("phy_ca.crt")

print('start producer')
producer = KafkaProducer(bootstrap_servers=conf['bootstrap_servers'],
                         sasl_mechanism="PLAIN",
                         ssl_context=context,
                         security_protocol='SASL_SSL',
                         sasl_plain_username=conf['sasl_username'],
                         sasl_plain_password=conf['sasl_password'])

data = bytes("hello kafka!", encoding="utf-8")
producer.send(conf['topic_name'], data)
producer.close()
print('end producer')
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **bootstrap_servers**: instance connection address and port
- **topic_name**: topic name
- **sasl_plain_username** and **sasl_plain_password**: username and password you set when enabling SASL_SSL during Kafka instance creation or when creating a SASL_SSL user.
- **context.load_verify_locations**: certificate file. CRT certificates are used for connecting to instances in Python.
- **sasl_mechanism**: SASL authentication mechanism. View it on the **Basic Information** page of the Kafka instance console. If both SCRAM-SHA-512 and PLAIN are enabled, configure either of them for connections. If **SASL Mechanism** is not displayed, PLAIN is used by default.

- Without SASL

```
from kafka import KafkaProducer

conf = {
    'bootstrap_servers': ["ip1:port1","ip2:port2","ip3:port3"],
    'topic_name': 'topic-name',
}

print('start producer')
producer = KafkaProducer(bootstrap_servers=conf['bootstrap_servers'])

data = bytes("hello kafka!", encoding="utf-8")
producer.send(conf['topic_name'], data)
producer.close()
print('end producer')
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **bootstrap_servers**: instance connection address and port
- **topic_name**: topic name

Consuming Messages

- With SASL

```
from kafka import KafkaConsumer
import ssl
##Connection information
conf = {
    'bootstrap_servers': ["ip1:port1","ip2:port2","ip3:port3"],
    'topic_name': 'topic_name',
    'sasl_username': 'username',
    'sasl_password': 'password',
    'consumer_id': 'consumer_id'
}

context = ssl.create_default_context()
context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
context.verify_mode = ssl.CERT_REQUIRED
## Certificate file. For details about how to obtain an SSL certificate, see section "Collecting 
Connection Information."
context.load_verify_locations("phy_ca.crt")

print('start consumer')
consumer = KafkaConsumer(conf['topic_name'],
                        bootstrap_servers=conf['bootstrap_servers'],
                        group_id=conf['consumer_id'],
                        sasl_mechanism="PLAIN",
                        ssl_context=context,
                        security_protocol='SASL_SSL',
                        sasl_plain_username=conf['sasl_username'],
                        sasl_plain_password=conf['sasl_password'])

for message in consumer:
    print("%s:%d:%d: key=%s value=%s" % (message.topic, message.partition,message.offset, 
message.key,message.value))

print('end consumer')
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **bootstrap_servers**: instance connection address and port
- **topic_name**: topic name
- **sasl_plain_username** and **sasl_plain_password**: username and password you set when enabling SASL_SSL during Kafka instance creation or when creating a SASL_SSL user.
- **consumer_id**: custom consumer group name. If the specified consumer group does not exist, Kafka automatically creates one.
- **context.load_verify_locations**: certificate file. CRT certificates are used for connecting to instances in Python.
- **sasl_mechanism**: SASL authentication mechanism. View it on the **Basic Information** page of the Kafka instance console. If both SCRAM-SHA-512 and PLAIN are enabled, configure either of them for connections. If **SASL Mechanism** is not displayed, PLAIN is used by default.

● Without SASL

Replace the information in bold with the actual values.

```
from kafka import KafkaConsumer

conf = {
    'bootstrap_servers': ["ip1:port1","ip2:port2","ip3:port3"],
    'topic_name': 'topic-name',
    'consumer_id': 'consumer-id'
}

print('start consumer')
```

```
consumer = KafkaConsumer(conf['topic_name'],
                        bootstrap_servers=conf['bootstrap_servers'],
                        group_id=conf['consumer_id'])

for message in consumer:
    print("%s:%d:%d: key=%s value=%s" % (message.topic, message.partition,message.offset,
message.key,message.value))

print('end consumer')
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **bootstrap_servers**: instance connection address and port
- **topic_name**: topic name
- **consumer_id**: custom consumer group name. If the specified consumer group does not exist, Kafka automatically creates one.

5 Go

This section describes how to access a Kafka instance using Go 1.16.5 on the Linux CentOS, how to obtain the demo code, and how to produce and consume messages.

Before getting started, ensure that you have collected the information listed in **Collecting Connection Information**.

Preparing the Environment

- Run the following command to check whether Go has been installed:
  ```
  go version
  ```
  If the following information is displayed, Go has been installed.
  ```
  [root@ecs-test confluent-kafka-go]# go version
  go version go1.16.5 linux/amd64
  ```
  If Go is not installed, **download** and install it.
- Run the following command to obtain the code used in the demo:
  ```
  go get github.com/confluentinc/confluent-kafka-go/kafka
  ```

Producing Messages

NOTE

Replace the following information in bold with the actual values.

- With SASL
  ```
  package main

  import (
      "bufio"
      "fmt"
      "github.com/confluentinc/confluent-kafka-go/kafka"
      "log"
      "os"
      "os/signal"
      "syscall"
  )

  var (
      brokers  = "ip1:port1,ip2:port2,ip3:port3"
      topics   = "topic_name"
      user     = "username"
      password = "password"
      caFile   = "phy_ca.crt"     //Certificate file. For details about how to obtain an SSL certificate, see
  ```

```
section "Collecting Connection Information."
)

func main() {
    log.Println("Starting a new kafka producer")

    config := &kafka.ConfigMap{
        "bootstrap.servers": brokers,
        "security.protocol": "SASL_SSL",
        "sasl.mechanism":    "PLAIN",
        "sasl.username":     user,
        "sasl.password":     password,
        "ssl.ca.location":   caFile,
        "ssl.endpoint.identification.algorithm": "none"
    }
    producer, err := kafka.NewProducer(config)
    if err != nil {
        log.Panicf("producer error, err: %v", err)
        return
    }

    go func() {
        for e := range producer.Events() {
            switch ev := e.(type) {
            case *kafka.Message:
                if ev.TopicPartition.Error != nil {
                    log.Printf("Delivery failed: %v\n", ev.TopicPartition)
                } else {
                    log.Printf("Delivered message to %v\n", ev.TopicPartition)
                }
            }
        }
    }()

    // Produce messages to topic (asynchronously)
    fmt.Println("please enter message:")
    go func() {
        for {
            err := producer.Produce(&kafka.Message{
                TopicPartition: kafka.TopicPartition{Topic: &topics, Partition: kafka.PartitionAny},
                Value:          GetInput(),
            }, nil)
            if err != nil {
                log.Panicf("send message fail, err: %v", err)
                return
            }
        }
    }()

    sigterm := make(chan os.Signal, 1)
    signal.Notify(sigterm, syscall.SIGINT, syscall.SIGTERM)
    select {
    case <-sigterm:
        log.Println("terminating: via signal")
    }
    // Wait for message deliveries before shutting down
    producer.Flush(15 * 1000)
    producer.Close()
}

func GetInput() []byte {
    reader := bufio.NewReader(os.Stdin)
    data, _, _ := reader.ReadLine()
    return data
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **brokers**: instance connection address and port
- **topics**: topic name
- **user** and **password**: username and password you set when enabling SASL_SSL during Kafka instance creation or when creating a SASL_SSL user.
- **caFile**: certificate file
- **sasl.mechanism**: SASL authentication mechanism. View it on the **Basic Information** page of the Kafka instance console. If both SCRAM-SHA-512 and PLAIN are enabled, configure either of them for connections. If **SASL Mechanism** is not displayed, PLAIN is used by default.

- Without SASL

```
package main

import (
    "bufio"
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
    "log"
    "os"
    "os/signal"
    "syscall"
)

var (
    brokers  = "ip1:port1,ip2:port2,ip3:port3"
    topics   = "topic_name"
)

func main() {
    log.Println("Starting a new kafka producer")

    config := &kafka.ConfigMap{
        "bootstrap.servers": brokers,
    }
    producer, err := kafka.NewProducer(config)
    if err != nil {
        log.Panicf("producer error, err: %v", err)
        return
    }

    go func() {
        for e := range producer.Events() {
            switch ev := e.(type) {
            case *kafka.Message:
                if ev.TopicPartition.Error != nil {
                    log.Printf("Delivery failed: %v\n", ev.TopicPartition)
                } else {
                    log.Printf("Delivered message to %v\n", ev.TopicPartition)
                }
            }
        }
    }()

    // Produce messages to topic (asynchronously)
    fmt.Println("please enter message:")
    go func() {
        for {
            err := producer.Produce(&kafka.Message{
                TopicPartition: kafka.TopicPartition{Topic: &topics, Partition: kafka.PartitionAny},
                Value:          GetInput(),
            }, nil)
            if err != nil {
                log.Panicf("send message fail, err: %v", err)
                return
            }
        }
    }()

    sigterm := make(chan os.Signal, 1)
    signal.Notify(sigterm, syscall.SIGINT, syscall.SIGTERM)
    select {
    case <-sigterm:
        log.Println("terminating: via signal")
    }
    // Wait for message deliveries before shutting down
    producer.Flush(15 * 1000)
    producer.Close()
}

func GetInput() []byte {
    reader := bufio.NewReader(os.Stdin)
    data, _, _ := reader.ReadLine()
    return data
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

– **brokers**: instance connection address and port
– **topics**: topic name

Consuming Messages

NOTE

Replace the following information in bold with the actual values.

- With SASL

```
package main

import (
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
    "log"
    "os"
    "os/signal"
    "syscall"
)

var (
    brokers  = "ip1:port1,ip2:port2,ip3:port3"
    group    = "group-id"
    topics   = "topic_name"
    user     = "username"
    password = "password"
    caFile   = "phy_ca.crt"     //Certificate file. For details about how to obtain an SSL certificate, see
section "Collecting Connection Information."
)

func main() {
    log.Println("Starting a new kafka consumer")

    config := &kafka.ConfigMap{
        "bootstrap.servers": brokers,
        "group.id":          group,
        "auto.offset.reset": "earliest",
        "security.protocol": "SASL_SSL",
        "sasl.mechanism":    "PLAIN",
        "sasl.username":     user,
        "sasl.password":     password,
        "ssl.ca.location":   caFile,
        "ssl.endpoint.identification.algorithm": "none"
    }

    consumer, err := kafka.NewConsumer(config)
    if err != nil {
        log.Panicf("Error creating consumer: %v", err)
        return
    }

    err = consumer.SubscribeTopics([]string{topics}, nil)
    if err != nil {
        log.Panicf("Error subscribe consumer: %v", err)
        return
    }

    go func() {
        for {
            msg, err := consumer.ReadMessage(-1)
            if err != nil {
                log.Printf("Consumer error: %v (%v)", err, msg)
            } else {
                fmt.Printf("Message on %s: %s\n", msg.TopicPartition, string(msg.Value))
            }
        }
    }()

    sigterm := make(chan os.Signal, 1)
    signal.Notify(sigterm, syscall.SIGINT, syscall.SIGTERM)
    select {
    case <-sigterm:
        log.Println("terminating: via signal")
    }
    if err = consumer.Close(); err != nil {
        log.Panicf("Error closing consumer: %v", err)
    }
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **brokers**: instance connection address and port
- **group**: custom consumer group name. If the specified consumer group does not exist, Kafka automatically creates one.
- **topics**: topic name
- **user** and **password**: username and password you set when enabling SASL_SSL during Kafka instance creation or when creating a SASL_SSL user.
- **caFile**: certificate file
- **sasl.mechanism**: SASL authentication mechanism. View it on the **Basic Information** page of the Kafka instance console. If both SCRAM-SHA-512 and PLAIN are enabled, configure either of them for connections. If **SASL Mechanism** is not displayed, PLAIN is used by default.

- Without SASL

```
package main

import (
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
    "log"
    "os"
    "os/signal"
    "syscall"
)
```

```
var (
    brokers  = "ip1:port1,ip2:port2,ip3:port3"
    group    = "group-id"
    topics   = "topic_name"
)

func main() {
    log.Println("Starting a new kafka consumer")

    config := &kafka.ConfigMap{
        "bootstrap.servers": brokers,
        "group.id":          group,
        "auto.offset.reset": "earliest",
    }

    consumer, err := kafka.NewConsumer(config)
    if err != nil {
        log.Panicf("Error creating consumer: %v", err)
        return
    }

    err = consumer.SubscribeTopics([]string{topics}, nil)
    if err != nil {
        log.Panicf("Error subscribe consumer: %v", err)
        return
    }

    go func() {
        for {
            msg, err := consumer.ReadMessage(-1)
            if err != nil {
                log.Printf("Consumer error: %v (%v)", err, msg)
            } else {
                fmt.Printf("Message on %s: %s\n", msg.TopicPartition, string(msg.Value))
            }
        }
    }()

    sigterm := make(chan os.Signal, 1)
    signal.Notify(sigterm, syscall.SIGINT, syscall.SIGTERM)
    select {
    case <-sigterm:
        log.Println("terminating: via signal")
    }
    if err = consumer.Close(); err != nil {
        log.Panicf("Error closing consumer: %v", err)
    }
}
```

The parameters in the example code are described as follows. For details about how to obtain the parameter values, see **Collecting Connection Information**.

- **brokers**: instance connection address and port
- **group**: custom consumer group name. If the specified consumer group does not exist, Kafka automatically creates one.
- **topics**: topic name

6 Obtaining Kafka Clients

Kafka instances are fully compatible with open-source clients. You can obtain **clients in other programming languages** and access your instance as instructed by the official Kafka website.

7 Using spring-kafka

This section describes how to use spring-kafka to connect to a Huawei Cloud Kafka instance to produce and consume messages. Obtain the related code from **kafka-springboot-demo**.

The Kafka instance connection addresses, topic name, and user information used in the following examples are available in **Collecting Connection Information**.

Adding the spring-kafka Dependency to the pom.xml File

```
<dependency>
   <groupId>org.springframework.kafka</groupId>
   <artifactId>spring-kafka</artifactId>
</dependency>
```

Configuring the application.properties File

```
#=============== Kafka ==========================
## Broker information of the Kafka instance. ip:port indicates the connection address and port number of the instance.
spring.kafka.bootstrap-servers=ip1:port1,ip2:port2,ip3:port3
#=============== Producer Configuration =======================
spring.kafka.producer.retries=0
spring.kafka.producer.batch-size=16384
spring.kafka.producer.buffer-memory=33554432
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer
#=============== Consumer Configuration =======================
spring.kafka.consumer.group-id=test-consumer-group
spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.enable-auto-commit=true
spring.kafka.consumer.auto-commit-interval=100
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=org.apache.kafka.common.serialization.StringDeserializer
#======== SASL Configuration (Delete the following configuration if SASL is disabled.) =======
## Set the SASL authentication mechanism, username, and password.
# spring.kafka.properties.sasl.mechanism indicates the SASL authentication mechanism. username and password indicate the username and password of SASL_SSL. Obtain them by referring to "Collecting Connection Information."
## If the SASL mechanism is PLAIN, the configuration is as follows:
spring.kafka.properties.sasl.mechanism=PLAIN
spring.kafka.properties.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
   username="username" \
   password="password";
## If the SASL mechanism is SCRAM-SHA-512, the configuration is as follows:
spring.kafka.properties.sasl.mechanism=SCRAM-SHA-512
spring.kafka.properties.sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required \
    username="**username**" \
    password="**password**";

# Set spring.kafka.security.protocol.
## **If the security protocol is SASL_SSL, the configuration is as follows:**
spring.kafka.security.protocol=SASL_SSL
# spring.kafka.ssl.trust-store-location is the path for storing the SSL certificate. The following code uses the path format in Windows as an example. Change the path format based on the actual running environment.
spring.kafka.ssl.trust-store-location=**E:\\temp\\client.truststore.jks**
# spring.kafka.ssl.trust-store-password is the password of the server certificate. This password does not need to be modified. It is used for accessing the JKS file generated by Java.
spring.kafka.ssl.trust-store-password=dms@kafka
# spring.kafka.properties.ssl.endpoint.identification.algorithm indicates whether to verify the certificate domain name. **This parameter must be left blank, which indicates disabling domain name verification.**
spring.kafka.properties.ssl.endpoint.identification.algorithm=
```

Producing Messages

```
package com.huaweicloud.dms.example.producer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.util.UUID;

/**
 * @author huaweicloud DMS
 */
@Component
public class DmsKafkaProducer {
    /**
     * Topic name. Use the actual topic name.
     */
    public static final String TOPIC = "test_topic";

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    /**
     * One message is produced every five seconds as scheduled.
     */
    @Scheduled(cron = "*/5 * * * * ?")
    public void send() {
        String message = String.format("{id:%s,timestamp:%s}", UUID.randomUUID().toString(), 
System.currentTimeMillis());
        kafkaTemplate.send(TOPIC, message);
        System.out.println("send finished, message = " + message);
    }
}
```

Consuming Messages

```
package com.huaweicloud.dms.example.consumer;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;

import java.util.Optional;

/**
 * @author huaweicloud DMS
 */
@Component
public class DmsKafkaConsumer {
    /**
     * Topic name. Use the actual topic name.
     */
    private static final String TOPIC = "test_topic";

    @KafkaListener(topics = {TOPIC})
    public void listen(ConsumerRecord<String, String> record) {
        Optional<String> message = Optional.ofNullable(record.value());
        if (message.isPresent()) {
            System.out.println("consume finished, message = " + message.get());
        }
    }
}
```