开课吧 kaikeba.com

分布式消息系统 Kafka

课程讲义

主讲：Reythor 雷

2019

分布式消息系统 Kafka

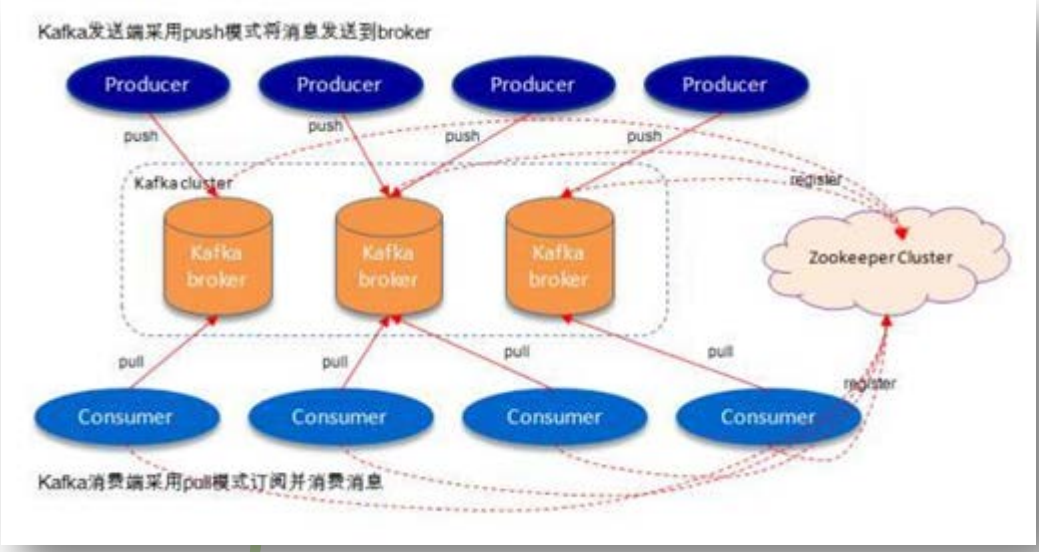
第1章 Kafka 概述

1.1 kafaka 简介

Apache Kafka 是一个快速、可扩展的、高吞吐的、可容错的分布式“发布-订阅”消息系统，使用 Scala 与 Java 语言编写，能够将消息从一个端点传递到另一个端点，较之传统的消息中间件（例如 ActiveMQ、RabbitMQ），Kafka 具有高吞吐量、内置分区、支持消息副本和高容错的特性，非常适合大规模消息处理应用程序。

Kafka 官网： http://kafka.apache.org/

1.2 Kafa 系统架构



1.3 应用场景

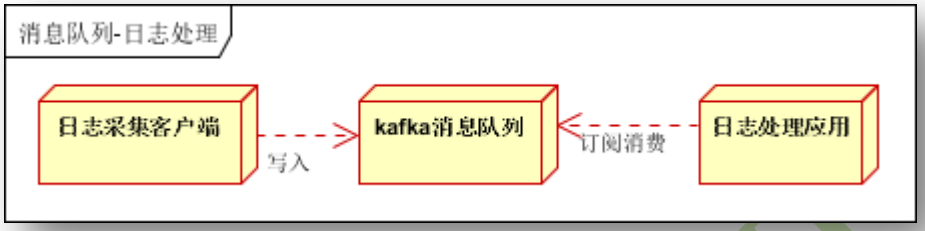
Kafka 的应用场景很多，这里就举几个最常见的场景。

1.3.1 用户的活动追踪

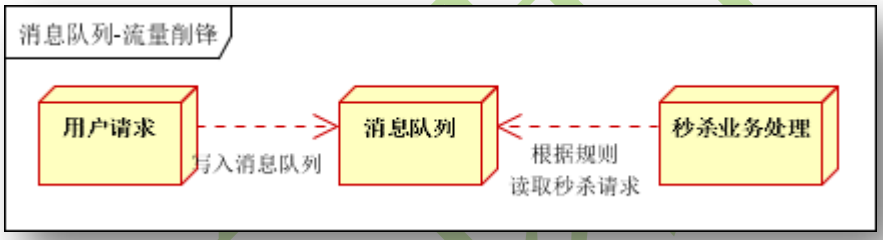
用户在网站的不同活动消息发布到不同的主题中心，然后可以对这些消息进行实时监测

实时处理。当然，也可加载到 Hadoop 或离线处理数据仓库，对用户进行画像。像淘宝、京东这些大型的电商平台，用户的所有活动都是要进行追踪的。

1.3.2 日志聚合



1.3.3 限流削峰



1.4 kafka 高吞吐率实现

Kafka 与其它 MQ 相比，其最大的特点就是高吞吐率。为了增加存储能力，Kafka 将所有的消息都写入到了低速大容的硬盘。按理说，这将导致性能损失，但实际上，kafka 仍可保持超高的吞吐率，性能并未受到影响。其主要采用了如下的方式实现了高吞吐率。

- 顺序读写：Kafka 将消息写入到了分区 partition 中，而分区中消息是顺序读写的。顺序读写要远快于随机读写。
- 零拷贝：生产者、消费者对于 kafka 中消息的操作是采用零拷贝实现的。
- 批量发送：Kafka 允许使用批量消息发送模式。
- 消息压缩：Kafka 支持对消息集合进行压缩。

第2章 Kafka 工作原理与工作过程

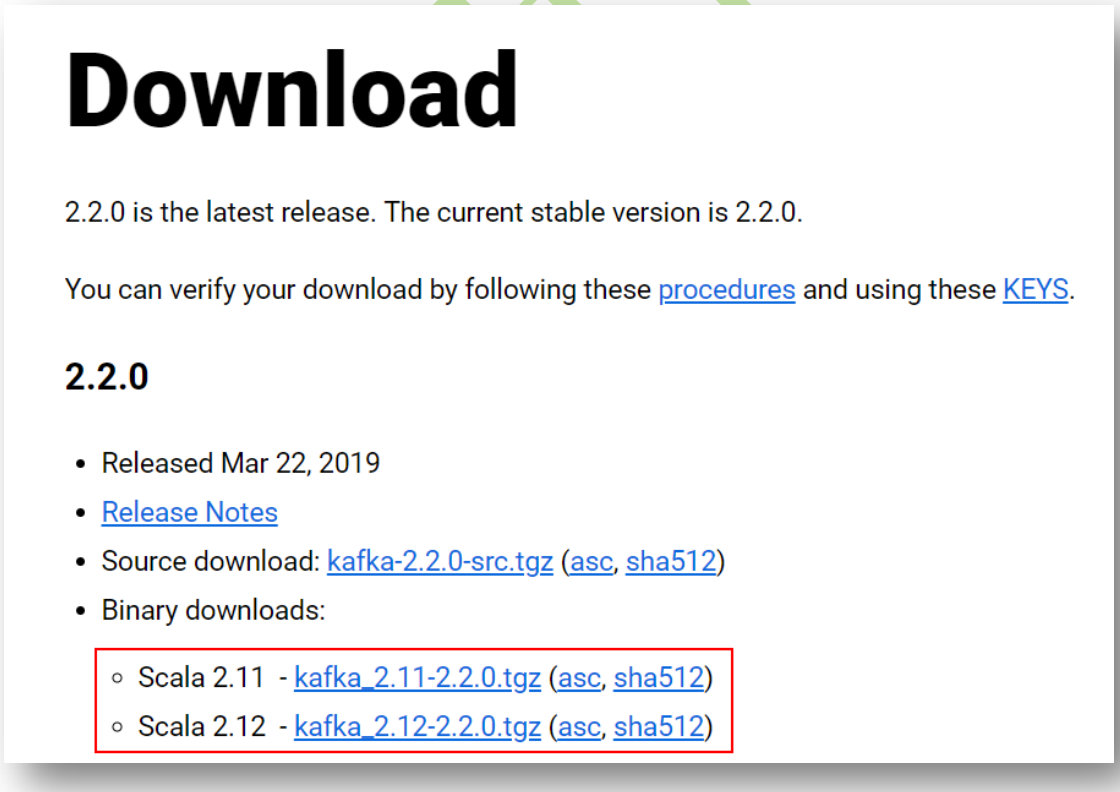
2.1 Kafka 基本原理

2.2 Kafka 工作原理与过程

2.3 Kafka 集群搭建

在生产环境中为了防止单点问题，Kafka 都是以集群方式出现的。下面要搭建一个 Kafka 集群，包含三个 Kafka 主机，即三个 Broker。

2.3.1 Kafka 的下载



Download

2.2.0 is the latest release. The current stable version is 2.2.0.

You can verify your download by following these procedures and using these KEYS.

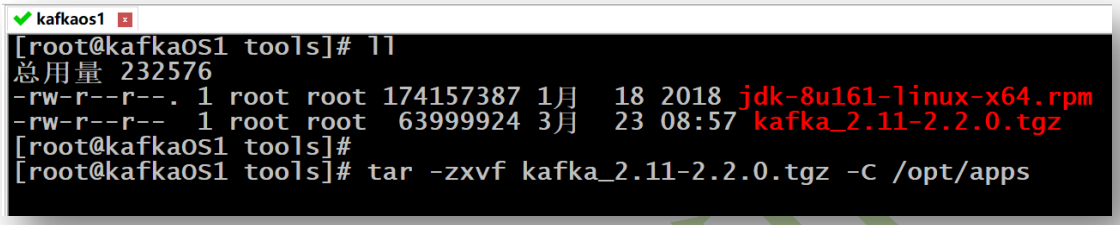
2.2.0

- Released Mar 22, 2019
- Release Notes
- Source download: kafka-2.2.0-src.tgz (asc, sha512)
- Binary downloads:
  - Scala 2.11 - kafka_2.11-2.2.0.tgz (asc, sha512)
  - Scala 2.12 - kafka_2.12-2.2.0.tgz (asc, sha512)

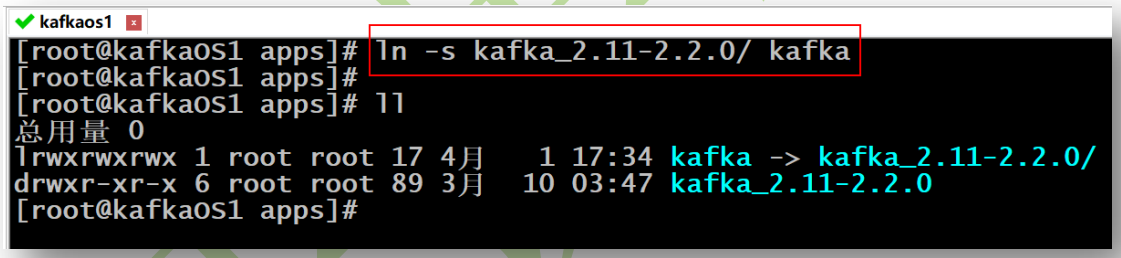
2.3.2 安装并配置第一台主机

（1） 上传并解压

将下载好的 Kafka 压缩包上传至 CentOS 虚拟机，并解压。



（2） 创建软链接



（3） 修改配置文件

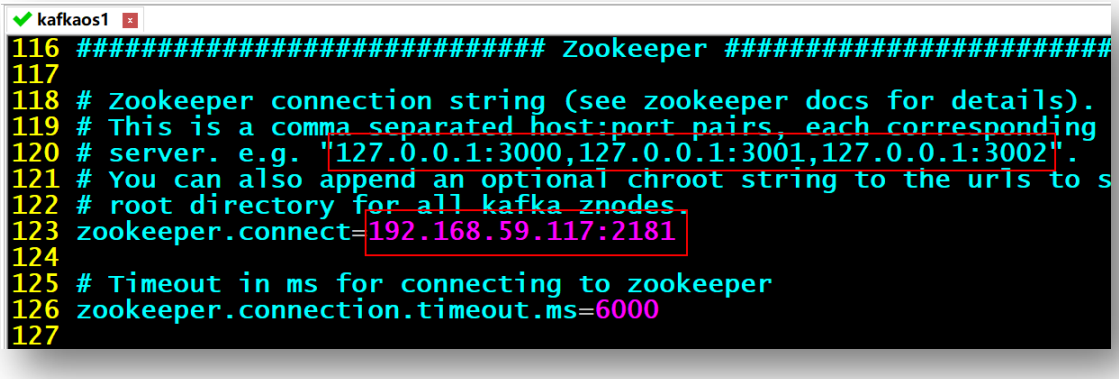
在 kafka 安装目录下有一个 config/server.properties 文件，修改该文件。

kafkaos1

```
17
18 ############################# Server Basics #############
19
20 # The id of the broker. This must be set to a unique int
21 broker.id=0
22
23 ############################# Socket Server Settings ###
24
25 # The address the socket server listens on. It will get
26 # java.net.InetAddress.getCanonicalHostName() if not cor
27 #   FORMAT:
28 #     listeners = listener_name://host_name:port
29 #   EXAMPLE:
30 #     listeners = PLAINTEXT://your.host.name:9092
31 listeners=PLAINTEXT://192.168.59.151:9092
32
33 # Hostname and port the broker will advertise to produce
34 # it uses the value for "listeners" if configured.  Othe
35 # returned from java.net.InetAddress.getCanonicalHostNam
36 #advertised.listeners=PLAINTEXT://your.host.name:9092
37
38 # Maps listener names to security protocols, the default
```

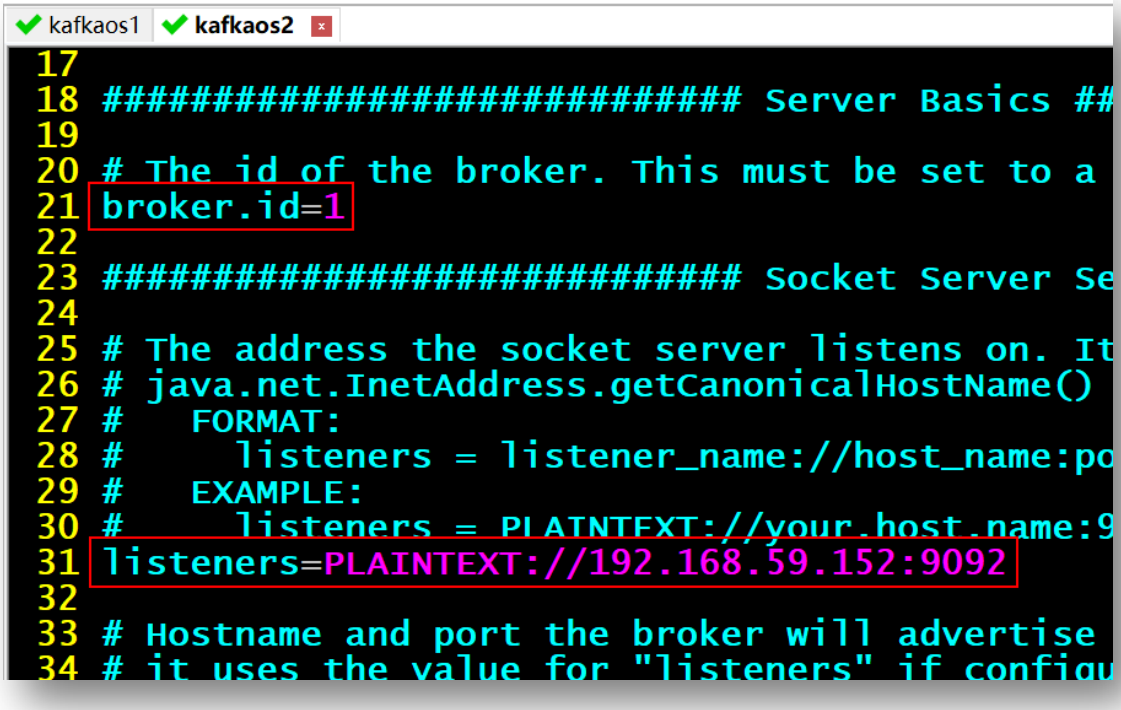
kafkaos1

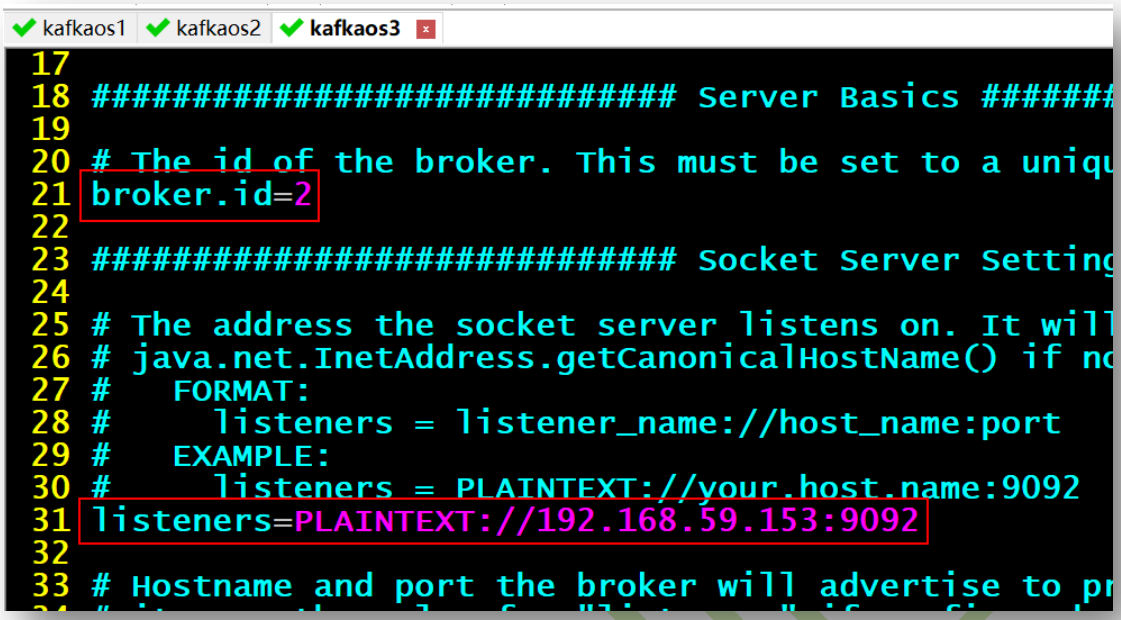
```
56
57 ############################# Log Basics ################
58
59 # A comma separated list of directories under wh
60 log.dirs=/tmp/kafka-logs
61
62 # The default number of log partitions per topic
63 # parallelism for consumption, but this will als
64 # the brokers.
65 num.partitions=1
66
67 # The number of threads per data directory to be
   flushing at shutdown.
68 # This value is recommended to be increased for
   n RAID array.
69 num.recovery.threads.per.data.dir=1
70
```



2.3.3 再克隆两台 Kafka

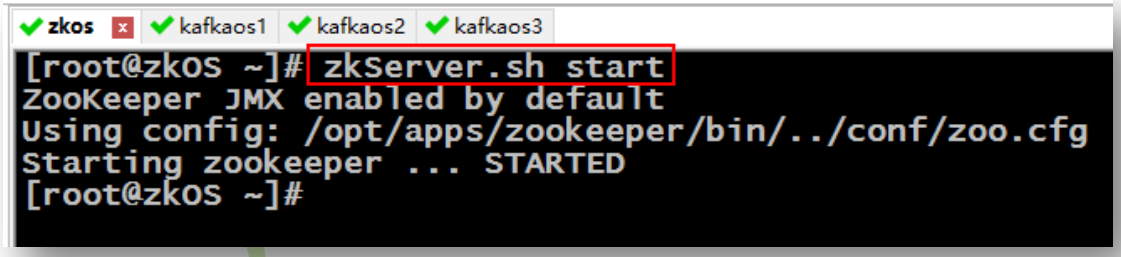
以 kafkaOS1 为母机再克隆两台 Kafka 主机。在克隆完毕后，需要修改 server.properties 中的 broker.id、listeners 与 advertised.listeners。





2.3.4 kafka 的启动与停止

（1） 启动 zookeeper



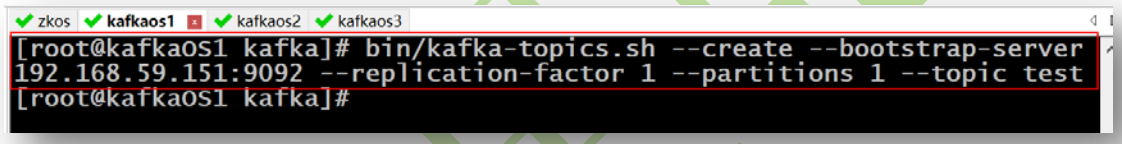
（2） 启动 kafka

在命令后添加-daemon 参数，可以使 kafka 以守护进程方式启动，即不占用窗口。

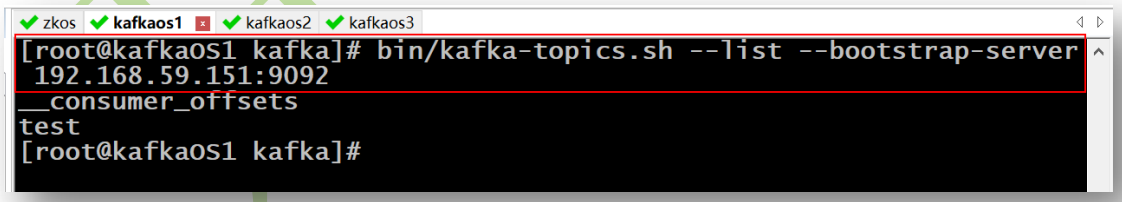
（3） 停止 kafka

2.3.5 kafka 操作

（1） 创建 topic

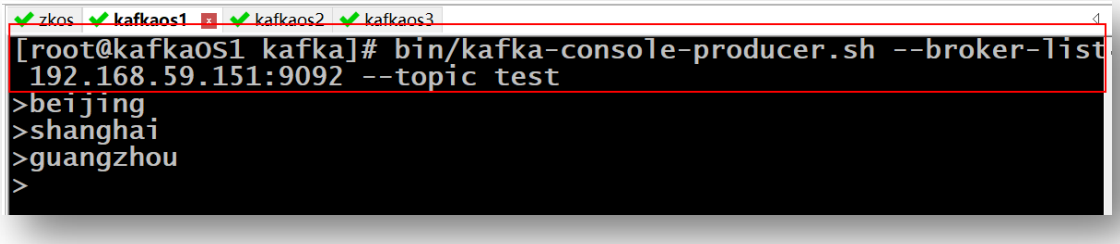


（2） 查看 topic

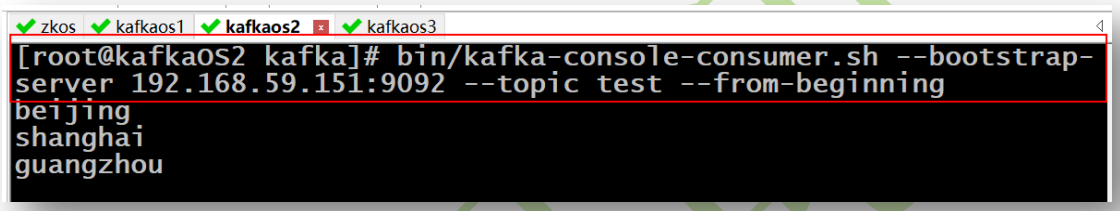


（3） 发送消息

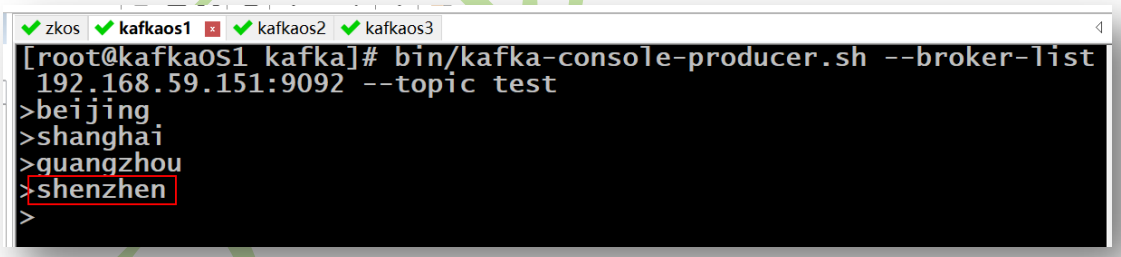
该命令会创建一个生产者，然后由其生产消息。

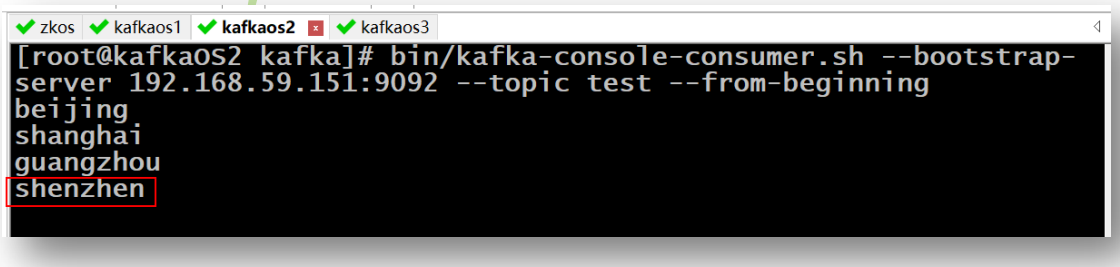


（4） 消费消息

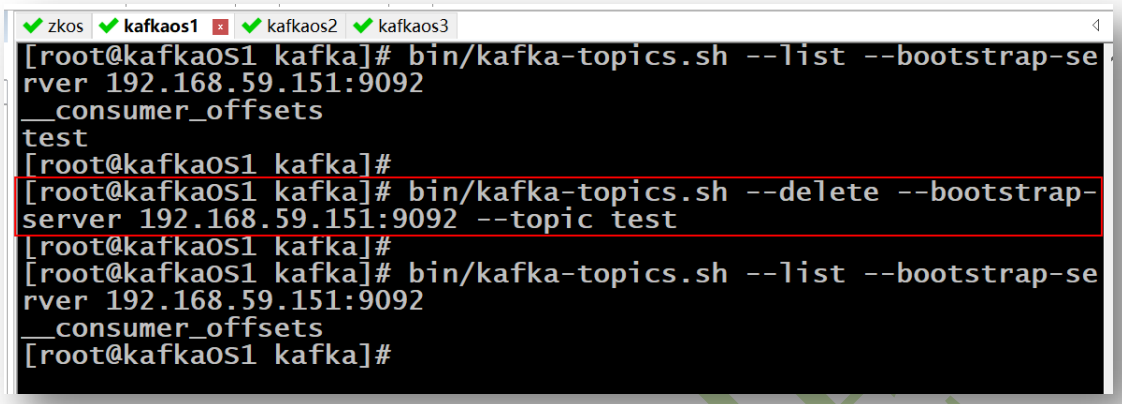


（5） 继续生产消费





（6） 删除 topic



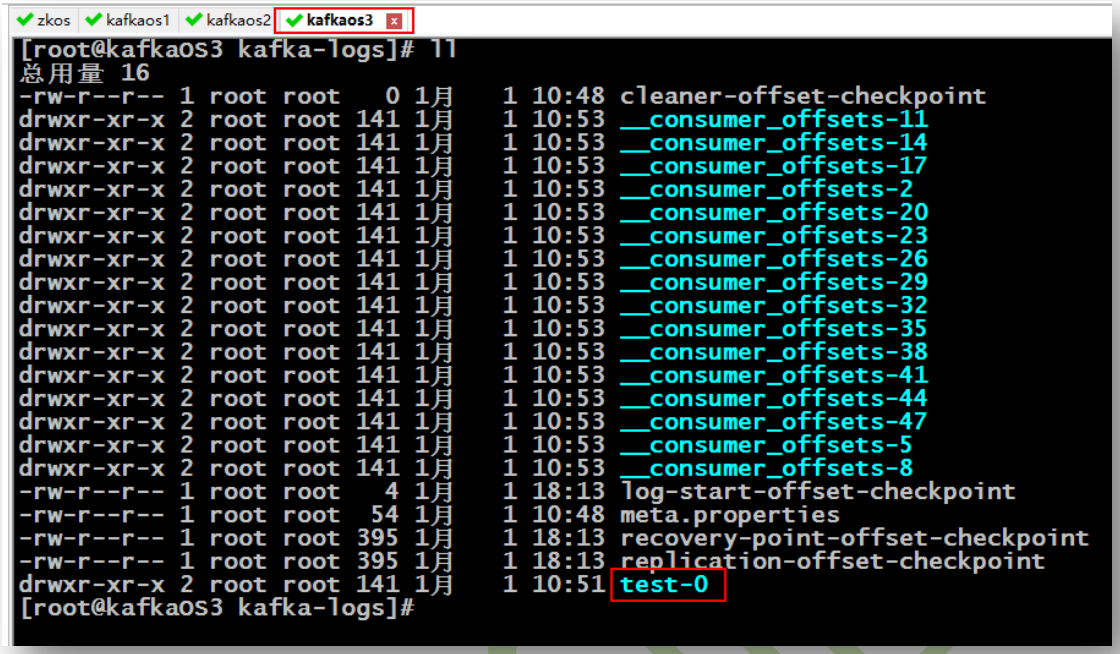
2.4 日志查看

我们这里说的日志不是 Kafka 的启动日志，启动日志在 Kafka 安装目录下的 logs/server.log 中。消息在磁盘上都是以日志的形式保存的。我们这里说的日志是存放在/tmp/kafka_logs 目录中的消息日志，即 partition 与 segment。

2.4.1 查看分区与备份

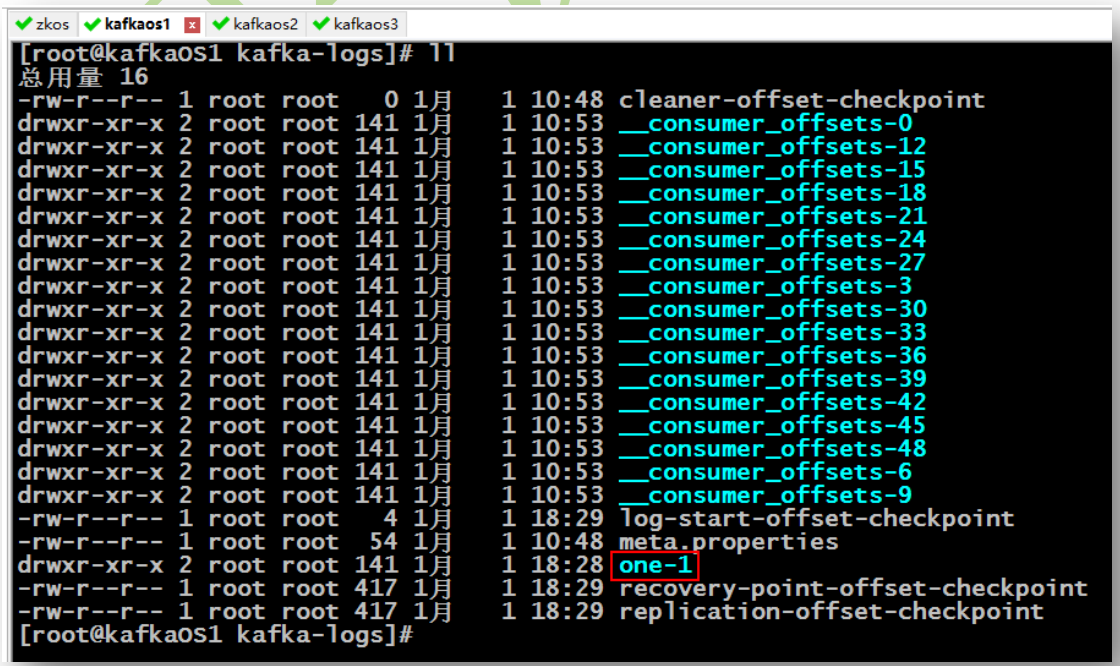
（1） 1 个分区 1 个备份

我们前面创建的 test 主题是 1 个分区 1 个备份。



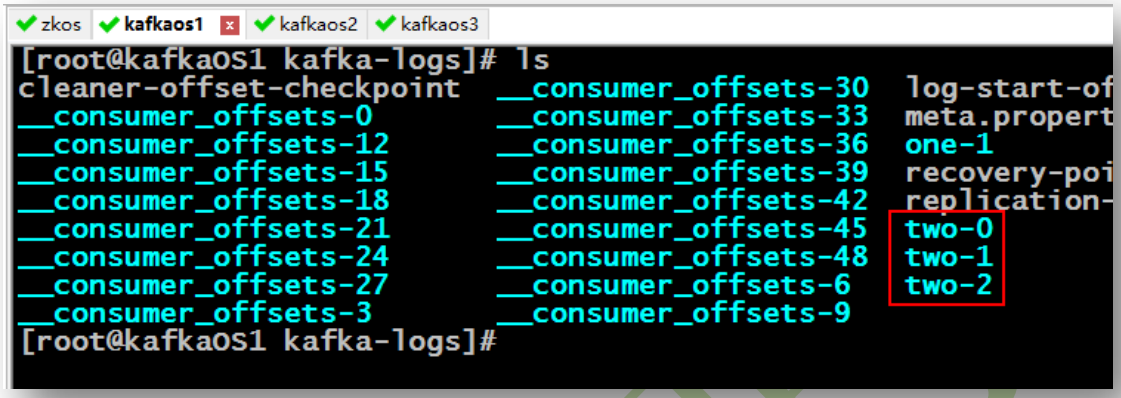
（2） 3 个分区 1 个备份

再次创建一个主题，命名为 one，创建三个分区，但仍为一个备份。 依次查看三台 broker，可以看到每台 broker 中都有一个 one 主题的分区。



（3） 3 个分区 3 个备份

再次创建一个主题，命名为 two，创建三个分区，三个备份。依次查看三台 broker，可以看到每台 broker 中都有三份 two 主题的分区。

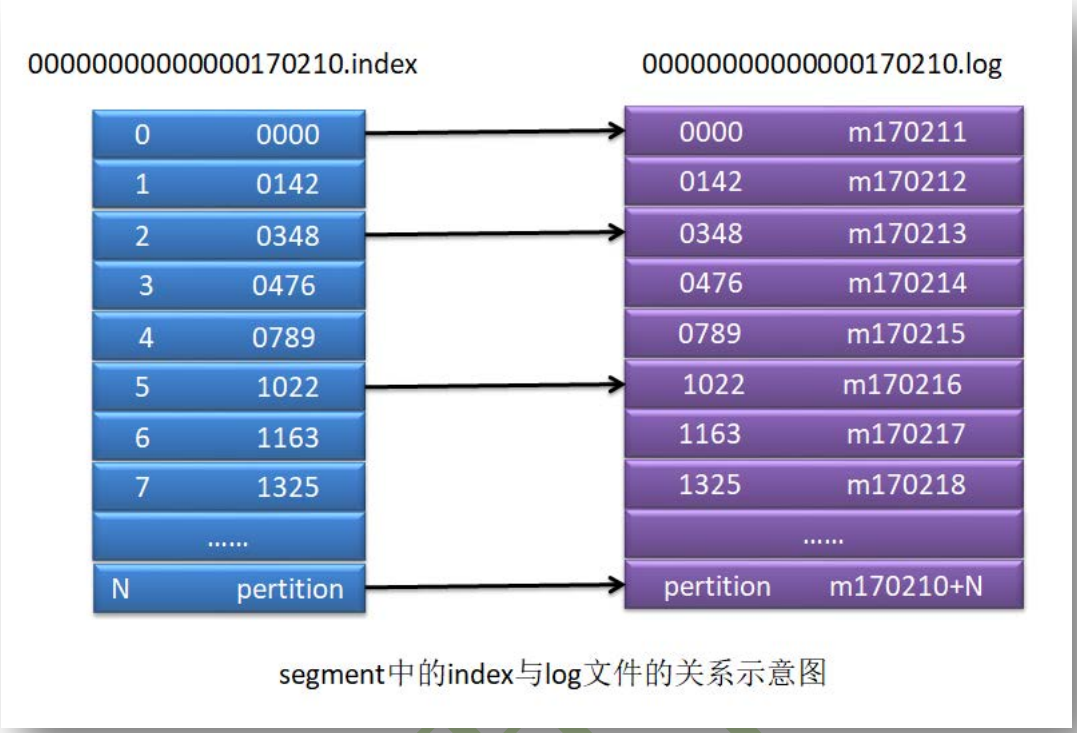


2.4.2 查看段 segment

（1） segment 文件

segment 是一个逻辑概念，其由两类物理文件组成，分别为“.index”文件和“.log”文件。“.log”文件中存放的是消息，而“.index”文件中存放的是“.log”文件中消息的索引。

00000000000000001456.log



segment中的index与log文件的关系示意图

（2） 查看 segment

对于 segment 中的 log 文件，不能直接通过 cat 命令查找其内容，而是需要通过 kafka 自带的一个工具查看。

```
bin/kafka-run-class.sh kafka.tools.DumpLogSegments --files
/tmp/kafka-logs/test-0/00000000000000000000.log  --print-data-log
```

一个用户的一个主题会被提交到一个__consumer_offsets 分区中。使用主题字符串的 hash 值与 50 取模，结果即为分区索引。

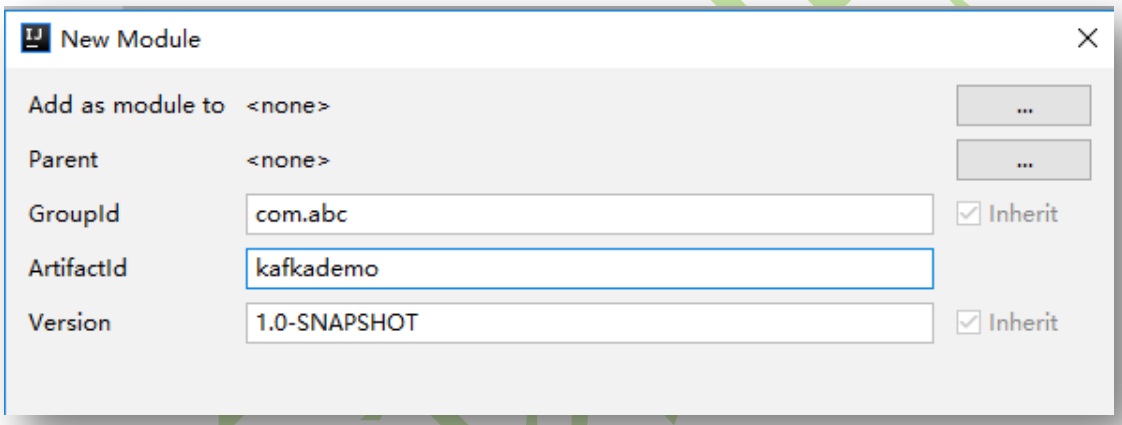
第3章 Kafka API

首先在命令行创建一个名称为 cities 的主题，并创建该主题的订阅者。

3.1 使用 kafka 原生 API

3.1.1 创建工程

创建一个 Maven 的 Java 工程，命名为 kafkaDemo。创建时无需导入依赖。为了简单，后面的发布者与消费者均创建在该工程中。



3.1.2 导入依赖

```
<!-- kafka 依赖 -->
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.12</artifactId>
  <version>1.1.1</version>
</dependency>
```

3.1.3 创建发布者 OneProducer

（1） 创建发布者类 OneProducer

```
public class OneProducer {
    // 第一个泛型为key的类型，第二个泛型为消息本身的类型
    private KafkaProducer<Integer, String> producer;

    public OneProducer() {
        Properties properties = new Properties();
        properties.put("bootstrap.servers",
                       "kafkaOS1:9092,kafkaOS2:9092,kafkaOS3:9092");
        properties.put("key.serializer",
                       "org.apache.kafka.common.serialization.IntegerSerializer");
        properties.put("value.serializer",
                       "org.apache.kafka.common.serialization.StringSerializer");

        this.producer = new KafkaProducer<Integer, String>(properties);
    }
```

```
    public void sendMsg() {
        // 创建记录（消息）
        // 指定主题及消息本身
        // ProducerRecord<Integer, String> record =
        //                         new ProducerRecord<>("cities", "shanghai");
        // 指定主题、key，及消息本身
        // ProducerRecord<Integer, String> record =
        //                         new ProducerRecord<>("cities", 1, "shanghai");
        // 指定主题、要写入的patition、key，及消息本身
        ProducerRecord<Integer, String> record =
                                new ProducerRecord<>("cities", 0, 1, "shanghai");

        // 发布消息，其返回值为Future对象，表示其发送过程为异步，不过这里不使用该返回结果
        // Future<RecordMetadata> future = producer.send(record);
        producer.send(record);
    }
}
```

（2） 创建测试类 OneProducerTest

```
public class OneProducerTest {

    public static void main(String[] args) throws IOException {
        OneProducer producer = new OneProducer();
        producer.sendMsg();
        System.in.read();
    }
}
```

3.1.4 创建发布者 TwoProducer

前面的方式在消息发送成功后，代码中没有任何提示，这里可以使用回调方式，即发送成功后，会触发回调方法的执行。

（1） 创建发布者类 TwoProducer

复制 OneProducer 类，仅修改 sendMsg()方法。

```
// 发布消息，其返回值为Future对象，表示其发送过程为异步，不过这里不使用该返回结果
// Future<RecordMetadata> future = producer.send(record);
// producer.send(record);

// 可以调用以下两个参数的send()方法，可以在消息发布成功后触发回调的执行
producer.send(record, new Callback() {
    // RecordMetadata，消息元数据，即主题、消息的key、消息本身等的封装对象
    @Override
    public void onCompletion(RecordMetadata metadata, Exception exception) {
        System.out.print("partition = " + metadata.partition());
        System.out.print(", topic = " + metadata.topic());
        System.out.println(", offset = " + metadata.offset());
    }
});
```

（2） 创建测试类 TwoProducerTest

```
public class TwoProducerTest {

    public static void main(String[] args) throws IOException {
        TwoProducer producer = new TwoProducer();
        producer.sendMsg();
        System.in.read();
    }
}
```

3.1.5 批量发送消息

（1） 创建发布者类 SomeProducerBatch

复制前面的发布者类，在其基础上进行修改。

```
public class SomeProducerBatch {
    // 第一个泛型为key的类型，第二个泛型为消息本身的类型
    private KafkaProducer<Integer, String> producer;

    public SomeProducerBatch() {
        Properties properties = new Properties();
        properties.put("bootstrap.servers", "kafkaOS1:9092,kafkaOS2:9092
        properties.put("key.serializer", "org.apache.kafka.common.serial
        properties.put("value.serializer", "org.apache.kafka.common.seri
        properties.put("batch.size ", 16384);  // 16K
        properties.put("linger.ms", 50);  // 50ms
        this.producer = new KafkaProducer<Integer, String>(properties);
    }
```

```
public void sendMsg() {
    for (int i=0; i<50; i++) {
        ProducerRecord<Integer, String> record =
                new ProducerRecord<>("cities", 0, i * 10, "city-" + i*100);

        producer.send(record, new Callback() {
            // RecordMetadata，消息元数据，即主题、消息的key、消息本身等的封装对象
            @Override
            public void onCompletion(RecordMetadata metadata, Exception exception) {
                System.out.print("partition = " + metadata.partition());
                System.out.print(", topic = " + metadata.topic());
                System.out.println(", offset = " + metadata.offset());
            }
        });
    }
}
```

（2） 创建测试类 ProducerBatchTest

```
public class ProducerBatchTest {

    public static void main(String[] args) throws IOException {
        SomeProducerBatch producer = new SomeProducerBatch();
        producer.sendMsg();
        System.in.read();
    }
}
```

3.1.6 消费者组

（1） 创建消费者类 SomeConsumer

```
public class SomeConsumer extends ShutdownableThread {
    private KafkaConsumer<Integer, String> consumer;

    public SomeConsumer() {
        super("KafkaConsumerTest", false);

        Properties properties = new Properties();
        String brokers = "kafkaOS1:9092,kafkaOS2:9092,kafkaOS3:9092";
        properties.put("bootstrap.servers", brokers);
        properties.put("group.id", "cityGroup1");
        properties.put("enable.auto.commit", "true");
        properties.put("auto.commit.interval.ms", "1000");
        properties.put("session.timeout.ms", "30000");
        properties.put("heartbeat.interval.ms", "10000");
        properties.put("auto.offset.reset", "earliest");
        properties.put("key.deserializer",
                "org.apache.kafka.common.serialization.IntegerDeserializer");
        properties.put("value.deserializer",
                "org.apache.kafka.common.serialization.StringDeserializer");

        this.consumer = new KafkaConsumer<Integer, String>(properties);
    }
```

```
    @Override
    public void doWork() {
        // 指定要消费的主题
        consumer.subscribe(Collections.singletonList("cities"));
        ConsumerRecords<Integer, String> records = consumer.poll(1000);
        for(ConsumerRecord record : records) {
            System.out.print("topic = " + record.topic());
            System.out.print(" partition = " + record.partition());
            System.out.print(" key = " + record.key());
            System.out.println(" value = " + record.value());
        }
    }
}
```

（2） 创建测试类 ConsumerTest

```
public class ConsumerTest {
    public static void main(String[] args) {
        SomeConsumer consumer = new SomeConsumer();
        consumer.start();
    }
}
```

3.1.7 消费者同步手动提交

（1） 自动提交的问题

前面的消费者都是以自动提交 offset 的方式对 broker 中的消息进行消费的，但自动提交可能会出现消息重复消费的情况。所以在生产环境下，很多时候需要对 offset 进行手动提交，以解决重复消费的问题。

（2） 手动提交分类

手动提交又可以划分为同步提交、异步提交，同异步联合提交。这些提交方式仅仅是doWork()方法不相同，其构造器是相同的。所以下面首先在前面消费者类的基础上进行构造器的修改，然后再分别实现三种不同的提交方式。

（3） 创建消费者类 SyncManualConsumer

A、 原理

同步提交方式是，消费者向 broker 提交 offset 后等待 broker 成功响应。若没有收到响应，则会重新提交，直到获取到响应。而在这个等待过程中，消费者是阻塞的。其严重影响了消费者的吞吐量。

B、 修改构造器

直接复制前面的 SomeConsumer，在其基础上进行修改。

```
public class SyncManualConsumer extends ShutdownableThread {
    private KafkaConsumer<Integer, String> consumer;

    public SyncManualConsumer() {
        super("KafkaConsumerTest", false);

        Properties properties = new Properties();
        String brokers = "kafkaOS1:9092,kafkaOS2:9092,kafkaOS3:9092";
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, brokers);
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "cityGro11");
        properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");
        // properties.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "1000");
        // 设置一次提交的offset个数
        properties.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 10);
        properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
```

C、 修改 doWork()方法

```
@Override
public void doWork() {
    // 指定要消费的主题
    consumer.subscribe(Collections.singletonList("cities"));
    ConsumerRecords<Integer, String> records = consumer.poll(1000);
    for(ConsumerRecord record : records) {
        System.out.print("topic = " + record.topic());
        System.out.print(" partition = " + record.partition());
        System.out.print(" key = " + record.key());
        System.out.println(" value = " + record.value());

        // 手动同步提交
        consumer.commitSync();
    }
}
```

（4） 创建测试类 SyncManulTest

```
public class SyncManualTest {
    public static void main(String[] args) {
        SyncManualConsumer consumer = new SyncManualConsumer();
        consumer.start();
    }
}
```

3.1.8 消费者异步手动提交

（1） 原理

手动同步提交方式需要等待 broker 的成功响应，效率太低，影响消费者的吞吐量。异

步提交方式是，消费者向 broker 提交 offset 后不用等待成功响应，所以其增加了消费者的吞吐量。

（2） 创建消费者类 AsyncManualConsumer

复制前面的 SyncManualConsumer 类，在其基础上进行修改。

```
@Override
public void doWork() {
    // 指定要消费的主题
    consumer.subscribe(Collections.singletonList("cities"));
    ConsumerRecords<Integer, String> records = consumer.poll(1000);
    for(ConsumerRecord record : records) {
        System.out.print("topic = " + record.topic());
        System.out.print(" partition = " + record.partition());
        System.out.print(" key = " + record.key());
        System.out.println(" value = " + record.value());

        // 手动同步提交
        // consumer.commitSync();
        // 手动异步提交
        // consumer.commitAsync();
        // 带回调功能的手动异步提交
        consumer.commitAsync((offsets, e) -> {
            if(e != null) {
                System.out.print("提交失败, offsets = " + offsets);
                System.out.println(", exception = " + e);
            }
        });
    }
}
```

（3） 创建测试类 AsyncManulTest

```
public class AsyncManualTest {
    public static void main(String[] args) {
        AsynManualConsumer consumer = new AsynManualConsumer();
        consumer.start();
    }
}
```

3.1.9 消费者同异步手动提交

（1） 原理

同异步提交，即同步提交与异步提交组合使用。一般情况下，若偶尔出现提交失败，其也不会影响消费者的消费。因为后续提交最终会将这次提交失败的 offset 给提交了。

但异步提交会产生重复消费，为了防止重复消费，可以将同步提交与异常提交联合使用。

（2） 创建消费者类 SyncAsyncManualConsumer

复制前面的 AsyncManualConsumer 类，在其基础上进行修改。

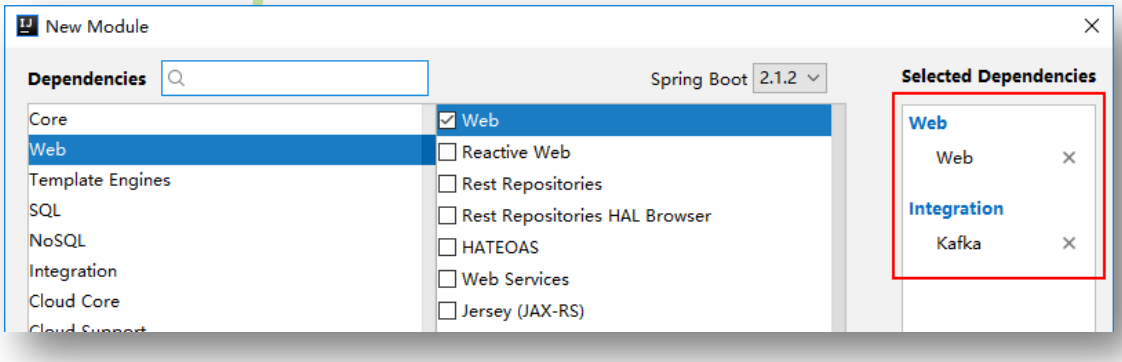
```java
@Override
public void doWork() {
    // 指定要消费的主题
    consumer.subscribe(Collections.singletonList("cities"));
    ConsumerRecords<Integer, String> records = consumer.poll(1000);
    for(ConsumerRecord record : records) {
        System.out.print("topic = " + record.topic());
        System.out.print(" partition = " + record.partition());
        System.out.print(" key = " + record.key());
        System.out.println(" value = " + record.value());

        // 带回调功能的手动异步提交
        consumer.commitAsync((offsets, e) -> {
            if (e != null) {
                System.out.print("提交失败, offsets = " + offsets);
                System.out.println(", exception = " + e);
                // 同步提交
                consumer.commitSync();
            }
        });
    }
}
```

3.2 Spring Boot Kafka

为了简单，以下代码是将消息发布者与订阅者定义到了一个工程中的。

3.2.1 创建工程

创建一个 Spring Boot 工程，导入如下依赖。



3.2.2 定义发布者

Spring 是通过 KafkaTemplate 来完成对 Kafka 的操作的。

（1） 修改配置文件

```yaml
# 自定义属性
kafka:
  topic: cities

# 配置Kafka
spring:
  kafka:
    bootstrap-servers: kafkaOS1:9092,kafkaOS2:9092,kafkaOS3:9092
    producer:   # 配置生产者
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.apache.kafka.common.serialization.StringSerializer
```

（2） 定义发布者处理器

Spring Kafka 通过 KafkaTemplate 完成消息的发布。

```java
@RestController
public class SomeProducer {
    @Autowired
    private KafkaTemplate<String, String> template;

    // 从配置文件读取自定义属性
    @Value("${kafka.topic}")
    private String topic;

    // 由于是提交数据，所以使用Post方式
    @PostMapping("/msg/send")
    public String sendMsg(@RequestParam("message") String message) {
        template.send(topic, message);
        return "send success";
    }
}
```

3.2.3 定义消费者

Spring 是通过监听方式实现消费者的。

（1） 修改配置文件

在配置文件中添加如下内容。注意，Spring 中要求必须为消费者指定组。

application.yml

```yaml
# 自定义属性
kafka:
  topic: cities

# 配置Kafka
spring:
  kafka:
    bootstrap-servers: kafkaOS1:9092,kafkaOS2:9092,kafkaOS3:9092
    producer:   # 配置生产者
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.apache.kafka.common.serialization.StringSerializer

    consumer:   # 配置消费者
      group-id: group0  # 消费者组
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
```

（2） 定义消费者

Spring Kafka 是通过 KafkaListener 监听方式来完成消息订阅与接收的。当监听到有指定主题的消息时，就会触发@KafkaListener 注解所标注的方法的执行。

```java
@Component
public class SomeConsumer {

    @KafkaListener(topics = "${kafka.topic}")
    public void onMsg(String message) {
        System.out.println("Kafka消费者接受到消息 " + message);
    }

}
```