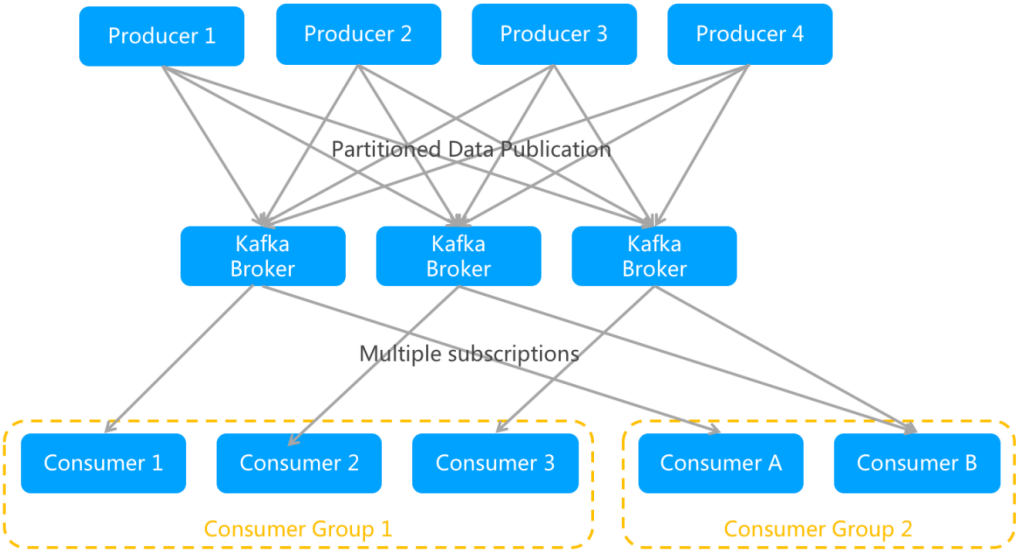
分布式消息 Kafka

用户指南

1 产品介绍

什么是分布式消息服务 Kafka？

分布式消息服务 Kafka 是一个分布式的、高吞吐量、高可扩展性消息队列服务，广泛用于日志收集、监控数据聚合、流式数据处理、在线和离线分析等大数据领域，是大数据生态中不可或缺的产品之一。



Produce（生产者）

生产者可以将数据发布到所选择的 topic（主题）中。生产者负责将记录分配到 topic 的哪一个 partition（分区）中。可以使用循环的方式来简单地实现负载均衡，也可以根据某些语义分区函数(例如：记录中的 key)来完成。下面会介绍更多关于分区的使用。

Consumer（消费者）

消费者使用一个 消费组 名称来进行标识，发布到 topic 中的每条记录被分配给订阅消费组中的一个消费者实例.消费者实例可以分布在多个进程中或者多个机器上。

Broker

kafka 集群包含一个活多个消息处理服务器，该服务器成为 Broker，提供数据刷盘等核心功能。可以横向扩展、在线扩容以提高集群性能。

分布式消息服务 Kafka 的优势

分布式消息队列 Kafka 针对开源的 Kafka 提供全托管服务，彻底解决开源产品长期以来的痛点，用户只需专注于业务开发，无需部署运维，低成本、更弹性、更可靠。

分布式消息服务 Kafka 的产品功能

（1） 消息能力

- 广播消息

在同一个消费组内对所有消费者投递相同消息。

- 消息回溯

支持根据时间重置消费进度

- 消息数据自动删除功能

在磁盘满后，在保护期外的数据，能自动删除，保证服务可用性

- 自动故障切换功能

生产消费自动负载均衡，消息节点故障时自动主备切换，保证服务的连续性。

（2）队列能力

- 高吞吐，消息多副本异步复制。
- 高可靠，消息多副本同步复制。

（3）可视化管理

- 应用用户管理

多个应用可调用同一个消息服务，通过应用用户，对消息服务下的应用接入权限进行管理。

- 主题管理

支持对实例下的主题进行管理，执行创建删除等操作。

- 消费组管理

支持对实例下的消费组进行管理。

- Broker 监控

提供 Broker 详细信息以及多维度的监控指标查看。

- Topic 监控

提供 Topic 详细信息以及多维度的监控指标查看。

（4）安全防护

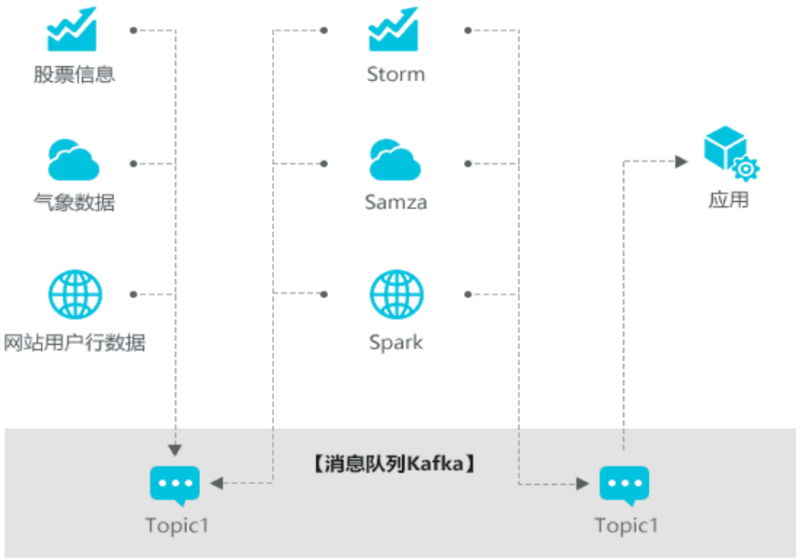
- 可追溯租户管理操作的记录。
- 提供用户鉴权和 SASL 授权访问机制，提供企业级的安全防护

分布式消息服务 Kafka 应用场景

（1）流计算处理

Kafka 能够做到流计算处理 ,比如股市走向分析、气象数据测控、网站用户行为分析等领域，由于在这些领域中数据产生快、实时性强、数据量大，所以很难统一采集并入库存储后再做处理，这便导致传统的数据处理架构不能满足需求。而 Kafka Stream 以及 Storm/Samza/Spark 等流计算引擎的出现，可以根据业务需求对数据进行计算分析，最终把结果保存或者分发给需要的组件。

- 构建应用系统和分析系统的桥梁，并将它们之间的关联解耦；
- 通过支持流计算引擎，可对接开源 Storm/Samza/Spark 流计算引擎。

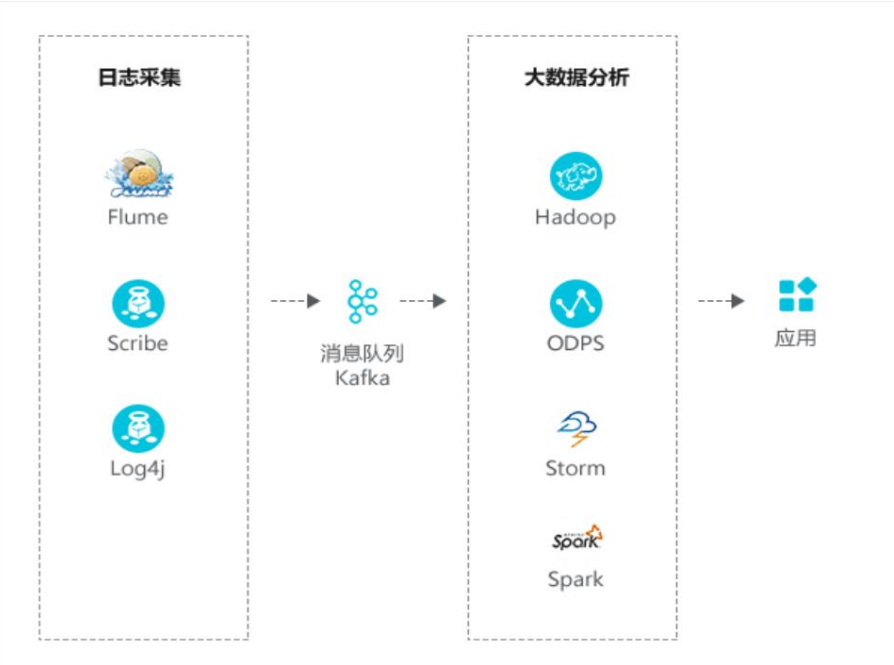


（2）日志聚合

Kafka 本身的性能是非常高效的，同时 Kafka 的特性决定它非常适合作为"日志收集中心"，这是因为 Kafka 在采集日志的时候业务是无感知的，其能够兼容自己的上游，能够直接地通过配置加密消息。当日志数据发送到 Kafka 集群里面，其实对于业务而言是完全无侵入的。同时其在下游又能够直接地对接 Hadoop/ODPS 等离线仓库存储和 Strom/Spark 等实现实

时在线分析。在这样的情况之下，使用 Kafka，只需要用户去关注整个流程里面的业务逻辑，而无需做更多的开发就能够实现统计、分析以及报表。

- 构建应用系统和分析系统的桥梁，并将它们之间的关联解耦；
- 支持实时在线分析系统和类似于 Hadoop 的离线分析系统。



分布式消息服务 Kafka 相关术语解释

➢ **Broker**

kafka 集群包含一个活多个消息处理服务器，该服务器成为 Broker，提供数据刷盘等核心功能。可以横向扩展、在线扩容以提高集群性能。

➢ **Topic**

每条发布到 Kafka 集群的消息都有一个主题，这个主题被称为 Topic。通过 Topic 可以对消息进行分类。每个 Topic 可以由一个或多个分区（Partition）组成，存储于一个或多个 Broker 上。

➢ 分区（**partition**）

消息分区是物理上的概念，一个 Topic 可以分为多个 partition，每个 partition 是一个有序的队列。partition 中的每条消息都会被分配一个有序的 id（offset）。

➢ **Producer**

消息和数据生成者，一般为应用调用 API 进行消息生产，并向 Kafka 的一个 Topic 发布消息。

➢ **Consumer**

消息订阅者，也成为消息消费者，负责向 Kafka Broker 读取消息并进行消费。

➢ **消费组（Consumer Group）**

一类 Consumer 的集合名称，这类 Consumer 通常消费一类消息，且消费逻辑一致。Consumer Group 和 Topic 的关系是 N：N，同一个 Consumer Group 可以订阅多个 Topic，同一个 Topic 也可以被多个 Consumer Group 订阅。

Broker

kafka 集群包含一个活多个消息处理服务器，该服务器成为 Broker，提供数据刷盘等核心功能。可以横向扩展、在线扩容以提高集群性能。

➢ **Topic**

每条发布到 Kafka 集群的消息都有一个主题，这个主题被称为 Topic。通过 Topic 可以对消息进行分类。每个 Topic 可以由一个或多个分区（Partition）组成，存储于一个或多个 Broker 上。

➢ 分区（**partition**）

消息分区是物理上的概念，一个 Topic 可以分为多个 partition，每个 partition 是一个有序的队列。partition 中的每条消息都会被分配一个有序的 id（offset）。

➢ **Producer**

消息和数据生成者，一般为应用调用 API 进行消息生产，并向 Kafka 的一个 Topic 发布消息。

➢ **Consumer**

消息订阅者，也成为消息消费者，负责向 Kafka Broker 读取消息并进行消费。

➢ **消费组（Consumer Group）**

一类 Consumer 的集合名称，这类 Consumer 通常消费一类消息，且消费逻辑一致。Consumer Group 和 Topic 的关系是 N：N，同一个 Consumer Group 可以订阅多个 Topic，同一个 Topic 也可以被多个 Consumer Group 订阅。

Broker

kafka 集群包含一个活多个消息处理服务器，该服务器成为 Broker，提供数据刷盘等核心功能。可以横向扩展、在线扩容以提高集群性能。

➢ **Topic**

每条发布到 Kafka 集群的消息都有一个主题，这个主题被称为 Topic。通过 Topic 可以对消息进行分类。每个 Topic 可以由一个或多个分区（Partition）组成，存储于一个或多个

Broker 上。

➢ 分区（**partition**）

消息分区是物理上的概念，一个 Topic 可以分为多个 partition，每个 partition 是一个有序的队列。partition 中的每条消息都会被分配一个有序的 id（offset）。

➢ **Producer**

消息和数据生成者，一般为应用调用 API 进行消息生产，并向 Kafka 的一个 Topic 发布消息。

➢ **Consumer**

消息订阅者，也成为消息消费者，负责向 Kafka Broker 读取消息并进行消费。

➢ **消费组（Consumer Group）**

一类 Consumer 的集合名称 这类 Consumer 通常消费一类消息 且消费逻辑一致 Consumer Group 和 Topic 的关系是 N：N，同一个 Consumer Group 可以订阅多个 Topic，同一个 Topic 也可以被多个 Consumer Group 订阅。

2 购买指南

资源节点

自研资源池

产品规格

目前基础版和高级版规格如下：

产品类型	产品规格
分布式消息服务 KAFKA-高级版本	三节点 8 核 32GB TOPIC 数上限 100 流量峰值 300MB/s 总磁盘范围 1200GB - 6000GB

分布式消息服务 KAFKA -基础版本	三节点 4 核 16G TOPIC 数上限 50 流量峰值 100MB/s 总磁盘范围 600GB - 6000GB

产品价格

实例价格：

实例规格	标准价格（元/月）
100 MB/s	1300
300 MB/s	2100
600 MB/s	4005
1200 MB/s	7270

存储价格：

存储类型	标准价格（元/G/月）
普通 IO（SATA）	0.45

计费项

消息队列的实例规格、存储空间。

计费方式

包年包月

计费方式变更

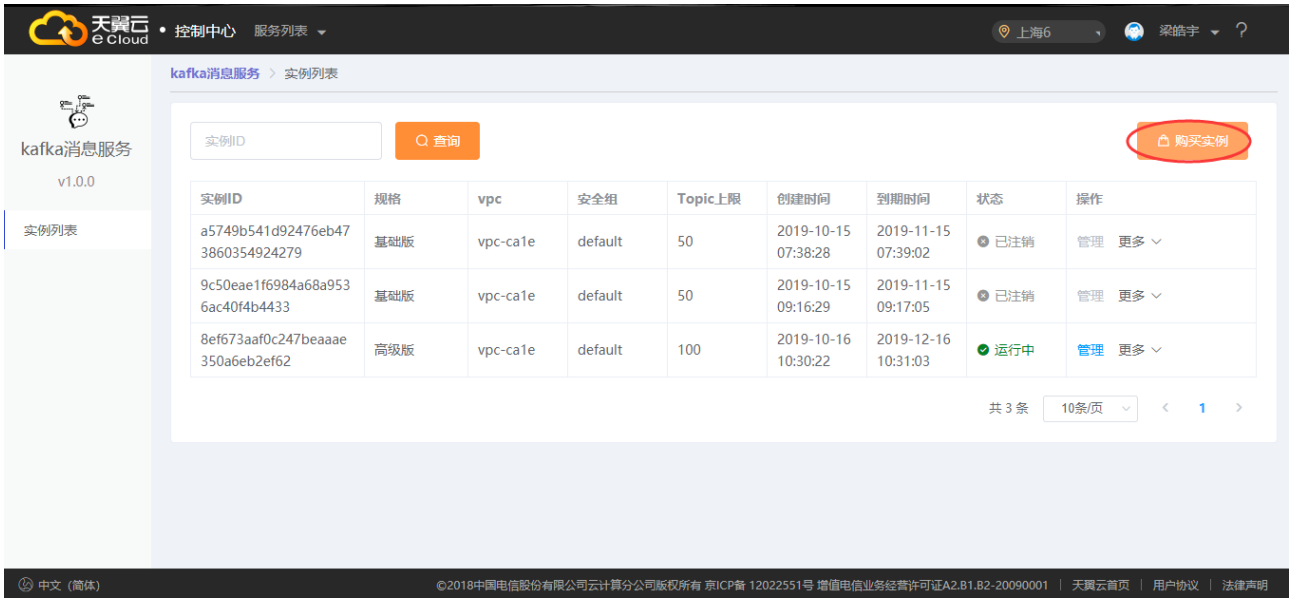
暂无

到期欠费

到期后暂停服务，数据保留，15 天后实例注销，数据将无法找回。

购买

1. **登录管理控制台。**
2. **进入 Kafka 管理控制台。**
3. **在管理控制台右上角单击“地域名称”，选择区域。**
   此处请选择与您的应用服务相同的区域。
4. **点击“购买实例”跳转到购买页面。**



5. 可以选择普通版和高级版。

目前基础版和高级版规格如下：

产品类型	产品规格
分布式消息服务 KAFKA-高级版本	三节点 8 核 32GB TOPIC 数上限 100 流量峰值 300MB/s 总磁盘范围 1200GB - 6000GB
分布式消息服务 KAFKA -基础版本	三节点 4 核 16G TOPIC 数上限 50 流量峰值 100MB/s 总磁盘范围 600GB - 6000GB

存储空间说明：

Kafka 支持多副本存储，副本数量为 3。存储空间包含所有副本存储空间总和，因此，您在创建 Kafka 实例，选择初始存储空间时，建议根据业务消息体积预估以及副本数量选择合适的存储空间。

例如：业务消息体积预估 100GB，则磁盘容量最少应为 100GB*3+ 预留磁盘大小 100GB。

6. 下单购买。

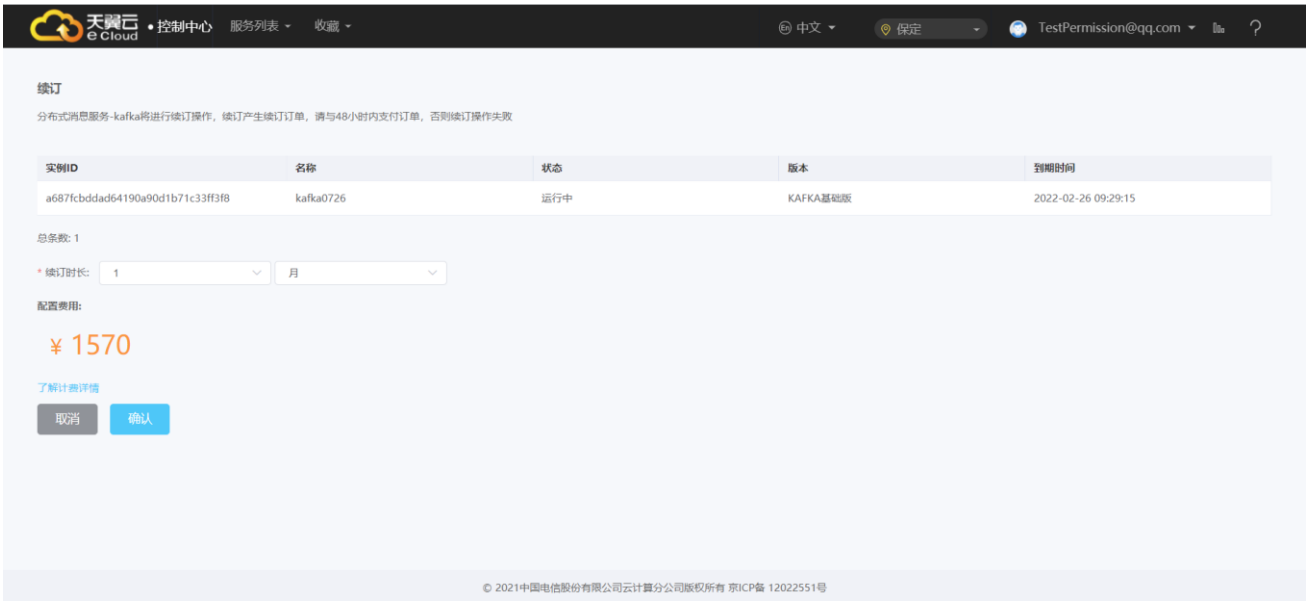
选择理解并接受《公测产品服务协议》，然后下一步订购支付完成购买。

变更

暂无

续订

1. 登录管理控制台。
2. 进入 Kafka 管理控制台。
3. 在实例列表页在操作列，目标实例行点击“更多”“续订”。
4. 在新页面选择续订时长，确认配置费用后，点击“确认”。



5. 在新页面上点击“立即支付”，进入付款页面，在 48 小时内完成付款。
6. 订单处理中，订单完成后将以短信方式通知用户。

退订

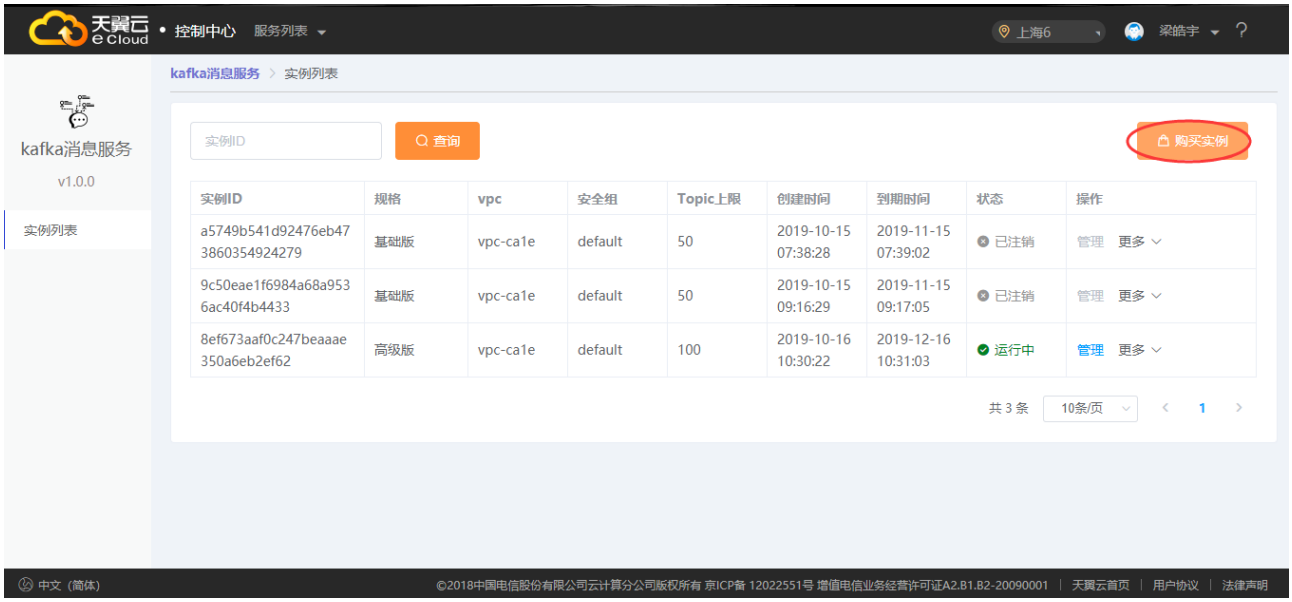
1. 登录管理控制台。
2. 进入 Kafka 管理控制台。
3. 在实例列表页在操作列，目标实例行点击“更多”“退订”。
4. 在弹出的页面中点击“退订”。

3 快速入门

创建分布式消息服务 Kafka

环境准备

1.购买 Kafka 实例

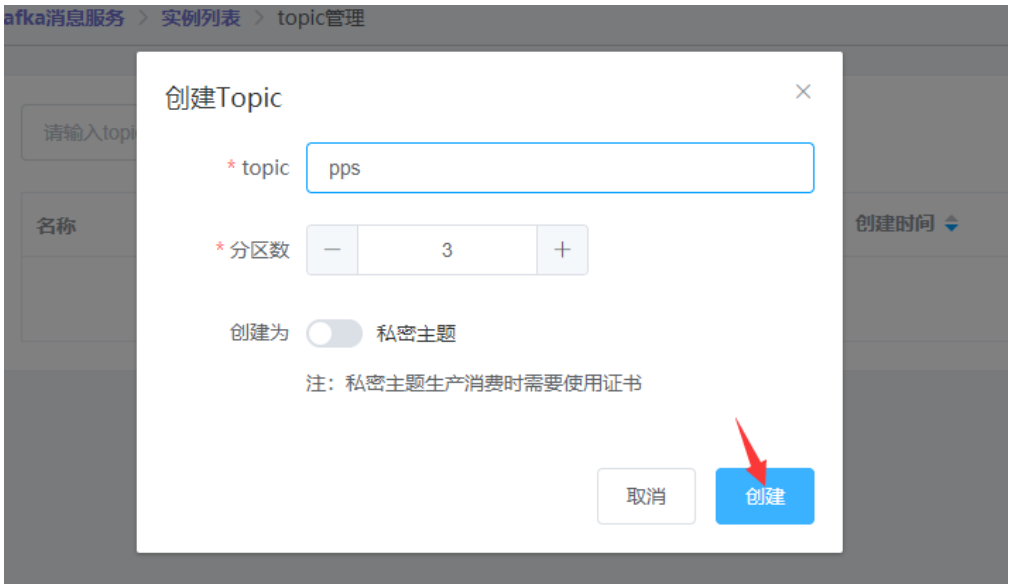


2.下载安装工具 Eclipse3.6.0 以上上版本 或者 IntelliJ ，JDK 1.8.111 以上版本。

创建 Topic

1.点击 Topic 管理后、点击“新建 Topic”。

2.点击“新建 Topic”后，出现如下创建，输入 Topic 名称、分区数、选择是否私密主题。

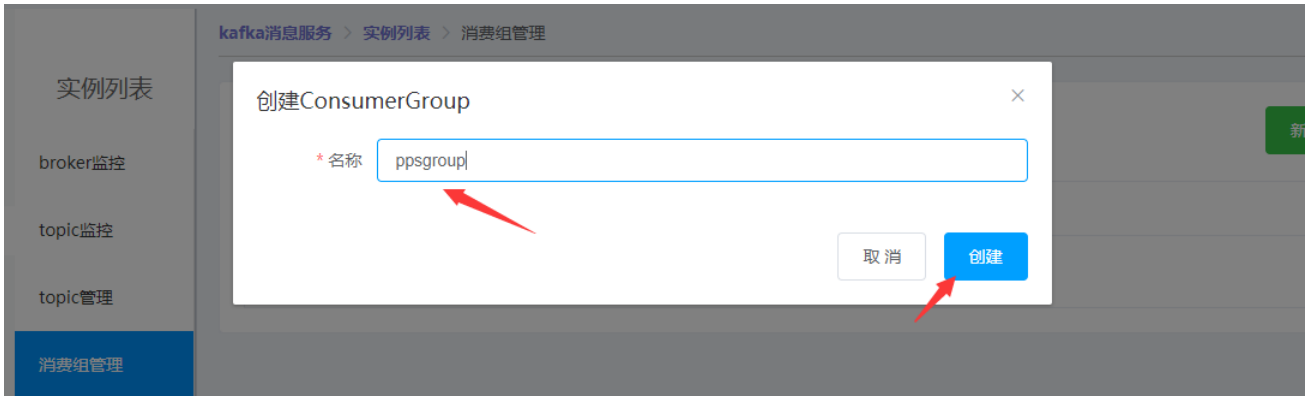


3.创建后的 Topic 出现在列表，可点击“生成拨测”来测试 Topic 是否正常

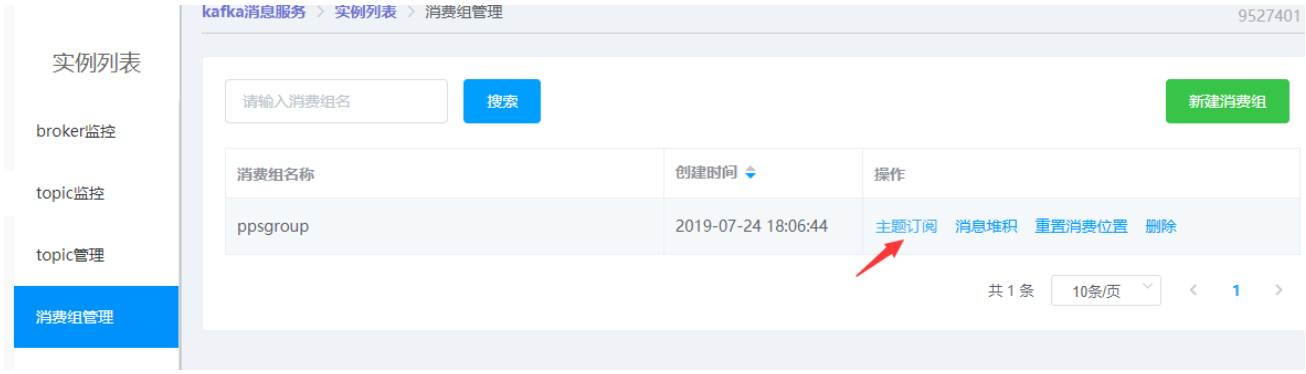
创建消费组

1.点击消费组管理后、点击“新建消费组”。

2.点击“新建消费组”后，输入消费组名称，点击创建。



3.创建后的消费组出现在消费组列表，点击“主题订阅”来订阅 step2 的主题，并且可通过“消费拨测”来测试是否正常消费。



编译运行 Demo Java 工程

Tips:　需要 kafka-clients 引入依赖：

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
</dependency>
```

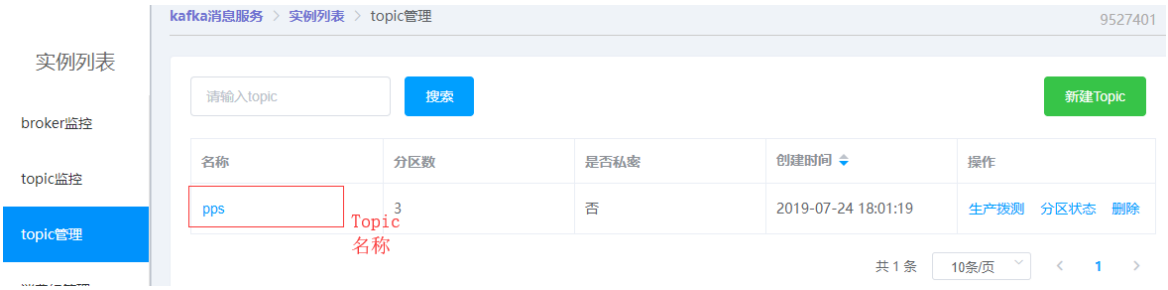
示例代码：

KafkaDemo.java

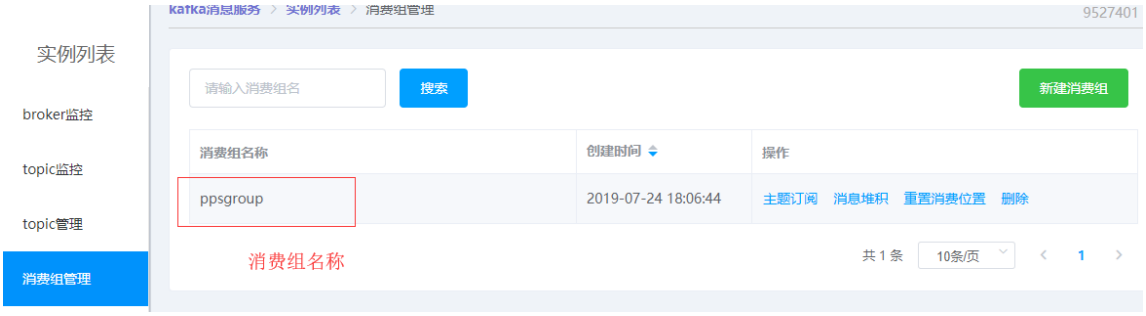
1. 从控制台获取以下信息：

连接地址：

Topic 名称：



消费组名称：



2. 在实例代码中替换以上信息即可实现消息。

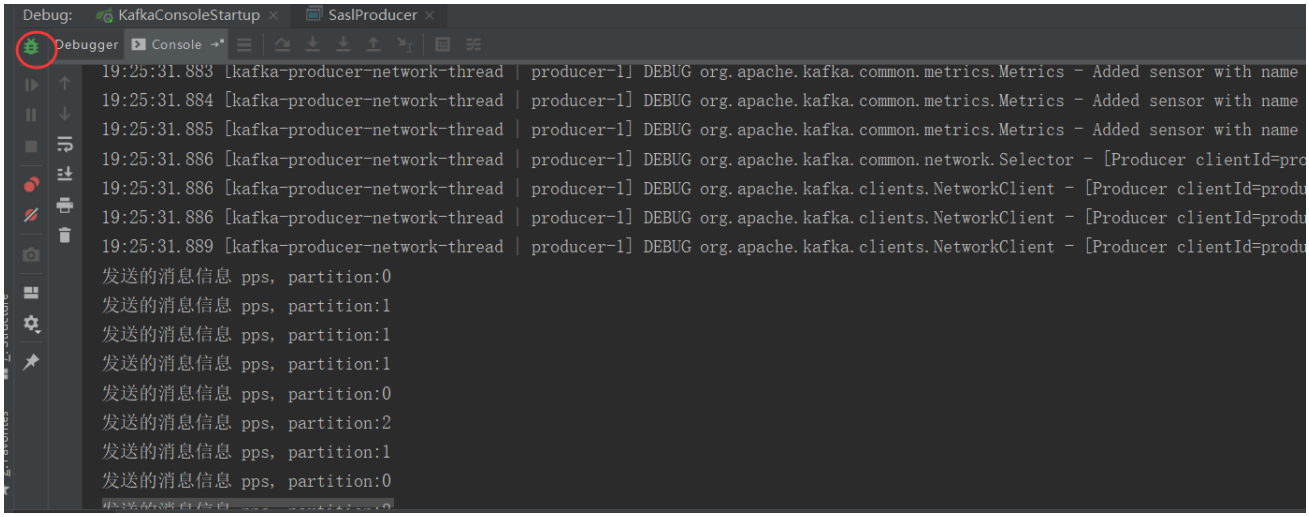
```java
private static void testPlainProducer() throws InterruptedException {
    Properties props = new Properties();
    props.put("bootstrap.servers", "192.168.90.139:8090");  //kafka server  ,192.168.1.159:8091
    props.put("acks", "all");
    props.put("retries", 1);
    props.put("batch.size", 1684);
    props.put("linger.ms", 0);
    props.put("buffer.memory", 33554432); // buffer空间32M
    props.put("request.timeout.ms", 1000);
    props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
    props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

    Producer<String, String> producer = new KafkaProducer<>(props);
    int index = 0;
    while(true) {
        String dvalue = "hello " ;
        ProducerRecord record = new ProducerRecord<>( topic: "pps",  key: "pps"+index++, dvalue);
        producer.send(record, new Callback() {
            @Override
            public void onCompletion(RecordMetadata paramRecordMetadata, Exception paramException) {
                if (paramRecordMetadata == null) {
                    System.out.println("paramRecordMetadata is null ");
                    paramException.printStackTrace();
                    return;
                }
                System.out.println("发送的消息信息 " + paramRecordMetadata.topic() + ", partition:" + paramRecordMetadata.partition
            }
        });
        TimeUnit.SECONDS.sleep( timeout: 1);
    }
      producer.close();
}
```

连接地址

消息内容

主题名称

点击启动后成功发送消息。



3. 同样在实例代码中替换以上信息即可消费消息。

```java
private static void testPlainConsumer() throws InterruptedException {
    Properties properties = new Properties();
    properties.put("bootstrap.servers", "10.142.233.65:9092");
    properties.put("group.id", "ppsgroup");
    properties.put("enable.auto.commit", "true");
    properties.put("auto.offset.reset", "earliest");
    properties.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    properties.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    KafkaConsumer<Object, Object> consumer = new KafkaConsumer<>(properties);
    consumer.subscribe(Arrays.asList("pps"));
    while (true) {
        ConsumerRecords<Object, Object> records = consumer.poll( timeout: 100);
        records.forEach(record->{
            String format = String.format("offset = %d, key = %s, value = %s", record.offset(), record.key(), record.value());
            System.out.println(format);
        });
        TimeUnit.SECONDS.sleep( timeout: 1);
    }
}
```

连接地址

消费组名称

主题名称

如下图运行代码成功获取刚才发送的消息

4 操作指导

查看实例

1. 登录管理控制台。
2. 进入 Kafka 管理控制台。
3. 当前页面会列出所购买的 Kafka 实例，并查看状态，状态说明如下

表 Kafka 实例状态说明

状态	说明
运行中	Kafka 实例正常运行状态。 在这个状态的实例可以运行您的业务。
已关闭	Kafka 实例处于故障的状态。
变更中	Kafka 实例正在进行规格变更操作。

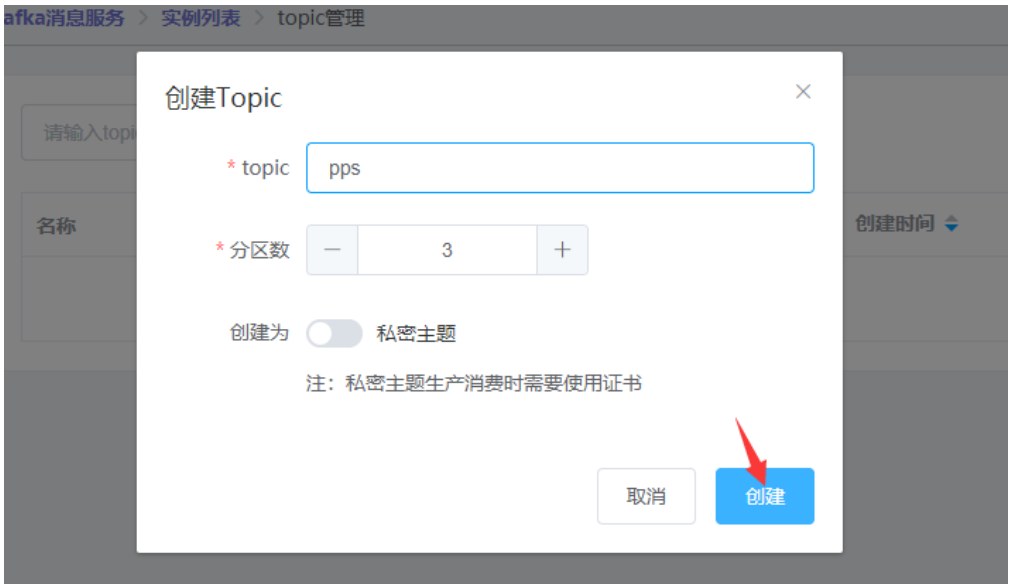
表 Kafka 实例状态说明

状态	说明
变更失败	Kafka 实例处于规格变更失败状态。
暂停	Kafka 专享版实例处于已冻结状态，用户可以在“更多”中续费开启冻结的 Kafka 实例。
注销	Kafka 实例已经过期并关闭，需要重新购买实例。

创建 Topic

1. 登录管理控制台。
2. 进入 Kafka 管理控制台。
3. 在实例列表页在操作列，目标实例行点击“管理”。
4. 点击“Topic 管理”后、点击“新建 Topic”。

5. 点击“新建 Topic”后，出现如下创建，输入 Topic 名称、分区数、选择是否私密主题，详细参数见 Topic 参数说明。



1.

表 Topic 参数说明

参数	说明
Topic 名称	Topic 名称只能包含 a~z，A~Z，0-9，-，_，长度为 4~64 的字符串。 创建 Topic 后不能修改名称。
分区数	您可以设置 Topic 的分区数，分区数越大消费的并发度越大。 该参数设置为 1 时，消费消息时会按照先入先出的顺序进行消费。 取值范围：1-20 默认值：3
副本数	每个 Topic 设置副本的数量，Kafka 会自动在每个副本上备份数据，当其中一个 Broker 节点故障时数据依然可用的，副本数越大可靠性越高。 固定值：3
老化时间（小时）	Topic 中的消息超过老化时间后，消息将会被删除，老化的消息无法被消费。 固定值：72 小时
是否私密主题	普通主题可以直接对其生产消费消息，私密主题需要使用证书生产消费主题。

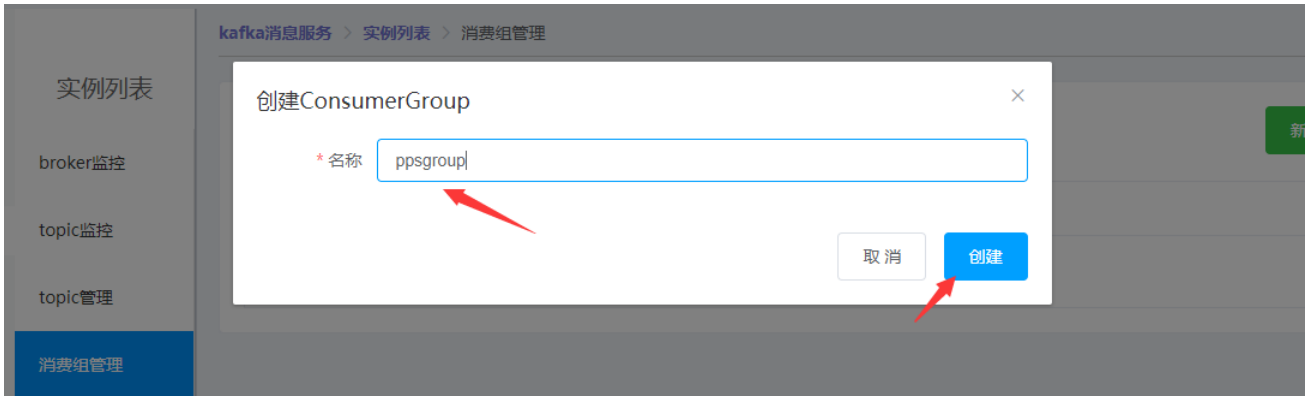
6. **创建后的 Topic 出现在列表，可点击“生成拨测”来测试 Topic 是否正常**

删除 Topic

1. 登录管理控制台。
2. 进入 Kafka 管理控制台。
3. 在实例列表页在操作列，目标实例行点击“管理”。
4. 点击“Topic 管理”后进入 Topic 管理页面
5. 在 Topic 所在行，单击“删除”，并选择确定。

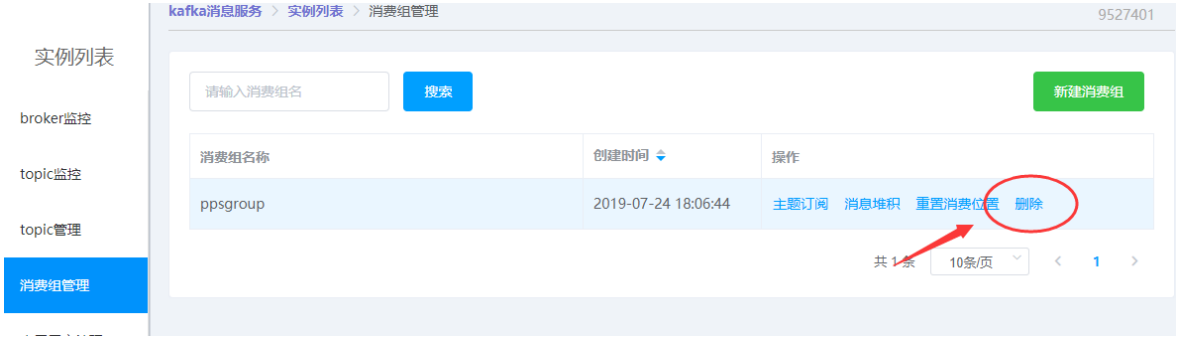
创建消费组

1. 登录管理控制台。
2. 进入 Kafka 管理控制台。
3. 在实例列表页在操作列，目标实例行点击“管理”。
4. 点击“消费组管理”后进入消费组管理页面。
5. 点击“新建消费组”后，输入消费组名称，点击创建。



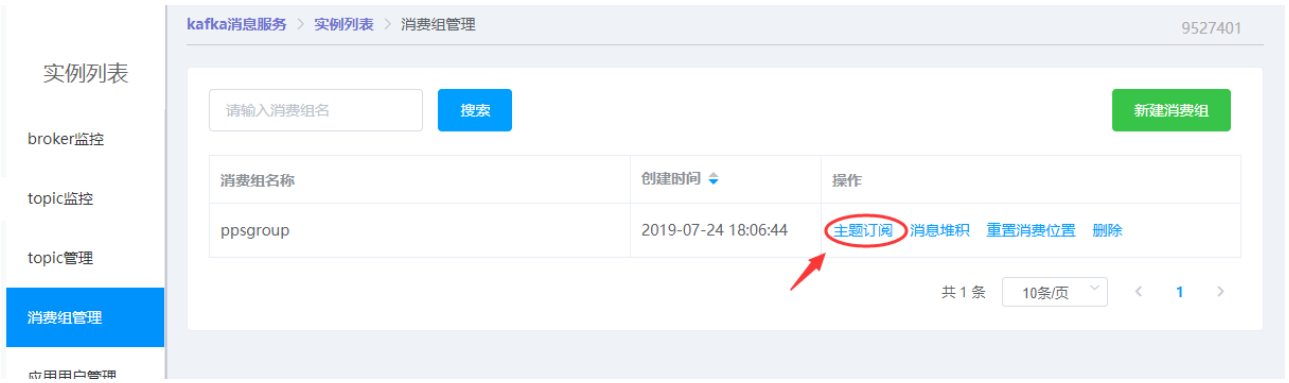
删除消费组

5. 登录管理控制台。
6. 进入 Kafka 管理控制台。
7. 在实例列表页在操作列，目标实例行点击“管理”。
8. 点击“消费组管理”后进入消费组管理页面。
9. 在目标消费组所在行，单击“删除”，并选择确定。



消费组订阅主题

1. 登录管理控制台。
2. 进入 Kafka 管理控制台。
3. 在实例列表页在操作列，目标实例行点击“管理”。
4. 点击“消费组管理”后进入消费组管理页面。
5. 在目标消费组所在行，单击“主题订阅”，并选择确定。

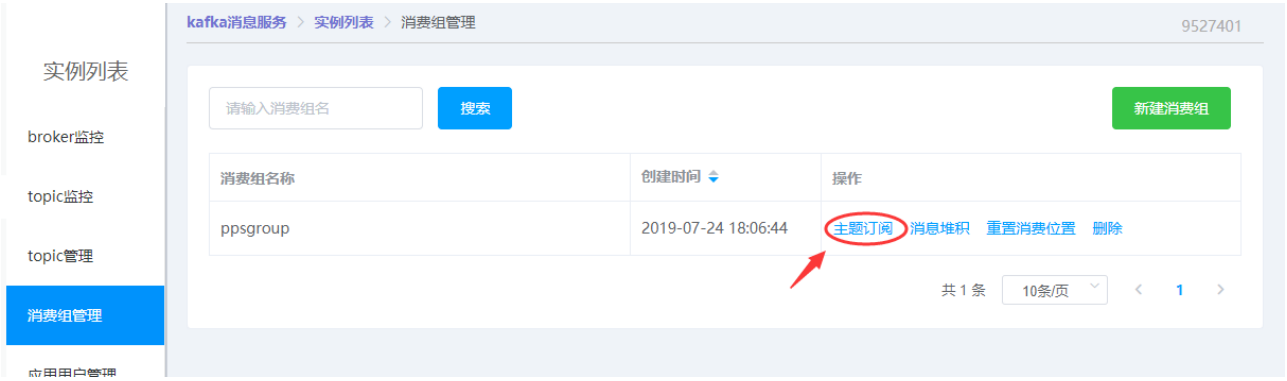


6. **出现主题订阅窗口后，点击“添加主题”。**

7. **出现主题选择窗口后，把目标主题移到右面已选区域，并点击“保存”。**

消费组取消订阅主题

1. 登录管理控制台。
2. 进入 Kafka 管理控制台。
3. 在实例列表页在操作列，目标实例行点击“管理”。
4. 点击“消费组管理”后进入消费组管理页面。
5. 在目标消费组所在行，单击“主题订阅”，并选择确定。

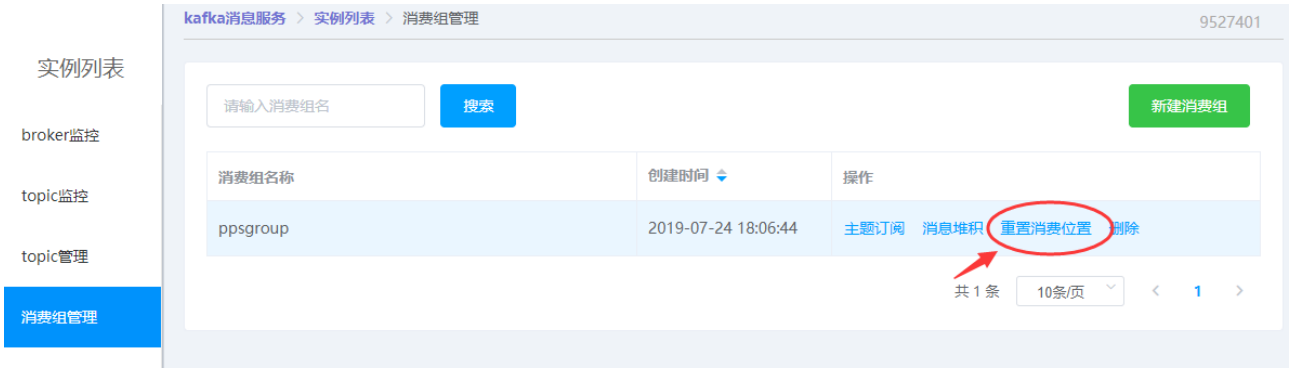


6. 出现主题订阅窗口后，在目标主题行点击“删除”即可取消订阅主题。

消费组消费重置

Tips：目前消费只能重置 72 小时内的消息，可选择 72 小时内时间点重置。

1. 登录管理控制台。
2. 进入 Kafka 管理控制台。
3. 在实例列表页在操作列，目标实例行点击“管理”。
4. 点击“消费组管理”后进入消费组管理页面。
5. 在目标消费组所在行，单击“重置消费位置”。

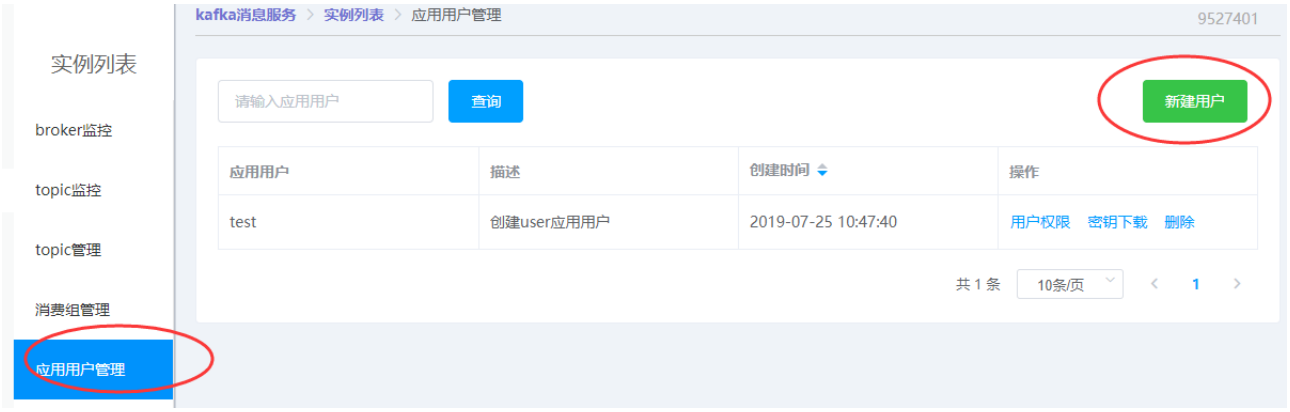


6. 出现重置消费位置窗口后，选择 topic、时间点，然后点击确定，即可重置消费组对应主题的消费位置。

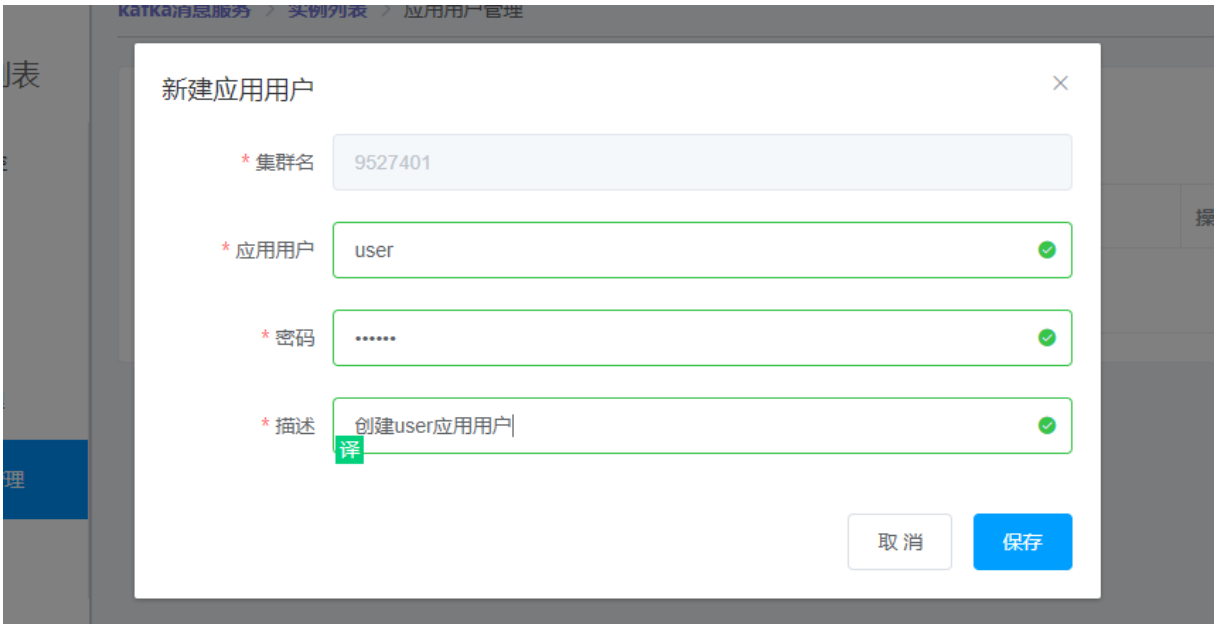
创建应用用户

应用用户：主要用于规定生产消费指定加密主题的策略而需要，例如规定用户 A 可生产消费加密 Topic1，用户 B 可生产消费加密 Topic2，用户 C 可生产消费加密 Topic1、Topic2，则需要为这三个用户创建应用用户，并分配加密主题权限。

1. 登录管理控制台。
2. 进入 Kafka 管理控制台。
3. 在实例列表页在操作列，目标实例行点击“管理”。
4. 点击“应用用户管理”后进入应用用户管理页面，点击新建用户。

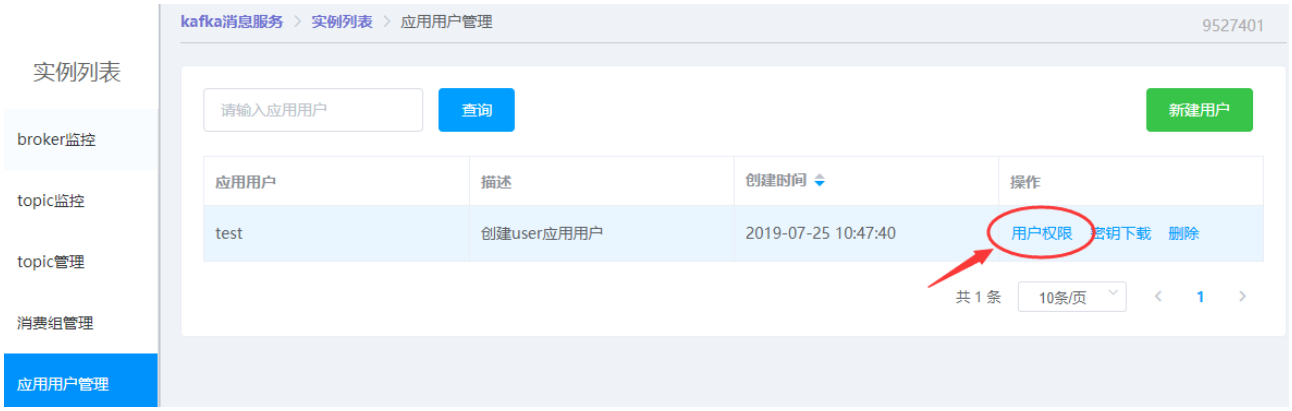


5. 点击“应用用户管理”后进入应用用户管理页面，点击 新建用户。
6. 在 新建用户的窗口中填入集群名、应用用户、密码、描述，然后保存。



管理应用用户生产消费权限

1. 登录管理控制台。
2. 进入 Kafka 管理控制台。
3. 在实例列表页在操作列，目标实例行点击“管理”。
4. 点击“应用用户管理”后进入应用用户管理页面，点击“用户权限”。



5. 在应用用户权限窗口中，可以添加和删除生产权限、消费组权限。

5 最佳实践

生产者实践

本文主要介绍消息队列 Kafka 发布者的最佳实践，从而帮助您更好的使用该产品。
文中的最佳实践基于消息队列 Kafka 的 Java 客户端 ;对于其它语言的客户端，其基本概念与思想是通用的，但实现细节可能有差异，仅供参考。
Kafka 的发送非常简单，示例代码片段如下：

```
Properties props = new Properties();
props.put("bootstrap.servers",    "192.168.90.139:8090");       //kafka
server  ,192.168.1.159:8091
props.put("acks", "all");
props.put("retries", 1);
props.put("batch.size", 1684);
```

```
props.put("linger.ms", 0);
props.put("buffer.memory", 33554432); // buffer 空间 32M
props.put("request.timeout.ms", 1000);
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<String,
String>(props);
int index = 0;
while(true) {
    String dvalue = "hello " ;
    ProducerRecord record = new ProducerRecord<>("pps", "pps"+index++,
dvalue);
    producer.send(record, new Callback() {
        @Override
        public void onCompletion(RecordMetadata paramRecordMetadata,
Exception paramException) {
            if (paramRecordMetadata == null) {
                System.out.println("paramRecordMetadata is null ");
                paramException.printStackTrace();
                return;
            }
            System.out.println(" 发 送 的 消 息 信 息 " +
paramRecordMetadata.topic() + ", partition:" +
paramRecordMetadata.partition());
        }
    });
    TimeUnit.SECONDS.sleep(1);
}
```

Key 和 Value

Kafka 0.10.0.0 的消息字段只有两个：Key 和 Value。Key 是消息的标识，Value 即消息内容。为了便于追踪，重要消息最好都设置一个唯一的 Key。通过 Key 追踪某消息，打印发送日志和消费日志，了解该消息的发送和消费情况。

失败重试

在分布式环境下，由于网络等原因，偶尔的发送失败是常见的。导致这种失败的原因有可能是消息已经发送成功，但是 Ack 失败，也有可能是确实没发送成功。

消息队列 Kafka 是 VIP 网络架构，会主动掐掉空闲连接（30 秒没活动），也就是说，不是一直活跃的客户端会经常收到“connection rest by peer”这样的错误，因此建议都考虑重试消息发送。

异步发送

发送接口是异步的；如果你想得到发送的结果，可以调用 metadataFuture.get(timeout, TimeUnit.MILLISECONDS)。

线程安全

Producer 是线程安全的，且可以往任何 Topic 发送消息。通常情况下，一个应用对应一个 Producer 就足够了。

Acks

Acks 的说明如下：

- acks=0，表示无需服务端的 Response，性能较高，丢数据风险较大；
- acks=1，服务端主节点写成功即返回 Response，性能中等，丢数据风险中等，主节点宕机可能导致数据丢失；
- acks=all，服务端主节点写成功，且备节点同步成功，才返回 Response，性能较差，数据较为安全，主节点和备节点都宕机才会导致数据丢失。

一般建议选择 acks=1，重要的服务可以设置 acks=all。

Batch

Batch 的基本思路是：把消息缓存在内存中，并进行打包发送。Kafka 通过 Batch 来提高吞吐，但同时也会增加延迟，生产时应该对两者予以权衡。
在构建 Producer 时，需要考虑以下两个参数：

- batch.size：发往每个分区（Partition）的消息缓存量（消息内容的字节数之和，不是条数）达到这个数值时，就会触发一次网络请求，然后客户端把消息真正发往服务器；
- linger.ms：每条消息待在缓存中的最长时间。若超过这个时间，就会忽略 batch.size 的限制，然后客户端立即把消息发往服务器。

由此可见，Kafka 客户端什么时候把消息真正发往服务器，是通过上面两个参数共同决定的：batch.size 有助于提高吞吐，linger.ms 有助于控制延迟。您可以根据具体业务需求进行调整。

OOM

结合 Kafka 的 Batch 设计思路，Kafka 会缓存消息并打包发送，如果缓存太多，则有可能造成 OOM（Out of Memory）。

- buffer.memory：所有缓存消息的总体大小超过这个数值后，就会触发把消息发往服务器。此时会忽略 batch.size 和 linger.ms 的限制。
- buffer.memory 的默认数值是 32 MB，对于单个 Producer 来说，可以保证足够的性能。需要注意的是，如果你在同一个 JVM 中启动多个 Producer，那么每个 Producer 都有可能占用 32 MB 缓存空间，此时便有可能触发 OOM。
- 在生产时，一般没有必要启动多个 Producer；如果特殊情况需要，则需要考虑 buffer.memory 的大小，避免触发 OOM。

消费者实践

Kafka 的消费者示例代码片段如下：

```
Properties properties = new Properties();
properties.put("bootstrap.servers",
"192.168.90.139:8090,192.168.90.41:8090,192.168.90.42:8090");
properties.put("group.id", "ppsgroup");
properties.put("enable.auto.commit", "true");
properties.put("auto.offset.reset", "earliest");
properties.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
properties.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<Object, Object> consumer = new
KafkaConsumer<>(properties);
consumer.subscribe(Arrays.asList("pps"));
while (true) {
    ConsumerRecords<Object, Object> records = consumer.poll(100);
    records.forEach(record->{
        String format = String.format("offset = %d, key = %s, value
= %s", record.offset(), record.key(), record.value());
        System.out.println(format);
    });
    TimeUnit.SECONDS.sleep(1);
}
```

负载均衡

每个 Consumer Group 可以包含多个消费实例，即可以启动多个 Kafka Consumer，并把参数 group.id 设置成相同的值。属于同一个 Consumer Group 的消费实例会负载消费订阅的 Topic。

举例：Consumer Group A 订阅了 Topic A，并开启三个消费实例 C1、C2、C3，则发送到 Topic A 的每条消息最终只会传给 C1、C2、C3 的某一个。Kafka 默认会均匀地把消息传给各个消息实例，以做到消费负载均衡。

Kafka 负载消费的内部原理是，把订阅的 Topic 的分区，平均分配给各个消费实例。因此，消费实例的个数不要大于分区的数量，否则会有实例分配不到任何分区而处于空跑状态。这个负载均衡发生的时间，除了第一次启动上线之外，后续消费实例发生重启、增加、减少等变更时，都会触发一次负载均衡。

消息队列 Kafka 的每个 Topic 的分区数量默认是 16 个，已经足够满足大部分场景的需求，且云上服务会根据容量调整分区数。

多个订阅

一个 Consumer Group 可以订阅多个 Topic。一个 Topic 也可以被多个 Consumer Group 订阅，且各个 Consumer Group 独立消费 Topic 下的所有消息。

举例：Consumer Group A 订阅了 Topic A，Consumer Group B 也订阅了 Topic A，则发送到 Topic A 的每条消息，不仅会传一份给 Consumer Group A 的消费实例，也会传一份给 Consumer Group B 的消费实例，且这两个过程相互独立，相互没有任何影响。

消费位点

每个 Topic 会有多个分区，每个分区会统计当前消息的总条数，这个称为最大位点 MaxOffset。Kafka Consumer 会按顺序依次消费分区内的每条消息，记录已经消费了的消息条数，称为 ConsumerOffset。

剩余的未消费的条数（也称为消息堆积量） = MaxOffset - ConsumerOffset

消费位点提交

Kafka 消费者有两个相关参数：

- enable.auto.commit：默认值为 true。
- auto.commit.interval.ms： 默认值为 1000，也即 1s。

这两个参数组合的结果就是，每次 poll 数据前会先检查上次提交位点的时间，如果距离当前时间已经超过参数 auto.commit.interval.ms 规定的时长，则客户端会启动位点提交动作。

因此，如果将 enable.auto.commit 设置为 true，则需要在每次 poll 数据时，确保前一次 poll 出来的数据已经消费完毕，否则可能导致位点跳跃。

如果想自己控制位点提交，请把 enable.auto.commit 设为 false，并调用 commit(offsets)函数自行控制位点提交。

消费位点重置

以下两种情况，会发生消费位点重置：

1. 当服务端不存在曾经提交过的位点时（比如客户端第一次上线）
2. 当从非法位点拉取消息时（比如某个分区最大位点是 10，但客户端却从 11 开始拉取消息）

Java 客户端可以通过 auto.offset.reset 来配置重置策略，主要策略有：

1. “latest”，从最大位点开始消费
2. “earliest”，从最小位点开始消费
3. ‘none’，不做任何操作，也即不重置
4.

建议：

1. 强烈建议设置成“latest”，而不要设置成“earliest”，避免因位点非法时从头开始消费，从而造成大量重复
2. 如果是客户自己管理位点，可以设置成”none”
3. 拉取大消息

消费过程是由客户端主动去服务端拉取消息的，在拉取大消息时，需要注意控制拉取速度，注意修改配置：

1. “max.poll.records”，如果单条消息超过 1MB，建议这里设置为 1.
2. “fetch.max.bytes”，设置比单条消息的大小略大一点.
3. “max.partition.fetch.bytes”，设置比单条消息的大小略大一点。

拉取大消息的核心是一条一条拉。

消息重复和消费幂等

Kafka 消费的语义是“at least once”， 也就是至少投递一次，保证消息不丢，但是不会保证消息不重复。在出现网络问题、客户端重启时均有可能出现少量重复消息，此时应用消费端如果对消息重复比较敏感（比如说订单交易类），则应该做到消息幂等。

以数据库类应用为例，常用做法是：

- 发送消息时，传入 key 作为唯一流水号 ID；
- 消费消息时，判断 key 是否已经消费过，如果已经消费过了，则忽略，如果没消费过，则消费一次；

当然，如果应用本身对少量消息重复不敏感，则不需要做此类幂等检查。

消费失败

Kafka 是按分区一条一条消息顺序向前推进消费的，如果消费端拿到某条消息后执行消费逻辑失败，比如应用服务器出现了脏数据，导致某条消息处理失败，等待人工干预，那么有以下两种处理方式：

- 失败后一直尝试再次执行消费逻辑。这种方式有可能造成消费线程阻塞在当前消息，无法向前推进，造成消息堆积；
- 由于 Kafka 自身没有处理失败消息的设计，实践中通常会打印失败的消息、或者存储到某个服务（比如创建一个 Topic 专门用来放失败的消息），然后定时 check 失败消息的情况，分析失败原因，根据情况处理。

消费延迟

Kafka 的消费机制是由客户端主动去服务端拉取消息进行消费的。因此，一般来说，如果客户端能够及时消费，则不会产生较大延迟。如果产生了较大延迟，请先关注是否有堆积，并注意提高消费速度。

消费阻塞以及堆积

消费端最常见的问题就是消费堆积，最常造成堆积的原因是：

- 消费速度跟不上生产速度，此时应该提高消费速度，详见下一节《提高消费速度》；
- 消费端产生了阻塞。

消费端拿到消息后，执行消费逻辑，通常会执行一些远程调用，如果这个时候同步等待结果，则有可能造成一直等待，消费进程无法向前推进。

消费端应该竭力避免堵塞消费线程，如果存在等待调用结果的情况，建议设置等待的超时时间，超时后作消费失败处理。

提高消费速度

提高消费速度有以下两个办法：

- 增加 Consumer 实例个数
- 增加消费线程

增加 Consumer 实例

可以在进程内直接增加（需要保证每个实例对应一个线程，否则没有太大意义），也可以部署多个消费实例进程；需要注意的是，实例个数超过分区数量后就不再能提高速度，将会有消费实例不工作。

增加消费线程

增加 Consumer 实例本质上也是增加线程的方式来提升速度，因此更加重要的性能提升方式是增加消费线程，最基本的步骤如下：

- 定义一个线程池；
- Poll 数据；
- 把数据提交到线程池进行并发处理；
- 等并发结果返回成功后，再次 poll 数据执行。

消息过滤

Kafka 自身没有消息过滤的语义。实践中可以采取以下两个办法：

- 如果过滤的种类不多，可以采取多个 Topic 的方式达到过滤的目的；
- 如果过滤的种类多，则最好在客户端业务层面自行过滤。

实践中请根据业务具体情况进行选择，也可以综合运用上面两种办法。

消息广播

Kafka 自身没有消息广播的语义，可以通过创建不同的 Consumer Group 来模拟实现。

订阅关系

同一个 Consumer Group 内，各个消费实例订阅的 Topic 最好保持一致，避免给排查问题带来干扰。

通过认证生产与消费加密主题的消息。

分布式消息 Kafka 使用 SASL 认证协议来实现身份验证的能力，加密的主题需要经过身份校验才能正常地消费和生产消息。

创建用户并配置用户权限

进入应用用户管理界面，新建用户，并给用户添加主题和消费组的权限，并且下载密钥。详

细操作可以查看用户指南。

运行生产者客户端，如下是 Java 客户端代码示例：
注意修改内容（用户、密码、接入地址、主题名称）

```
private static void testPlainSaslProducer() throws Exception {
    Properties props = new Properties();
    // 填写应用用户密码
    String username="user";
    String password="password";
    //注意！密码需要 md5
    password = DigestUtils.md5DigestAsHex(password.getBytes());
    String
template="org.apache.kafka.common.security.scram.ScramLoginModule
required " +
```

```
            "username=\"%s\" password=\"%s\";";
    String jaasCfg = String.format(template, username, password);
    props.put("sasl.mechanism", "SCRAM-SHA-512");
    props.put("sasl.jaas.config",jaasCfg);
    props.put("security.protocol", "SASL_PLAINTEXT");
    // 填写 sasl 接入地址
    props.put("bootstrap.servers", "!sasl_address!");
    props.put("acks", "0");
    props.put("retries", 3);
    props.put("batch.size", 1684);
    props.put("linger.ms", 100);
    props.put("buffer.memory", 33554432); // buffer 空间 32M
    props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
    props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
    Producer<String, String> producer = new KafkaProducer<String,
String>(props);

    int index = 0;
    TimeUnit.SECONDS.sleep(2);
    while (true) {
        String dvalue = "hello kafka";
        // 填写主题和消息体内容
        ProducerRecord record = new ProducerRecord<>("topicName",
"pps200" + index++, dvalue);
        producer.send(record, new Callback() {
            @Override
            public void onCompletion(RecordMetadata
paramRecordMetadata, Exception paramException) {
                if (paramRecordMetadata == null) {
                    System.out.println("paramRecordMetadata is null
");
                    paramException.printStackTrace();
                    return;
                }
                System.out.println("发送的消息信息 " +
paramRecordMetadata.topic() + ", partition:" +
paramRecordMetadata.partition());
            }
        });
        TimeUnit.SECONDS.sleep(1);
```

```
    }
}
```

运行消费者客户端，如下是 Java 客户端代码示例：
注意修改内容（用户、密码、接入地址、消费组名称、主题名称）

```
private static void testPlainSaslConsumer() throws Exception {
    Properties props = new Properties();
    // 填写应用用户密码
    String username="user";
    String password="password";
    //注意！密码需要 md5
    password = DigestUtils.md5DigestAsHex(password.getBytes());
    String
template="org.apache.kafka.common.security.scram.ScramLoginModule
required " +
            "username=\"%s\" password=\"%s\";";
    String jaasCfg = String.format(template, username, password);
    // 填写 sasl 接入地址
    props.put("bootstrap.servers", "!sasl_address!");
    // 填写消费组名称
    props.put("group.id", "!consumerGroup!");
    props.put("security.protocol", "SASL_PLAINTEXT");
    props.put("sasl.mechanism", "SCRAM-SHA-512");
    props.put("enable.auto.commit", "false");
    props.put("auto.commit.interval.ms", "1000");
    props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
    props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
    props.put("sasl.jaas.config",jaasCfg);
    KafkaConsumer<String, String> consumer = new
KafkaConsumer<>(props);
    // // 填写订阅的 topic
    consumer.subscribe(Arrays.asList("topicName"));
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        System.out.printf("==============>poll size = %d%n",
records.count());
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("offset = %d, key = %s, value = %s
```

```
partition = %s%n", record.offset(), record.key(), record.value(),
record.partition());
        }
        consumer.commitAsync();//手动提交进度
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

6 SDK 参考

生产者

生产者接口

org.apache.kafka.clients.producer.KafkaProducer

1） 发送消息，返回值为 Future<RecordMetadata>

```
public Future<RecordMetadata> send(ProducerRecord<K, V> record);

public Future<RecordMetadata> send(ProducerRecord<K, V> record, Callback callback);
```

创建生产者

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "all");
props.put("retries", 0);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);
```

创建加密生产者

```
        Properties props = new Properties();
        String username="user";
        String password="qwe123";
        //注意！密码需要 md5
        password = DigestUtils.md5DigestAsHex(password.getBytes());
        String      template="org.apache.kafka.common.security.scram.ScramLoginModule
required " +
                "username=\"%s\" password=\"%s\";";
        String jaasCfg = String.format(template, username, password);
        props.put("sasl.mechanism", "SCRAM-SHA-512");
        props.put("sasl.jaas.config",jaasCfg);
        props.put("security.protocol", "SASL_PLAINTEXT");
        props.put("bootstrap.servers", "192.168.90.201:8092");  // 安全接入点地址
        props.put("acks", "0");
        props.put("retries", 3);
        props.put("batch.size", 1684);
        props.put("linger.ms", 100);
        props.put("buffer.memory", 33554432); // buffer 空间 32M
        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        Producer<String, String> producer = new KafkaProducer<String, String>(props);
```

生产者 Property 说明

常量字段	说明
bootstrap.servers	用于建立初始连接到 kafka 集群的"主机/端口对"配置列表

acks	• acks=0 如果设置为 0，则 producer 不会等待服务器的反馈。 • acks=1 如果设置为 1，leader 节点会将记录写入本地日志，并且在所有 follower 节点反馈之前就先确认成功。 • acks=all 如果设置为 all，这就意味着 leader 节点会等待所有同步中的副本确认之后再确认这条记录是否发送完成。
retries	若设置大于 0 的值，则客户端会将发送失败的记录重新发送
batch.size	当将多个记录被发送到同一个分区时， Producer 将尝试将记录组合到更少的请求中。这有助于提升客户端和服务器端的性能。这个配置控制一个批次的默认大小（以字节为单位）。
linger.ms	producer 会将两个请求发送时间间隔内到达的记录合并到一个单独的批处理请求中。
buffer.memory	Producer 用来缓冲等待被发送到服务器的记录的总字节数。
key.serializer	关键字的序列化类，实现以下接口：org.apache.kafka.common.serialization.Serializer 接口。
value.serializer	值的序列化类，实现以下接口：org.apache.kafka.common.serialization.Serializer 接口。

方法调用说明

创建&连接

```
Properties props = new Properties();
 props.put("bootstrap.servers", "localhost:9092");
 props.put("acks", "all");
 props.put("retries", 0);
 props.put("batch.size", 16384);
 props.put("linger.ms", 1);
 props.put("buffer.memory", 33554432);
 props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
 props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

 Producer<String, String> producer = new KafkaProducer<>(props);;
```

关闭连接

```
producer.close();
```

发送消息

函数名：send

功能描述：发送消息，设置 message 相关属性，将消息发送至 broker

入参说明：

参数	类型	是否可为空	说明

record	ProducerRecord	N	消息记录

输出说明：

参数	类型	说明
result	Future<RecordMetadata>	包含消息时间戳，序号，和将要发送到的分区，

使用例子：

```
byte[] key = "key".getBytes();
byte[] value = "value".getBytes(); ProducerRecord<byte[],byte[]> record = new ProducerRecord<byte[],byte[]>("my-topic", key, value)
producer.send(record);
```

发送消息[包含回调]

函数名：send

功能描述：发送消息，设置 message 相关属性，将消息发送至 broker

入参说明：

参数	类型	是否可为空	说明
record	ProducerRecord	N	消息
callback	Callback	N	回调实例

输出说明：

参数	类型	说明
result	Future<RecordMetadata>	包含消息时间戳，序号，和将要发送到的分区，

使用例子：

```
byte[] key = "key".getBytes();
byte[] value = "value".getBytes(); ProducerRecord<byte[],byte[]> record = new
ProducerRecord<byte[],byte[]>("my-topic", key, value)
producer.send(record, new Callback() {
    @Override
    public void onCompletion(RecordMetadata paramRecordMetadata, Exception
paramException) {
        if (paramRecordMetadata == null) {
            System.out.println("paramRecordMetadata is null ");
            paramException.printStackTrace();
            return;
        }
        System.out.println("发送的消息信息 " + paramRecordMetadata.topic() + ",
partition:" + paramRecordMetadata.partition());
    }
});
```

代码示例

非加密方式：

```
Properties props = new Properties();
 props.put("bootstrap.servers", "localhost:9092");
 props.put("acks", "all");
 props.put("retries", 0);
 props.put("batch.size", 16384);
 props.put("linger.ms", 1);
 props.put("buffer.memory", 33554432);
 props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
 props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

 Producer<String, String> producer = new KafkaProducer<>(props);
 for (int i = 0; i < 100; i++)
     producer.send(new ProducerRecord<String, String>("my-topic", Integer.toString(i),
Integer.toString(i)));
```

加密方式：

```
// 加入加密密钥路径配置
System.setProperty("java.security.krb5.conf", "path/to/krb5.conf");
System.setProperty("java.security.auth.login.config", "path/to/kafka_client.conf");
System.setProperty("sun.security.krb5.debug", "true");

Properties props = new Properties();
props.put("security.protocol", "SASL_PLAINTEXT");
props.put("sasl.mechanism", "GSSAPI");
props.put("sasl.kerberos.service.name", "ctg_kafka");
props.put("bootstrap.servers", "sasl_address, ex: localhost:8092");  // 安全接入点地址
props.put("acks", "0");
props.put("retries", 3);
props.put("batch.size", 1684);
props.put("linger.ms", 100);
props.put("buffer.memory", 33554432); // buffer 空间 32M
props.put("request.timeout.ms", 1000);
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<String, String>(props);
ProducerRecord record = new ProducerRecord<>("ppx", "pps200"+index++, dvalue);
producer.send(record, new Callback() {
    @Override
    public void onCompletion(RecordMetadata paramRecordMetadata, Exception paramException) {
        if (paramRecordMetadata == null) {
            System.out.println("paramRecordMetadata is null ");
            paramException.printStackTrace();
            return;
        }
        System.out.println("发送的消息信息  " + paramRecordMetadata.topic() + ", partition:" + paramRecordMetadata.partition());
    }
});
```

消费者接口

org.apache.kafka.clients.producer. KafkaConsumer

1） 拉取消息，返回值为 ConsumerRecords<K, V>

```
ConsumerRecords<K, V> poll(long timeout)
```

创建消费者

```
 Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
```

创建加密消费者

```
        Properties props = new Properties();
      String username="user";
      String password="password";
      //注意！密码需要 md5
      password = DigestUtils.md5DigestAsHex(password.getBytes());
      String     template="org.apache.kafka.common.security.scram.ScramLoginModule
required " +
              "username=\"%s\" password=\"%s\";";
      String jaasCfg = String.format(template, username, password);
```

```
        // sasl 接入地址
        props.put("bootstrap.servers", "!sasl_address!");
        // 消费组名称
        props.put("group.id", "!consumerGroup!");
        props.put("security.protocol", "SASL_PLAINTEXT");
        props.put("sasl.mechanism", "SCRAM-SHA-512");
        props.put("enable.auto.commit", "false");
        props.put("auto.commit.interval.ms", "1000");
        props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        props.put("sasl.jaas.config",jaasCfg);
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
```

消费者 Property 说明

常量字段	说明
bootstrap.servers	用于建立初始连接到 kafka 集群的"主机/端口对"配置列表
acks	• acks=0 如果设置为 0，则 producer 不会等待服务器的反馈。 • acks=1 如果设置为 1，leader 节点会将记录写入本地日志，并且在所有 follower 节点反馈之前就先确认成功。 • acks=all 如果设置为 all，这就意味着 leader 节点会等待所有同步中的副本确认之后再确认这条记录是否发送完成。
retries	若设置大于 0 的值，则客户端会将发送失败的记录重新发送

batch.size	当将多个记录被发送到同一个分区时， Producer 将尝试将记录组合到更少的请求中。这有助于提升客户端和服务器端的性能。这个配置控制一个批次的默认大小（以字节为单位）。
linger.ms	producer 会将两个请求发送时间间隔内到达的记录合并到一个单独的批处理请求中。
buffer.memory	Producer 用来缓冲等待被发送到服务器的记录的总字节数。
key.serializer	关键字的序列化类，实现以下接口：org.apache.kafka.common.serialization.Serializer 接口。
value.serializer	值的序列化类，实现以下接口：org.apache.kafka.common.serialization.Serializer 接口。

方法调用说明

创建&连接

```
 Properties props = new Properties();
 props.put("bootstrap.servers", "localhost:9092");
 props.put("group.id", "test");
 props.put("enable.auto.commit", "true");
 props.put("auto.commit.interval.ms", "1000");
 props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
 props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
 KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
```

拉取消息

函数名：poll

功能描述：接受消息

入参说明：

参数	类型	是否可为空	说明
timeout	long	N	超时时间

输出说明：

参数	类型	说明
result	ConsumerRecords<K, V>	拉取消息

使用例子：

```
ConsumerRecords<String, String> records = consumer.poll(100);
```

代码示例

非加密方式：

```
Properties props = new Properties();
 props.put("bootstrap.servers", "localhost:9092");
 props.put("group.id", "test");
 props.put("enable.auto.commit", "true");
 props.put("auto.commit.interval.ms", "1000");
 props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
 props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
 KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
 consumer.subscribe(Arrays.asList("foo", "bar"));
```

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s%n",
record.offset(),record.key(), record.value());
 }
```

加密方式：

```
System.setProperty("java.security.krb5.conf", "path/to/krb5.conf");
System.setProperty("java.security.auth.login.config", "path/to/kafka_client.conf");
System.setProperty("sun.security.krb5.debug", "true");

Properties properties = new Properties();
properties.put("bootstrap.servers", "sasl_address, ex: localhost:8092"); // 安全接入
地址
properties.put("group.id", "ppxgroup");
properties.put("enable.auto.commit", "true");
properties.put("auto.offset.reset", "earliest");//earliest
properties.put("max.poll.records", 1);//earliest
properties.put("security.protocol", "SASL_PLAINTEXT");
properties.put("sasl.mechanism", "GSSAPI");
properties.put("sasl.kerberos.service.name", "ctg_kafka");
properties.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
properties.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<Object, Object> consumer = new KafkaConsumer<>(properties);
consumer.subscribe(Arrays.asList("ppx"));
       /* 读取数据，读取超时时间为100ms */

ConsumerRecords<Object, Object> records = consumer.poll(100);
records.forEach(record->{
    String format = String.format("offset = %d, key = %s, value = %s", record.offset(),
record.key(), record.value());
    System.out.println("================================="+format);
});
```

7 视频专区

暂无

8 常见问题

计费类

1. 支持哪些付费方式：

支持包年包月

2. 收费依据有哪些：

根据基准带宽和磁盘容量收费

3. 基准带宽可选择哪些：

高级版基准带宽 300MB/s，基础班基础带宽 100MB/s

4. 高级版磁盘容量可选范围是多少：

高级版磁盘容量为 1200GB-6000GB

5. 基础版磁盘容量可选范围是多少：

基础版磁盘容量为 600GB-6000GB

购买类

1. 可以购买哪些版本：

当前可以购买基础版和高级版

2. 到期后如何续费：

在集群列表中点击“续费”，进入购买时长页面，购买成功后自动续费

3. 支持自营资源池与合营资源池吗：

目前支持自营资源池

4. 如何退订实例：

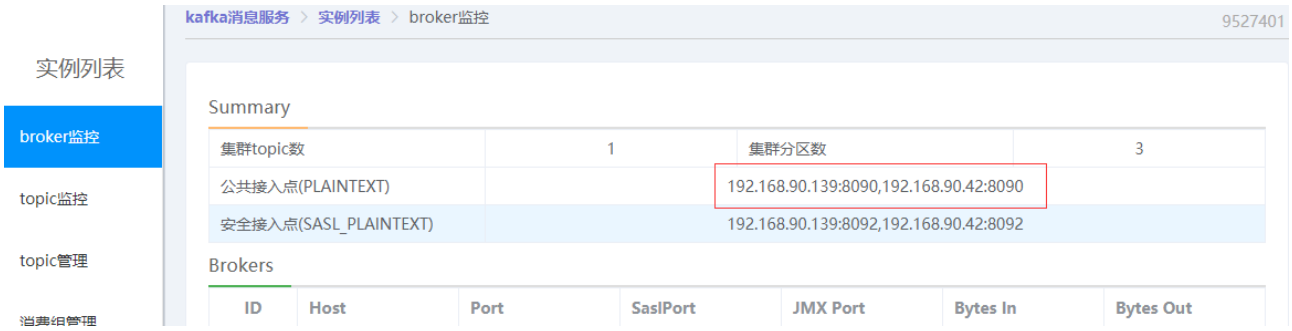
在集群列表中点击“退订”，进入相关页面退订

5. 如何变更实例：

暂不支持变更

系统类

1. 接入地址出现不足三个 ip+端口：



问题：集群三台机器正常运作的情况下，接入点会出现三个 ip:port 连起来，当出现不足三

个时候，说明其中一台机器不正常工作（没出现在接入点的机器）。

解决：尽快联系管理人员查看不正常工作的节点，尽快恢复。

2. 消息在 kafka 保留多长时间？

消息保存 72 小时，超过 72 小时的消息将会被删除。

3. Kafka 可以创建多少个主题？

Kafka 基础版可以创建 50 个主题、Kafka 高级版可以创建 100 个主题。

4. 如果想消费已经被消费过的数据。

问题描述：consumer 是底层采用的是一个阻塞队列，只要一有 producer 生产数据，那 consumer 就会将数据消费。当然这里会产生一个很严重的问题，如果你重启一消费者程序，那你连一条数据都抓不到，但是 log 文件中明明可以看到所有数据都好好的存在。换句话说，一旦你消费过这些数据，那你就无法再次用同一个 groupid 消费同一组数据了。

解决：可在控制台重置消费组消费点（3 天内）。

5. 如何保证消息发布的可靠性？

生产者配置：

- 如果 acks=0，生产者在成功写入消息之前是不会等待任何的来自服务器的响应。可靠性最低、性能最优

- 如果 acks=1，只要集群的首领节点收到消息，生产者就会收到来自服务器成功的响应。
- 如果 acks=all / -1，只有在集群所有的跟随副本都接收到消息后，生产者才会受到一个来自服务器的成功响应。可靠性最高，性能最差。

6. 如何保证保证消息的顺序？

- Kafka 每个 Partition 都是相互独立的，Kafka 只能保证单个 Partition 下的有序。
- 局部有序：当我们所需要的有序其实是针对单个用户的有序，而不要求全局有序。我们可以以用户的 ID 作为 key，确保单个用户一定会被分配到某个固定的 Partition 上（可能会引起数据倾斜问题），这样我们就能够实现单个用户维度的有序了。
- 如果一定要全局的有序，所有消息都使用同一个 key ，这样他们一定会被分配到同一个 Partition 上，这种做法适用于临时性且数据量不大的小需求，消息量大了会有性能压力。

7. 如何选择 Partiton 的数量？

- 在创建 Topic 的时候可以指定 Partiton 数量，也可以在创建完后手动修改。但 Partiton 数量只能增加不能减少。中途增加 Partiton 会导致各个 Partiton 之间数据量的不平等。
- Partition 的数量直接决定了该 Topic 的并发处理能力。但也并不是越多越好。Partition 的数量对消息延迟性会产生影响。

一般建议选择 Broker Num * Consumer Num ，这样平均每个 Consumer 会同时读取 Broker 数目个 Partition ， 这些 Partition 压力可以平摊到每台 Broker 上

8. 如何选择磁盘空间？

存储空间说明：
Kafka 支持多副本存储，副本数量为 3。存储空间包含所有副本存储空间总和，因此，您在创建 Kafka 实例，选择初始存储空间时，建议根据业务消息体积预估以及副本数量选择合适

的存储空间。

例如：业务消息体积预估 100GB，则磁盘容量最少应为 100GB*3+ 预留磁盘大小 100GB。

9. 如何选择实例带宽？

Kafka 实例的网络带宽指单向（读或写）最大带宽。一般建议选择带宽时建议预留 30%，确保您的应用运行更稳定。

- 100MB/s

业务流量为 70M 以内时推荐选用。

- 300MB/s

业务流量为 210M 以内时推荐选用。

9 文档下载

用户使用手册

天翼kafka产品用户使用指南20-8-1

10 相关协议

产品服务协议

https://www.ctyun.cn/h5/home/protocol/10008634

产品服务等级协议

https://www.ctyun.cn/h5/home/protocol/10403556