序列化

生产者需要用序列化器（Serializer）把对象转换成字节数组才能通过网络发送给 Kafka。而在对侧，消费者需要用反序列化器（Deserializer）把从 Kafka 中收到的字节数组转换成相应的对象。在代码清单3-1中，为了方便，消息的 key 和 value 都使用了字符串，对应程序中的序列化器也使用了客户端自带的 org.apache.kafka.common.serialization.StringSerializer，除了用于 String 类型的序列化器，还有 ByteArray、ByteBuffer、Bytes、Double、Integer、Long 这几种类型，它们都实现了 org.apache.kafka.common.serialization.Serializer 接口，此接口有3个方法：

```
public void configure(Map<String, ?> configs,
boolean isKey)
public byte[] serialize(String topic, T data)
public void close()
```

configure() 方法用来配置当前类，serialize() 方法用来执行序列化操作。而 close() 方法用来关闭当前的序列化器，一般情况下 close() 是一个空方法，如果实现了此方法，则必须确保此方法的幂等性，因为这个方法很可能会被 KafkaProducer 调用多次。

生产者使用的序列化器和消费者使用的反序列化器是需要一一对应的，如果生产者使用了某种序列化器，比如 StringSerializer，而消费者使用了另一种序列化器，比如 IntegerSerializer，那么是无法解析出想要的数据的。本节讨论的都是与生产者相关的，对于与消费者相关的反序列化器的内容请参见第9节。

下面就以 StringSerializer 为例来看看 Serializer 接口中的3个方法的使用方法，StringSerializer 类的具体实现如代码清单4-1所示。

```
//代码清单4-1 StringSerializer的代码实现
```

```
public class StringSerializer implements
Serializer<String> {
    private String encoding = "UTF8";

    @Override
    public void configure(Map<String, ?> configs,
boolean isKey) {
        String propertyName = isKey ?
"key.serializer.encoding" :
                "value.serializer.encoding";
        Object encodingValue =
configs.get(propertyName);
        if (encodingValue == null)
            encodingValue =
configs.get("serializer.encoding");
        if (encodingValue != null &&
encodingValue instanceof String)
            encoding = (String) encodingValue;
    }

    @Override
    public byte[] serialize(String topic, String
data) {
        try {
            if (data == null)
                return null;
            else
                return data.getBytes(encoding);
        } catch (UnsupportedEncodingException e)
{
            throw new
SerializationException("Error when serializing "
+
```

```
                    "string to byte[] due to
unsupported encoding " + encoding);
        }
    }

    @Override
    public void close() {
        // nothing to do
    }
}
```

首先是 configure() 方法，这个方法是在创建 KafkaProducer 实例的时候调用的，主要用来确定编码类型，不过一般客户端对于 key.serializer.encoding、value.serializer. encoding 和 serializer.encoding 这几个参数都不会配置，在 KafkaProducer 的参数集合（ProducerConfig）里也没有这几个参数（它们可以看作用户自定义的参数），所以一般情况下 encoding 的值就为默认的“UTF-8”。serialize() 方法非常直观，就是将 String 类型转为 byte[] 类型。

如果 Kafka 客户端提供的几种序列化器都无法满足应用需求，则可以选择使用如 Avro、JSON、Thrift、ProtoBuf 和 Protostuff 等通用的序列化工具来实现，或者使用自定义类型的序列化器来实现。下面就以一个简单的例子来介绍自定义类型的使用方法。

假设我们要发送的消息都是 Company 对象，这个 Company 的定义很简单，只有名称 name 和地址 address，示例代码参考如下（为了构建方便，示例中使用了 lombok 工具）：

```
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Company {
    private String name;
    private String address;
}
```

下面我们再来看一下 Company 对应的序列化器 CompanySerializer，示例代码如代码清单4-2所示。

```
//代码清单4-2 自定义的序列化器CompanySerializer
public class CompanySerializer implements
Serializer<Company> {
    @Override
    public void configure(Map configs, boolean
isKey) {}

    @Override
    public byte[] serialize(String topic, Company
data) {
        if (data == null) {
            return null;
        }
        byte[] name, address;
        try {
            if (data.getName() != null) {
```

```
                name =
data.getName().getBytes("UTF-8");
            } else {
                name = new byte[0];
            }
            if (data.getAddress() != null) {
                address =
data.getAddress().getBytes("UTF-8");
            } else {
                address = new byte[0];
            }
            ByteBuffer buffer = ByteBuffer.
                    allocate(4+4+name.length +
address.length);
            buffer.putInt(name.length);
            buffer.put(name);
            buffer.putInt(address.length);
            buffer.put(address);
            return buffer.array();
        } catch (UnsupportedEncodingException e)
{
            e.printStackTrace();
        }
        return new byte[0];
    }

    @Override
    public void close() {}
}
```

上面的这段代码的逻辑很简单，configure() 和close() 方法也都为空。与此对应的反序列化器 CompanyDeserializer 的详细实现参见第9节。

如何使用自定义的序列化器 CompanySerializer 呢？只需将 KafkaProducer 的 value.serializer 参数设置为 CompanySerializer 类的全限定名即可。假如我们要发送一个 Company 对象到 Kafka，关键代码如代码清单4-3所示。

```
//代码清单4-3 自定义序列化器使用示例
Properties properties = new Properties();
properties.put(ProducerConfig.KEY_SERIALIZER_CLAS
S_CONFIG,
        StringSerializer.class.getName());
properties.put(ProducerConfig.VALUE_SERIALIZER_CL
ASS_CONFIG,
        CompanySerializer.class.getName());
properties.put("bootstrap.servers", brokerList);

KafkaProducer<String, Company> producer =
        new KafkaProducer<>(properties);
Company company =
Company.builder().name("hiddenkafka")
        .address("China").build();
ProducerRecord<String, Company> record =
        new ProducerRecord<>(topic, company);
producer.send(record).get();
```

注意，示例中消息的 key 对应的序列化器还是 StringSerializer，这个并没有改动。其实 key.serializer 和 value.serializer 并没有太大的区别，读者可以自行修改 key 对应的序列化器，看看会不会有不一样的效果。

分区器

消息在通过 send() 方法发往 broker 的过程中，有可能需要经过拦截器（Interceptor）、序列化器（Serializer）和分区器（Partitioner）的一系列作用之后才能被真正地发往 broker。拦截器（下一章会详细介绍）一般不是必需的，而序列化器是必需的。消息经过序列化之后就需要确定它发往的分区，如果消息 ProducerRecord 中指定了 partition 字段，那么就不需要分区器的作用，因为 partition 代表的就是所要发往的分区号。

如果消息 ProducerRecord 中没有指定 partition 字段，那么就需要依赖分区器，根据 key 这个字段来计算 partition 的值。分区器的作用就是为消息分配分区。

Kafka 中提供的默认分区器是 org.apache.kafka.clients.producer.internals.DefaultPartitioner，它实现了 org.apache.kafka.clients.producer.Partitioner 接口，这个接口中定义了2个方法，具体如下所示。

```
public int partition(String topic, Object key,
byte[] keyBytes,
                     Object value, byte[]
valueBytes, Cluster cluster);
public void close();
```

其中 partition() 方法用来计算分区号，返回值为 int 类型。partition() 方法中的参数分别表示主题、键、序列化后的键、值、序列化后的值，以及集群的元数据信息，通过这些信息可以实现功能丰富的分区器。close() 方法在关闭分区器的时候用来回收一些资源。

Partitioner 接口还有一个父接口 org.apache.kafka.common.Configurable，这个接口中只有一个方法：

```
void configure(Map<String, ?> configs);
```

Configurable 接口中的 configure() 方法主要用来获取配置信息及初始化数据。

在默认分区器 DefaultPartitioner 的实现中，close() 是空方法，而在 partition() 方法中定义了主要的分区分配逻辑。如果 key 不为 null，那么默认的分区器会对 key 进行哈希（采用 MurmurHash2 算法，具备高运算性能及低碰撞率），最终根据得到的哈希值来计算分区号，拥有相同 key 的消息会被写入同一个分区。如果 key 为 null，那么消息将会以轮询的方式发往主题内的各个可用分区。

> 注意：如果 key 不为 null，那么计算得到的分区号会是所有分区中的任意一个；如果 key 为 null 并且有可用分区时，那么计算得到的分区号仅为可用分区中的任意一个，注意两者之间的差别。

在不改变主题分区数量的情况下，key 与分区之间的映射可以保持不变。不过，一旦主题中增加了分区，那么就难以保证 key 与分区之间的映射关系了。

除了使用 Kafka 提供的默认分区器进行分区分配，还可以使用自定义的分区器，只需同 DefaultPartitioner 一样实现 Partitioner 接口即可。默认的分区器在 key 为 null 时不会选择非可用的分区，我们可以通过自定义的分区器 DemoPartitioner 来打破这一限制，具体的实现可以参考下面的示例代码，如代码清单4-4所示。

```
//代码清单4-4 自定义分区器实现
public class DemoPartitioner implements
Partitioner {
    private final AtomicInteger counter = new
AtomicInteger(0);

    @Override
    public int partition(String topic, Object
key, byte[] keyBytes,
                         Object value, byte[]
valueBytes, Cluster cluster) {
        List<PartitionInfo> partitions =
cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();
        if (null == keyBytes) {
            return counter.getAndIncrement() %
numPartitions;
        }else
            return
Utils.toPositive(Utils.murmur2(keyBytes)) %
numPartitions;
    }

    @Override public void close() {}

    @Override public void configure(Map<String, ?
> configs) {}
}
```

实现自定义的 DemoPartitioner 类之后，需要通过配置参数 partitioner.class 来显式指定这个分区器。示例如下：

```
props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG
,
        DemoPartitioner.class.getName());
```

这个自定义分区器的实现比较简单，读者也可以根据自身业务的需求来灵活实现分配分区的计算方式，比如一般大型电商都有多个仓库，可以将仓库的名称或 ID 作为 key 来灵活地记录商品信息。

生产者拦截器

拦截器（Interceptor）是早在 Kafka 0.10.0.0 中就已经引入的一个功能，Kafka 一共有两种拦截器：生产者拦截器和消费者拦截器。本节主要讲述生产者拦截器的相关内容，有关消费者拦截器的具体细节请参考13节。

生产者拦截器既可以用来在消息发送前做一些准备工作，比如按照某个规则过滤不符合要求的消息、修改消息的内容等，也可以用来在发送回调逻辑前做一些定制化的需求，比如统计类工作。

生产者拦截器的使用也很方便，主要是自定义实现 org.apache.kafka.clients.producer. ProducerInterceptor 接口。ProducerInterceptor 接口中包含3个方法：

```
public ProducerRecord<K, V>
onSend(ProducerRecord<K, V> record);
public void onAcknowledgement(RecordMetadata
metadata, Exception exception);
public void close();
```

KafkaProducer 在将消息序列化和计算分区之前会调用生产者拦截器的 onSend() 方法来对消息进行相应的定制化操作。一般来说最好不要修改消息 ProducerRecord 的 topic、key 和 partition 等信

息，如果要修改，则需确保对其有准确的判断，否则会与预想的效果出现偏差。比如修改 key 不仅会影响分区的计算，同样会影响 broker 端日志压缩（Log Compaction）的功能。

KafkaProducer 会在消息被应答（Acknowledgement）之前或消息发送失败时调用生产者拦截器的 onAcknowledgement() 方法，优先于用户设定的 Callback 之前执行。这个方法运行在 Producer 的I/O线程中，所以这个方法中实现的代码逻辑越简单越好，否则会影响消息的发送速度。

close() 方法主要用于在关闭拦截器时执行一些资源的清理工作。在这3个方法中抛出的异常都会被捕获并记录到日志中，但并不会再向上传递。

ProducerInterceptor 接口与 Partitioner 接口一样，它也有一个同样的父接口 Configurable，具体的内容可以参见 Partitioner 接口的相关介绍。

下面通过一个示例来演示生产者拦截器的具体用法，ProducerInterceptorPrefix 中通过 onSend() 方法来为每条消息添加一个前缀“prefix1-”，并且通过 onAcknowledgement() 方法来计算发送消息的成功率。ProducerInterceptorPrefix 类的具体实现如代码清单4-5所示。

```
//代码清单4-5生产者拦截器示例
public class ProducerInterceptorPrefix implements
        ProducerInterceptor<String,String>{
    private volatile long sendSuccess = 0;
    private volatile long sendFailure = 0;

    @Override
    public ProducerRecord<String, String> onSend(
            ProducerRecord<String, String>
record) {
```

```
        String modifiedValue = "prefix1-" +
record.value();
        return new ProducerRecord<>
(record.topic(),
                record.partition(),
record.timestamp(),
                record.key(), modifiedValue,
record.headers());
    }

    @Override
    public void onAcknowledgement(
            RecordMetadata recordMetadata,
            Exception e) {
        if (e == null) {
            sendSuccess++;
        } else {
            sendFailure ++;
        }
    }

    @Override
    public void close() {
        double successRatio = (double)sendSuccess
/ (sendFailure + sendSuccess);
        System.out.println("[INFO] 发送成功率="
                + String.format("%f",
successRatio * 100) + "%");
    }

    @Override
    public void configure(Map<String, ?> map) {}
}
```

实现自定义的 ProducerInterceptorPrefix 之后，需要在 KafkaProducer 的配置参数 interceptor.classes 中指定这个拦截器，此参数的默认值为“”。示例如下：

```
properties.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,

ProducerInterceptorPrefix.class.getName());
```

然后使用指定了 ProducerInterceptorPrefix 的生产者连续发送10条内容为“kafka”的消息，在发送完之后客户端打印出如下信息：

```
[INFO] 发送成功率=100.000000%
```

如果消费这10条消息，会发现消费了的消息都变成了“prefix1-kafka”，而不是原来的“kafka”。

KafkaProducer 中不仅可以指定一个拦截器，还可以指定多个拦截器以形成拦截链。拦截链会按照 interceptor.classes 参数配置的拦截器的顺序来一一执行（配置的时候，各个拦截器之间使用逗号隔开）。下面我们再添加一个自定义拦截器 ProducerInterceptorPrefixPlus，它只实现了 Interceptor 接口中的 onSend() 方法，主要用来为每条消息添加另一个前缀“prefix2-”，具体实现如下：

```
public ProducerRecord<String, String> onSend(
        ProducerRecord<String, String> record) {
    String modifiedValue = "prefix2-
"+record.value() ;
    return new ProducerRecord<>(record.topic(),
            record.partition(),
record.timestamp(),
            record.key(), modifiedValue,
record.headers());
}
```

接着修改生产者的 interceptor.classes 配置，具体实现如下：

```
properties.put(ProducerConfig.INTERCEPTOR_CLASSES
_CONFIG,
        ProducerInterceptorPrefix.class.getName()
+ ","
                +
ProducerInterceptorPrefixPlus.class.getName());
```

此时生产者再连续发送10条内容为“kafka”的消息，那么最终消费者消费到的是10条内容为“prefix2-prefix1-kafka”的消息。如果将 interceptor.classes 配置中的两个拦截器的位置互换：

```
properties.put(ProducerConfig.INTERCEPTOR_CLASSES
_CONFIG,

ProducerInterceptorPrefixPlus.class.getName() +
","
                +
ProducerInterceptorPrefix.class.getName());
```

那么最终消费者消费到的消息为“prefix1-prefix2-kafka”。

如果拦截链中的某个拦截器的执行需要依赖于前一个拦截器的输出，那么就有可能产生“副作用”。设想一下，如果前一个拦截器由于异常而执行失败，那么这个拦截器也就跟着无法继续执行。在拦截链中，如果某个拦截器执行失败，那么下一个拦截器会接着从上一个执行成功的拦截器继续执行。