Kafka的优势：

1. 极致的性能：基于Scala和Java语言开发，设计中大量使用了批量处理和异步的思想，最高可以每秒处理千万级别的消息。
2. 生态系统兼容性无可匹敌：Kafka与周边生态系统的兼容性是最好的没有之一，尤其在大数据和流计算方面。

一、初识Kafka

从官方介绍中我们可以得到以下信息：

Kafka是一个分布式流式处理平台。这到底是什么意思呢？

流平台具有三个关键功能：

1. 消息队列：发布和订阅消息流，这个功能类似于消息队列，这也是 Kafka 也被归类为消息队列的原因。
2. 容错的持久方式存储记录消息流： Kafka 会把消息持久化到磁盘，有效避免了消息丢失的风险·。
3. 流式处理平台： 在消息发布的时候进行处理，Kafka 提供了一个完整的流式处理类库。

Kafka 主要有两大应用场景：

1. 消息队列：建立实时流数据管道，以可靠地在系统或应用程序之间获取数据。
2. 数据处理：构建实时的流数据处理程序来转换或处理数据流。

关于 Kafka 几个非常重要的概念：

* Kafka 将记录流（流数据）存储在 topic 中。
* 每个记录由一个键、一个值、一个时间戳组成。

#### 1、Kafka消息模型

早期的JMS和AMQP属于消息服务领域权威组织所做的相关的标准。但是，这些标准的进化跟不上消息队列的演进速度，这些标准实际上已经属于废弃状态。所以，可能存在的情况是：不同的消息队列都有自己的一套消息模型。

队列模型（早期的消息模型）



使用队列（Queue）作为消息通信载体，满足生产者与消费者模式，一条消息只能被一个消费者使用，未被消费的消息在队列中保留直到被消费或超时。比如：我们生产者发送100条消息的话，两个消费者来消费一般情况下两个消费者会按照消息发送的顺序各自消费一半（也就是你一个我一个的消费。）

队列模型存在的问题：

假如我们存在这样一种情况：我们需要将生产者产生的消息分发给多个消费者，并且每个消费者都能接收到完成的消息内容。

这种情况，队列模型就不好解决了。很多比较杠精的人就说：我们可以为每个消费者创建一个单独的队列，让生产者发送多份。这是一种非常愚蠢的做法，浪费资源不说，还违背了使用消息队列的目的。

发布-订阅模型（Kafka消息模型）



发布订阅模型（Pub-Sub）使用主题（Topic）作为消息通信载体，类似于广播模式；发布者发布一条消息，该消息通过主题传递给所有的订阅者，在一条消息广播之后才订阅的用户则是收不到该条消息的。

在发布-订阅模型中，如果只有一个订阅者，那它和队列模型就基本是一样的了。所以说，发布-订阅模型在功能层面上是可以兼容队列模型的。

RocketMQ的消息模型和Kafka基本是完全一样的。唯一的区别是Kafka中没有队列这个概念，与之对应的是Partition（分区）。

#### 2、Kafka重要概念解读

Kafka将生产者发布的消息发送到Topic（主题）中，需要这些消息的消费者可以订阅这些Topic（主题），如下图所示：



* Producer（生产者） : 产生消息的一方。
* Consumer（消费者） : 消费消息的一方。
* Broker（代理） : 可以看作是一个独立的 Kafka 实例。多个 Kafka Broker 组成一个 Kafka Cluster。

同时，你一定也注意到每个 Broker 中又包含了 Topic 以及 Partition 这两个重要的概念：

* Topic（主题） : Producer 将消息发送到特定的主题，Consumer 通过订阅特定的 Topic(主题) 来消费消息。
* Partition（分区） : Partition 属于 Topic 的一部分。一个 Topic 可以有多个 Partition ，并且同一 Topic 下的 Partition 可以分布在不同的 Broker 上，这也就表明一个 Topic 可以横跨多个 Broker 。这正如我上面所画的图一样。

划重点：Kafka 中的 Partition（分区） 实际上可以对应成为消息队列中的队列。这样是不是更好理解一点？

另外，还有一点我觉得比较重要的是 Kafka 为分区（Partition）引入了多副本（Replica）机制。分区（Partition）中的多个副本之间会有一个叫做 leader 的家伙，其他副本称为 follower。我们发送的消息会被发送到 leader 副本，然后 follower 副本才能从 leader 副本中拉取消息进行同步。

生产者和消费者只与 leader 副本交互。你可以理解为其他副本只是 leader 副本的拷贝，它们的存在只是为了保证消息存储的安全性。当 leader 副本发生故障时会从 follower 中选举出一个 leader,但是 follower 中如果有和 leader 同步程度达不到要求的参加不了 leader 的竞选。

Kafka 的多分区（Partition）以及多副本（Replica）机制有什么好处呢？

1. Kafka 通过给特定 Topic 指定多个 Partition, 而各个 Partition 可以分布在不同的 Broker 上, 这样便能提供比较好的并发能力（负载均衡）。
2. Partition 可以指定对应的 Replica 数, 这也极大地提高了消息存储的安全性, 提高了容灾能力，不过也相应的增加了所需要的存储空间。

#### 3、Zookeeper在Kafka中的作用

要想搞懂zookeeper在Kafka中的作用一定要自己搭建一个Kafka环境然后自己进zookeeper去看一下有哪些文件夹和Kafka有关，每个节点又保存了什么信息。

下图就是本地Zookeeper，它成功和本地的Kafka关联上（以下文件夹结构借助idea插件Zookeeper tool实现）。



ZooKeeper 主要为 Kafka 提供元数据的管理的功能。

从图中我们可以看出，Zookeeper 主要为 Kafka 做了下面这些事情：

1. Broker 注册 ：在 Zookeeper 上会有一个专门用来进行 Broker 服务器列表记录的节点。每个 Broker 在启动时，都会到 Zookeeper 上进行注册，即到/brokers/ids 下创建属于自己的节点。每个 Broker 就会将自己的 IP 地址和端口等信息记录到该节点中去
2. Topic 注册 ： 在 Kafka 中，同一个Topic 的消息会被分成多个分区并将其分布在多个 Broker 上，这些分区信息及与 Broker 的对应关系也都是由 Zookeeper 在维护。比如我创建了一个名字为 my-topic 的主题并且它有两个分区，对应到 zookeeper 中会创建这些文件夹：/brokers/topics/my-topic/Partitions/0、/brokers/topics/my-topic/Partitions/1
3. 负载均衡 ：上面也说过了 Kafka 通过给特定 Topic 指定多个 Partition, 而各个 Partition 可以分布在不同的 Broker 上, 这样便能提供比较好的并发能力。 对于同一个 Topic 的不同 Partition，Kafka 会尽力将这些 Partition 分布到不同的 Broker 服务器上。当生产者产生消息后也会尽量投递到不同 Broker 的 Partition 里面。当 Consumer 消费的时候，Zookeeper 可以根据当前的 Partition 数量以及 Consumer 数量来实现动态负载均衡。
4. ......

#### 4、Kafka如何保证消息的消费顺序

我们在使用消息队列的过程中经常有业务场景需要严格保证消息的消费顺序，比如我们同时发了2个消息，这2个消息对应的操作分别对应的数据库操作是：更改用户会员等级、根据会员等级计算订单价格。假如这两条消息的消费顺序不一样造成的最终结果就会截然不同。

我们知道Kafka中Partition(分区)是真正保存消息的地方，我们发送的消息都被放在了这里。而我们的Partition(分区)又存在于Topic(主题)这个概念中，并且我们可以给特定Topic指定多个Partition。



每次添加消息到 Partition(分区) 的时候都会采用尾加法，如上图所示。Kafka 只能为我们保证 Partition(分区) 中的消息有序，而不能保证 Topic(主题) 中的 Partition(分区) 的有序。

消息在被追加到 Partition(分区)的时候都会分配一个特定的偏移量（offset）。Kafka 通过偏移量（offset）来保证消息在分区内的顺序性。

所以，我们就有一种很简单的保证消息消费顺序的方法：1 个 Topic 只对应一个 Partition。这样当然可以解决问题，但是破坏了 Kafka 的设计初衷。

Kafka 中发送 1 条消息的时候，可以指定 topic, partition, key,data（数据） 4 个参数。如果你发送消息的时候指定了 Partition 的话，所有消息都会被发送到指定的 Partition。并且，同一个 key 的消息可以保证只发送到同一个 partition，这个我们可以采用表/对象的 id 来作为 key 。

总结一下，对于如何保证 Kafka 中消息消费的顺序，有了下面两种方法：

1个Topic只对应一个Partition。

（推荐）发送消息的时候指定key/Partition。

当然不仅仅只有上面两种方法。

二、使用Docker安装搭建Kafka环境

使用单机版的Kafka来作为演示：

新建一个名为zk-single-kafka-single.yml的文件，文件内容如下：

version: '2.1'

services:

zoo1:

image: zookeeper:3.4.9

hostname: zoo1

ports:

- "2181:2181"

environment:

ZOO\_MY\_ID: 1

ZOO\_PORT: 2181

ZOO\_SERVERS: server.1=zoo1:2888:3888

volumes:

- ./zk-single-kafka-single/zoo1/data:/data

- ./zk-single-kafka-single/zoo1/datalog:/datalog

kafka1:

image: confluentinc/cp-kafka:5.3.1

hostname: kafka1

ports:

- "9092:9092"

environment:

KAFKA\_ADVERTISED\_LISTENERS: LISTENER\_DOCKER\_INTERNAL://kafka1:19092,LISTENER\_DOCKER\_EXTERNAL://${DOCKER\_HOST\_IP:-127.0.0.1}:9092

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: LISTENER\_DOCKER\_INTERNAL:PLAINTEXT,LISTENER\_DOCKER\_EXTERNAL:PLAINTEXT

KAFKA\_INTER\_BROKER\_LISTENER\_NAME: LISTENER\_DOCKER\_INTERNAL

KAFKA\_ZOOKEEPER\_CONNECT: "zoo1:2181"

KAFKA\_BROKER\_ID: 1

KAFKA\_LOG4J\_LOGGERS: "kafka.controller=INFO,kafka.producer.async.DefaultEventHandler=INFO,state.change.logger=INFO"

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1

volumes:

- ./zk-single-kafka-single/kafka1/data:/var/lib/kafka/data

depends\_on:

- zoo1

运行以下命令即可完成1个节点Zookeeper + 3个节点的Kafka的环境搭建。

docker-compose -f zk-single-kafka-single.yml up

如果需要停止Kafka相关容器的话，运行以下命令：

docker-compose -f zk-single-kafka-multiple.yml down

#### 1、使用命令行测试消息的生产和消费

一般情况下很少用到Kafka的命令行操作。

1. 进入Kafka container内部执行Kafka官方自带了一些命令

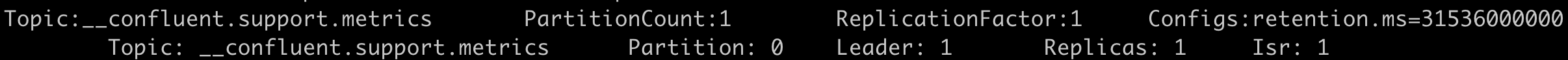
docker exec -ti custom\_file\_kafka1\_1 bash

--tty,-t 命令行交互模式

-interactive,-i 展示容器输入信息STDIN

1. 列出所有Topic

root@kafka1:/# kafka-topics --describe --zookeeper zoo1:2181



1. 创建一个Topic

kafka-topics --create --topic test --partitions 3

--zookeeper zoo1:2181 --replication-factor 1

创建了一个名为test的Topic、，partition数为3，replica数为1

1. 消费者订阅主题

kafka-console-consumer --bootstrap-server localhost:9092 --topic test

订阅了名为test的Topic

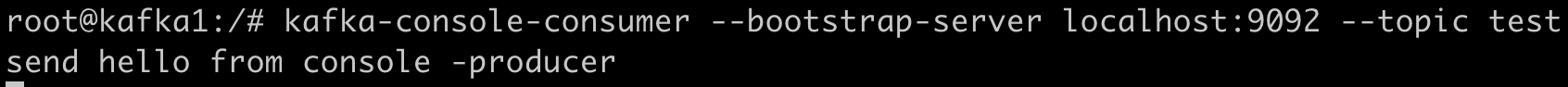
1. 生产者向Topic发送消息

root@kafka1:/# kafka-console-producer --broker-list localhost:9092 --topic test

>send hello from console -producer

使用kafka-console-producer命令向名为test的Topic发送了一条消息，消息内容为“send hello from console -producer”

这个时候，你会发现消费者成功接收到了消息：



三、IDEA相关插件推荐

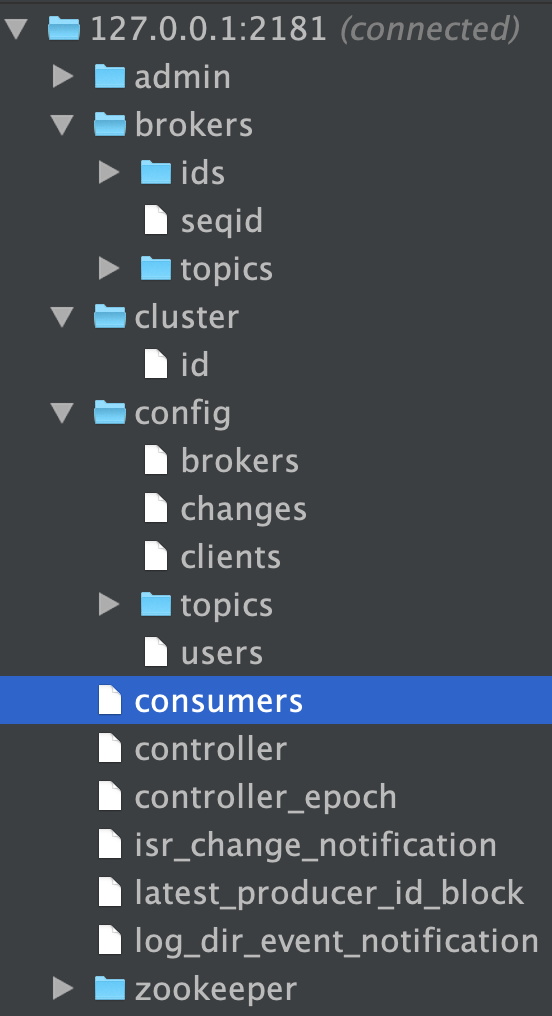
#### 1、Zoolytic-Zookeeper tool

一款Zookeeper可视化工具插件，非常好用！

1. 可视化ZkNodes节点信息
2. ZkNodes节点管理-添加/删除
3. 编辑ZkNodes数据
4. ......

使用方法：

1. 打开工具：View->Tool windows->Zoolytic；
2. 点击“+”号后在弹出框数据：“127.0.0.1:2181”连接zookeeper；
3. 连接之后点击新创建的连接然后点击“+”号旁边的刷新按钮即可。



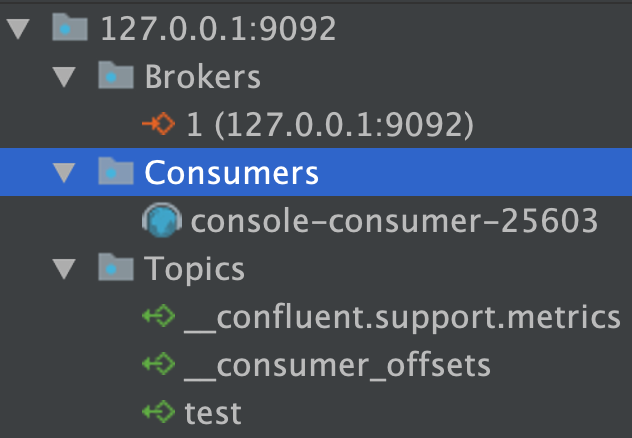
#### 2、Kafkalytic

IDEA提供的Kafka可视化管理插件。提供一下这些功能：

1. 多个集群支持
2. 主题管理：创建/删除/更改分区
3. 使用正则表达式搜索主题
4. 发布字符串/字节序列化的消息
5. 使用不同的策略消费消息

使用方法：

1. 打开工具：View->Tool windows->Kafkalytic；
2. 点击“+”号后在弹出框数据：“127.0.0.1:9092”连接；



四、Java程序中简单使用Kafka

1. 新建一个Maven项目
2. 添加相关依赖

<dependency>

<groupId>org.apache.kafka</groupId>  
 <artifactId>kafka-clients</artifactId>  
 <version>2.2.0</version>  
</dependency>

1. 初始化消费者和生产者

消费者：

public class ConsumerCreator {

*/\*\*  
 \* 该方法用于返回一个KafkaConsumer对象  
 \** ***@return*** *\*/* public static Consumer<String, String> createConsumer() {  
 Properties properties = new Properties();  
 properties.put(ConsumerConfig.*BOOTSTRAP\_SERVERS\_CONFIG*, KafkaConstants.*BROKER\_LIST*);  
 properties.put(ConsumerConfig.*GROUP\_ID\_CONFIG*, KafkaConstants.*GROUP\_ID\_CONFIG*);  
 properties.put(ConsumerConfig.*KEY\_DESERIALIZER\_CLASS\_CONFIG*, StringDeserializer.class.getName());  
 properties.put(ConsumerConfig.*VALUE\_DESERIALIZER\_CLASS\_CONFIG*, StringDeserializer.class.getName());  
 return new KafkaConsumer<>(properties);  
 }  
}

生产者：

public class ProducerCreator {

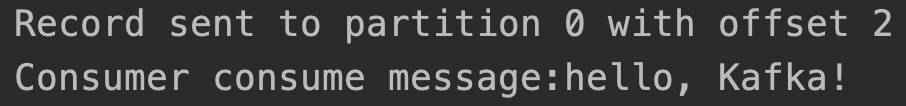
*/\*\*  
 \* 该方法用于返回一个KafkaProducer对象  
 \** ***@return*** *\*/* public static Producer<String, String> createProducer() {  
 Properties properties = new Properties();  
 properties.put(ProducerConfig.*BOOTSTRAP\_SERVERS\_CONFIG*, KafkaConstants.*BROKER\_LIST*);  
 properties.put(ProducerConfig.*CLIENT\_ID\_CONFIG*, KafkaConstants.*CLIENT\_ID*);  
 properties.put(ProducerConfig.*KEY\_SERIALIZER\_CLASS\_CONFIG*, StringSerializer.class.getName());  
 properties.put(ProducerConfig.*VALUE\_SERIALIZER\_CLASS\_CONFIG*, StringSerializer.class.getName());  
 return new KafkaProducer<>(properties);  
 }  
}

1. 发送和消费消息

private static final String *TOPIC* = "test-topic";

public static void main(String[] args) {  
 Producer<String, String> producer = ProducerCreator.*createProducer*();  
 ProducerRecord<String, String> record =  
 new ProducerRecord<>(*TOPIC*, "hello, Kafka!");  
 try {  
 //send message  
 RecordMetadata metadata = producer.send(record).get();  
 System.*out*.println("Record sent to partition " + metadata.partition()  
 + " with offset " + metadata.offset());  
 } catch (ExecutionException | InterruptedException e) {  
 System.*out*.println("Error in sending record");  
 e.printStackTrace();  
 }  
 producer.close();  
  
 Consumer<String, String> consumer = ConsumerCreator.*createConsumer*();  
 // 循环消费消息  
 while (true) {  
 //subscribe topic and consume message  
 consumer.subscribe(Collections.*singletonList*(*TOPIC*));  
  
 ConsumerRecords<String, String> consumerRecords =  
 consumer.poll(Duration.*ofMillis*(1000));  
 for (ConsumerRecord<String, String> consumerRecord : consumerRecords) {  
 System.*out*.println("Consumer consume message:" + consumerRecord.value());  
 }  
 }  
}

测试结果：



五、SpringBoot中使用Kafka

#### 1、创建SpringBoot项目

#### 2、配置Kafka

通过application.yml配置文件配置Kafka基本信息。

server:

port: 9090  
spring:  
 kafka:  
 consumer:  
 bootstrap-servers: localhost:9092  
 *# 配置消费者消息offset是否自动重置（消费者重连会能够接收最开始的消息）* auto-offset-reset: earliest  
 producer:  
 bootstrap-servers: localhost:9092  
 *# 发送的对象信息变为json格式* value-serializer: org.springframework.kafka.support.serializer.JsonSerializer  
  
kafka:  
 topic:  
 my-topic: my-topic  
 my-topic2: my-topic2

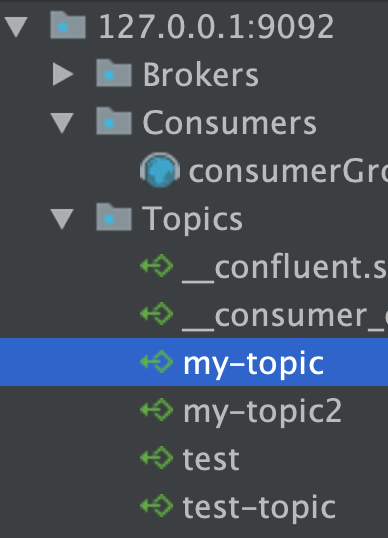
Kafka额外配置类:

@Configuration

public class KafkaConfig {  
  
 @Value("${kafka.topic.my-topic}")  
 private String myTopic;  
 @Value("${kafka.topic.my-topic2}")  
 private String myTopic2;  
  
 */\*\*  
 \* JSON消息转换器  
 \*/* @Bean  
 public RecordMessageConverter jsonConvertor() {  
 return new StringJsonMessageConverter();  
 }  
  
 */\*\*  
 \* 通过注入一个NewTopic类型的Bean来创建topic，如果已存在，则会忽略。  
 \*/* @Bean  
 public NewTopic myTopic() {  
 return new NewTopic(myTopic, 2, (short) 1);  
 }  
  
 @Bean  
 public NewTopic myTopic2() {  
 return new NewTopic(myTopic2, 1, (short) 1);  
 }  
}

当我们到了这一步之后，就可以试着运行项目了，运行成功后会发现SpringBoot为你创建了两个topic：

1. my-topic:partition数为2，replica数为1
2. my-topic2:partition数为1，replica数为2



kafka-topics --describe --zookeeper zoo1:2181命令查看或者直接通过IDEA提高的Kafka可视化管理插件Kafkalytic来查看。

#### 3、创建要发送的消息实体类

@Data

public class Book {  
  
 private Long id;  
 private String name;  
  
 public Book() {  
 }  
  
 public Book(Long id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
}

#### 4、创建发送消息的生产者

@Service

public class BookProducerService {  
 private static final Logger *logger* = LoggerFactory.*getLogger*(BookProducerService.class);  
  
 private final KafkaTemplate<String, Object> kafkaTemplate;  
  
 public BookProducerService(KafkaTemplate<String, Object> kafkaTemplate) {  
 this.kafkaTemplate = kafkaTemplate;  
 }  
  
 public void sendMessage(String topic, Object o) {  
 // 分区编号最好为null，交给kafka自己去分配  
 ProducerRecord<String, Object> producerRecord = new ProducerRecord<>(

topic,

null,

System.*currentTimeMillis*(),

String.*valueOf*(o.hashCode()),

o);  
  
 ListenableFuture<SendResult<String, Object>> future = kafkaTemplate.send(producerRecord);  
 future.addCallback(result -> *logger*.info("生产者成功发送消息到topic:{} partition:{}的消息", result.getRecordMetadata().topic(), result.getRecordMetadata().partition()),  
 ex -> *logger*.error("生产者发送消失败，原因：{}", ex.getMessage()));  
 }  
}

#### 5、创建消费消息的消费者

通过在方法上使用@KafkaListener注解监听消息，当有消息的时候就会通过poll下来消费。

@RestController

@RequestMapping(value = "/book")  
public class BookController {  
  
 @Value("${kafka.topic.my-topic}")  
 String myTopic;  
 @Value("${kafka.topic.my-topic2}")  
 String myTopic2;  
 private final BookProducerService producer;  
 private AtomicLong atomicLong = new AtomicLong();  
  
 BookController(BookProducerService producer) {  
 this.producer = producer;  
 }  
  
 @PostMapping  
 public void sendMessageToKafkaTopic(@RequestParam("name") String name) {  
 this.producer.sendMessage(myTopic, new Book(atomicLong.addAndGet(1), name));  
 this.producer.sendMessage(myTopic2, new Book(atomicLong.addAndGet(1), name));  
 }  
}

#### 6、创建一个RestController

@RestController

@RequestMapping(value = "/book")  
public class BookController {  
  
 @Value("${kafka.topic.my-topic}")  
 String myTopic;  
 @Value("${kafka.topic.my-topic2}")  
 String myTopic2;  
 private final BookProducerService producer;  
 private AtomicLong atomicLong = new AtomicLong();  
  
 BookController(BookProducerService producer) {  
 this.producer = producer;  
 }  
  
 @PostMapping  
 public void sendMessageToKafkaTopic(@RequestParam("name") String name) {  
 this.producer.sendMessage(myTopic, new Book(atomicLong.addAndGet(1), name));  
 this.producer.sendMessage(myTopic2, new Book(atomicLong.addAndGet(1), name));  
 }  
}

#### 7、测试

输入命令：

curl -X POST -F 'name=Java' <http://localhost:9090/book>

控制台打印出的效果：



my-topic有2个partition（分区）当你尝试发送多条消息的时候，你会发现消息会被比较均匀地发送到每个partion中。