反序列化

在第4节中我们讲述了 KafkaProducer 对应的序列化器，那么与此对应的 KafkaConsumer 就会有反序列化器。Kafka 所提供的反序列化器有 ByteBufferDeserializer、ByteArrayDeserializer、BytesDeserializer、DoubleDeserializer、FloatDeserializer、IntegerDeserializer、LongDeserializer、ShortDeserializer、StringDeserializer，它们分别用于 ByteBuffer、ByteArray、Bytes、Double、Float、Integer、Long、Short 及 String 类型的反序列化，这些序列化器也都实现了 Deserializer 接口，与 KafkaProducer 中提及的 Serializer 接口一样，Deserializer 接口也有三个方法。

- public void configure(Map<String, ?> configs, boolean isKey)：用来配置当前类。
- public byte[] deserialize(String topic, byte[] data)：用来执行反序列化。如果 data 为 null，那么处理的时候直接返回 null 而不是抛出一个异常。
- public void close()：用来关闭当前序列化器。

代码清单4-1中描述的是 Kafka 客户端自带的序列化器 StringSerializer 的具体实现，对应的反序列化器 StringDeserializer 的具体代码实现如下：

```
public class StringDeserializer implements
Deserializer<String> {
    private String encoding = "UTF8";

    @Override
    public void configure(Map<String, ?> configs,
boolean isKey) {

```

```
        String propertyName = isKey ?
"key.deserializer.encoding" :
                "value.deserializer.encoding";
        Object encodingValue =
configs.get(propertyName);
        if (encodingValue == null)
            encodingValue =
configs.get("deserializer.encoding");
        if (encodingValue != null &&
encodingValue instanceof String)
            encoding = (String) encodingValue;
    }

    @Override
    public String deserialize(String topic,
byte[] data) {
        try {
            if (data == null)
                return null;
            else
                return new String(data,
encoding);
        } catch (UnsupportedEncodingException e)
{
            throw new
SerializationException("Error when " +
                    "deserializing byte[] to
string due to " +
                    "unsupported encoding " +
encoding);
        }
    }
```

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

configure() 方法中也有3个参数：key.deserializer.encoding、value.deserializer. encoding 和 deserializer.encoding，用来配置反序列化的编码类型，这3个都是用户自定义的参数类型，在 KafkaConsumer 的参数集合（ConsumerConfig）中并没有它们的身影。一般情况下，也不需要配置这几个参数，如果配置了，则需要和 StringSerializer 中配置的一致。默认情况下，编码类型为“UTF-8”。上面示例代码中的 deserialize() 方法非常直观，就是把 byte[] 类型转换为 String 类型。

在代码清单4-2和代码清单4-3中，我们演示了如何通过自定义的序列化器来序列化自定义的 Company 类，这里我们再来看一看与 CompanySerializer 对应的 CompanyDeserializer 的具体实现：

```
public class CompanyDeserializer implements
Deserializer<Company> {
    public void configure(Map<String, ?> configs,
boolean isKey) {}

    public Company deserialize(String topic,
byte[] data) {
        if (data == null) {
            return null;
        }
        if (data.length < 8) {
            throw new
SerializationException("Size of data received " +
                    "by DemoDeserializer is
```

```
shorter than expected!");
        }
        ByteBuffer buffer =
ByteBuffer.wrap(data);
        int nameLen, addressLen;
        String name, address;

        nameLen = buffer.getInt();
        byte[] nameBytes = new byte[nameLen];
        buffer.get(nameBytes);
        addressLen = buffer.getInt();
        byte[] addressBytes = new
byte[addressLen];
        buffer.get(addressBytes);

        try {
            name = new String(nameBytes, "UTF-
8");
            address = new String(addressBytes,
"UTF-8");
        } catch (UnsupportedEncodingException e)
{
            throw new
SerializationException("Error occur when
deserializing!");
        }

        return new Company(name,address);
    }

    public void close() {}
}
```

configure() 方法和 close() 方法都是空实现，而 deserializer() 方法就是将字节数组转换成对应 Company 对象。在使用自定义的反序列化器的时候只需要将相应的 value.deserializer 参数配置为 CompanyDeserializer 即可，示例如下：

```
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        CompanyDeserializer.class.getName());
```

注意如无特殊需要，笔者还是不建议使用自定义的序列化器或反序列化器，因为这样会增加生产者与消费者之间的耦合度，在系统升级换代的时候很容易出错。自定义的类型有一个不得不面对的问题就是 KafkaProducer 和 KafkaConsumer 之间的序列化和反序列化的兼容性。对于 StringSerializer 来说，KafkaConsumer 可以顺其自然地采用 StringDeserializer，不过对于 Company 这种专用类型而言，某个上游应用采用 CompanySerializer 进行序列化之后，下游应用也必须实现对应的 CompanyDeserializer。再者，如果上游的 Company 类型改变，那么下游也需要跟着重新实现一个新的 CompanyDeserializer，后面所面临的难题可想而知。

在实际应用中，在 Kafka 提供的序列化器和反序列化器满足不了应用需求的前提下，推荐使用 Avro、JSON、Thrift、ProtoBuf 或 Protostuff 等通用的序列化工具来包装，以求尽可能实现得更加通用且前后兼容。使用通用的序列化工具也需要实现 Serializer 和 Deserializer 接口，因为 Kafka 客户端的序列化和反序列化入口必须是这两个类型。

本节的最后我们来看一下如何使用通用的序列化工具实现自定义的序列化器和反序列化器的封装。这里挑选了 Protostuff 来做演示，使用的 Protostuff 的 Maven 依赖如下：

```
<dependency>
    <groupId>io.protostuff</groupId>
    <artifactId>protostuff-core</artifactId>
    <version>1.5.4</version>
</dependency>

<dependency>
    <groupId>io.protostuff</groupId>
    <artifactId>protostuff-runtime</artifactId>
    <version>1.5.4</version>
</dependency>
```

为了简化说明，这里只展示出序列化器的 serialize() 方法和 deserialize() 方法，如下所示。

```
    //序列化器ProtostuffSerializer中的serialize()方法
    public byte[] serialize(String topic, Company data) {
        if (data == null) {
            return null;
        }
        Schema schema = (Schema) RuntimeSchema.getSchema(data.getClass());
        LinkedBuffer buffer =
                LinkedBuffer.allocate(LinkedBuffer.DEFAULT_BUFFER_SIZE);
        byte[] protostuff = null;
        try {
            protostuff = ProtostuffIOUtil.toByteArray(data, schema, buffer);
```

```
        } catch (Exception e) {
            throw new
IllegalStateException(e.getMessage(), e);
        } finally {
            buffer.clear();
        }
        return protostuff;
    }
    //反序列化器ProtostuffDeserializer中的
deserialize()方法
    public Company deserialize(String topic,
byte[] data) {
        if (data == null) {
            return null;
        }
        Schema schema =
RuntimeSchema.getSchema(Company.class);
        Company ans = new Company();
        ProtostuffIOUtil.mergeFrom(data, ans,
schema);
        return ans;
    }
```

接下来要做的工作就和 CompanyDeserializer 一样，这里就不一一赘述了。读者可以添加或减少 Company 类中的属性，以此查看采用通用序列化工具的前后兼容性的效能。