你真的了解Flink Kafka source吗?

Flink 提供了专门的 Kafka 连接器，向 Kafka topic 中读取或者写入数据。Flink Kafka Consumer 集成了 Flink 的 Checkpoint 机制，可提供 exactly-once 的处理语义。为此，Flink 并不完全依赖于跟踪 Kafka 消费组的偏移量，而是在内部跟踪和检查偏移量。

引言

当我们在使用Spark Streaming、Flink等计算框架进行数据实时处理时，使用Kafka作为一款发布与订阅的消息系统成为了标配。Spark Streaming与Flink都提供了相对应的Kafka Consumer，使用起来非常的方便，只需要设置一下Kafka的参数，然后添加kafka的source就万事大吉了。如果你真的觉得事情就是如此的so easy，感觉妈妈再也不用担心你的学习了，那就真的是too young too simple sometimes naive了。本文以Flink 的Kafka Source为讨论对象，首先从基本的使用入手，然后深入源码逐一剖析，一并为你拨开Flink Kafka connector的神秘面纱。值得注意的是，本文假定读者具备了Kafka的相关知识，关于Kafka的相关细节问题，不在本文的讨论范围之内。

Flink Kafka Consumer介绍

Flink Kafka Connector有很多个版本，可以根据你的kafka和Flink的版本选择相应的包（maven artifact id）和类名。本文所涉及的Flink版本为1.10，Kafka的版本为2.3.4。Flink所提供的Maven依赖于类名如下表所示：

Maven 依赖	自从哪个版本开始支持	类名	Kafka 版本	注意
flink-connector-kafka-0.8_2.11	1.0.0	FlinkKafkaConsumer08 FlinkKafkaProducer08	0.8.x	这个连接器在内部使用 Kafka 的 SimpleConsumer A
flink-connector-kafka-0.9_2.11	1.0.0	FlinkKafkaConsumer09 FlinkKafkaProducer09	0.9.x	这个连接器使用新的 Kafka Consumer API
flink-connector-	1.2.0	FlinkKafkaConsumer010	0.10.x	这个连接器支持 带有时间戳的 Kafka 消息，用于生产

kafka-0.10_2.11		FlinkKafkaProducer010		
flink-connector-kafka-0.11_2.11	1.4.0	FlinkKafkaConsumer011 FlinkKafkaProducer011	>= 0.11.x	Kafka 从 0.11.x 版本开始不支持 Scala 2.10。此连接 生产者提供 Exactly once 语义。
flink-connector-kafka_2.11	1.7.0	FlinkKafkaConsumer FlinkKafkaProducer	>= 1.0.0	这个通用的 Kafka 连接器尽力与 Kafka client 的最新 client 版本可能会在 Flink 版本之间发生变化。从 Fli client。当前 Kafka 客户端向后兼容 0.10.0 或更高版 0.11.x 和 0.10.x 版本，我们建议你分别使用专用的 fli connector-kafka-0.10_2.11 连接器。

Demo示例

添加Maven依赖

```
<!--本文使用的是通用型的connector-->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka_2.11</artifactId>
  <version>1.10.0</version>
</dependency>
```

简单代码案例

```
public class KafkaConnector {

    public static void main(String[] args) throws Exception {

        StreamExecutionEnvironment senv = StreamExecutionEnvironment.get
ExecutionEnvironment();
        // 开启checkpoint，时间间隔为毫秒
        senv.enableCheckpointing(5000L);
        // 选择状态后端
        senv.setStateBackend((StateBackend) new FsStateBackend("file:///E
://checkpoint"));
        //senv.setStateBackend((StateBackend) new FsStateBackend("hdfs:/
/kms-1:8020/checkpoint"));
        Properties props = new Properties();
        // kafka broker地址
        props.put("bootstrap.servers", "kms-2:9092,kms-3:9092,kms-4:9092
");
        // 仅kafka0.8版本需要配置
        props.put("zookeeper.connect", "kms-2:2181,kms-3:2181,kms-4:2181

```

```
");
        // 消费者组
        props.put("group.id", "test");
        // 自动偏移量提交
        props.put("enable.auto.commit", true);
        // 偏移量提交的时间间隔，毫秒
        props.put("auto.commit.interval.ms", 5000);
        // kafka 消息的key序列化器
        props.put("key.deserializer", "org.apache.kafka.common.serializa
tion.StringDeserializer");
        // kafka 消息的value序列化器
        props.put("value.deserializer", "org.apache.kafka.common.seriali
zation.StringDeserializer");
        // 指定kafka的消费者从哪里开始消费数据
        // 共有三种方式，
        // #earliest
        // 当各分区下有已提交的offset时，从提交的offset开始消费；
        // 无提交的offset时，从头开始消费
        // #latest
        // 当各分区下有已提交的offset时，从提交的offset开始消费；
        // 无提交的offset时，消费新产生的该分区下的数据
        // #none
        // topic各分区都存在已提交的offset时，
        // 从offset后开始消费；
        // 只要有一个分区不存在已提交的offset，则抛出异常
        props.put("auto.offset.reset", "latest");
        FlinkKafkaConsumer<String> consumer = new FlinkKafkaConsumer<>(
                "qfbap_ods.code_city",
                new SimpleStringSchema(),
                props);
        //设置checkpoint后在提交offset，即oncheckpoint模式
        // 该值默认为true，
        consumer.setCommitOffsetsOnCheckpoints(true);

        // 最早的数据开始消费
        // 该模式下，Kafka 中的 committed offset 将被忽略，不会用作起始位置。
        //consumer.setStartFromEarliest();

        // 消费者组最近一次提交的偏移量，默认。
        // 如果找不到分区的偏移量，那么将会使用配置中的 auto.offset.reset 设置
        //consumer.setStartFromGroupOffsets();

        // 最新的数据开始消费
        // 该模式下，Kafka 中的 committed offset 将被忽略，不会用作起始位置。
        //consumer.setStartFromLatest();

        // 指定具体的偏移量时间戳,毫秒
        // 对于每个分区，其时间戳大于或等于指定时间戳的记录将用作起始位置。
        // 如果一个分区的最新记录早于指定的时间戳，则只从最新记录读取该分区数据。
```

```
        // 在这种模式下，Kafka 中的已提交 offset 将被忽略，不会用作起始位置。
        //consumer.setStartFromTimestamp(1585047859000L);

        // 为每个分区指定偏移量
        /*Map<KafkaTopicPartition, Long> specificStartOffsets = new HashMap<>();
        specificStartOffsets.put(new KafkaTopicPartition("qfbap_ods.code_city", 0), 23L);
        specificStartOffsets.put(new KafkaTopicPartition("qfbap_ods.code_city", 1), 31L);
        specificStartOffsets.put(new KafkaTopicPartition("qfbap_ods.code_city", 2), 43L);
        consumer1.setStartFromSpecificOffsets(specificStartOffsets);*/
        /**
         *
         * 请注意：当 Job 从故障中自动恢复或使用 savepoint 手动恢复时，
         * 这些起始位置配置方法不会影响消费的起始位置。
         * 在恢复时，每个 Kafka 分区的起始位置由存储在 savepoint 或 checkpoint 中的 offset 确定
         *
         */

        DataStreamSource<String> source = senv.addSource(consumer);
        // TODO
        source.print();
        senv.execute("test kafka connector");
    }
}
```

参数配置解读

在Demo示例中，给出了详细的配置信息，下面将对上面的参数配置进行逐一分析。

kakfa的properties参数配置

- bootstrap.servers：kafka broker地址
- zookeeper.connect：仅kafka0.8版本需要配置
- group.id：消费者组
- enable.auto.commit：

  自动偏移量提交，该值的配置不是最终的偏移量提交模式，需要考虑用户是否开启了checkpoint，

  在下面的源码分析中会进行解读

- auto.commit.interval.ms：偏移量提交的时间间隔，毫秒
- key.deserializer：

  kafka 消息的key序列化器，如果不指定会使用ByteArrayDeserializer序列化器
- value.deserializer：

kafka 消息的value序列化器，如果不指定会使用ByteArrayDeserializer序列化器

- auto.offset.reset：

  指定kafka的消费者从哪里开始消费数据，共有三种方式，

  - 第一种：earliest
    当各分区下有已提交的offset时，从提交的offset开始消费； 无提交的offset时，从头开始消费
  - 第二种：latest
    当各分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，消费新产生的该分区下的数据
  - 第三种：none
    topic各分区都存在已提交的offset时，从offset后开始消费；只要有一个分区不存在已提交的offset，则抛出异常

  注意：上面的指定消费模式并不是最终的消费模式，取决于用户在Flink程序中配置的消费模式

Flink程序用户配置的参数

- consumer.setCommitOffsetsOnCheckpoints(true)

解释：设置checkpoint后在提交offset，即oncheckpoint模式，该值默认为true，该参数会影响偏移量的提交方式，下面的源码中会进行分析

- consumer.setStartFromEarliest()

  解释： 最早的数据开始消费 ，该模式下，Kafka 中的 committed offset 将被忽略，不会用作起始位置。该方法为继承父类FlinkKafkaConsumerBase的方法。
- consumer.setStartFromGroupOffsets()

  解释：消费者组最近一次提交的偏移量，默认。 如果找不到分区的偏移量，那么将会使用配置中的 auto.offset.reset 设置，该方法为继承父类FlinkKafkaConsumerBase的方法。

- consumer.setStartFromLatest()

解释：最新的数据开始消费，该模式下，Kafka 中的 committed offset 将被忽略，不会用作起始位置。该方法为继承父类FlinkKafkaConsumerBase的方法。

- consumer.setStartFromTimestamp(1585047859000L)

解释：指定具体的偏移量时间戳,毫秒。对于每个分区，其时间戳大于或等于指定时间戳的记录将用作起始位置。 如果一个分区的最新记录早于指定的时间戳，则只从最新记录读取该分区数据。在这种模式下，Kafka 中的已提交 offset 将被忽略，不会用作起始位置。

- consumer.setStartFromSpecificOffsets(specificStartOffsets)

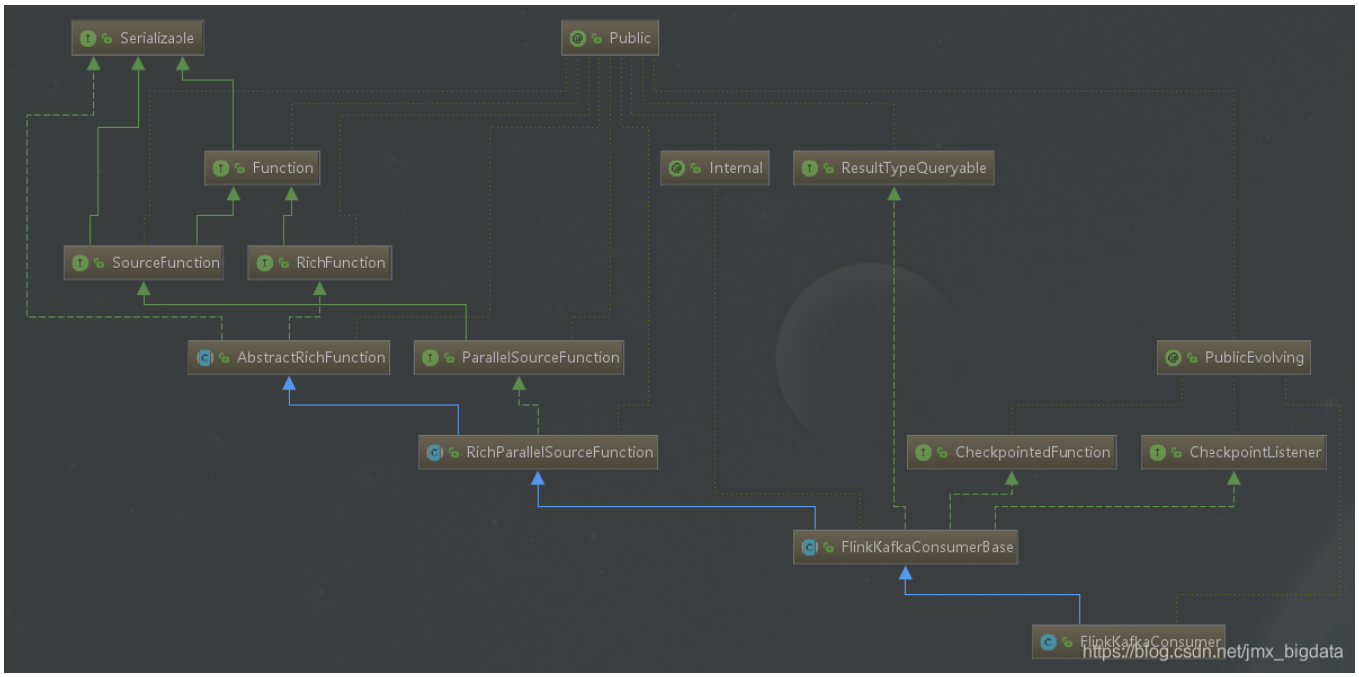
解释：为每个分区指定偏移量，该方法为继承父类FlinkKafkaConsumerBase的方法。

请注意：当 Job 从故障中自动恢复或使用 savepoint 手动恢复时，这些起始位置配置方法不会影响消费的起始位置。在恢复时，每个 Kafka 分区的起始位置由存储在 savepoint 或 checkpoint 中的 offset 确定。

Flink Kafka Consumer源码解读

继承关系

Flink Kafka Consumer继承了FlinkKafkaConsumerBase抽象类，而FlinkKafkaConsumerBase抽象类又继承了RichParallelSourceFunction，所以要实现一个自定义的source时，有两种实现方式：一种是通过实现SourceFunction接口来自定义并行度为1的数据源；另一种是通过实现ParallelSourceFunction接口或者继承RichParallelSourceFunction来自定义具有并行度的数据源。FlinkKafkaConsumer的继承关系如下图所示。



源码解读

FlinkKafkaConsumer源码

先看一下FlinkKafkaConsumer的源码，为了方面阅读，本文将尽量给出本比较完整的源代码片段，具体如下所示：代码较长，在这里可以先有有一个总体的印象，下面会对重要的代码片段详细进行分析。

```
public class FlinkKafkaConsumer<T> extends FlinkKafkaConsumerBase<T> {

        // 配置轮询超时超时时间，使用flink.poll-timeout参数在properties进行配置
        public static final String KEY_POLL_TIMEOUT = "flink.poll-timeout"
;
        // 如果没有可用数据，则等待轮询所需的时间（以毫秒为单位）。 如果为0，则立即返
回所有可用的记录
        //默认轮询超时时间
        public static final long DEFAULT_POLL_TIMEOUT = 100L;
        // 用户提供的kafka 参数配置
        protected final Properties properties;
        // 如果没有可用数据，则等待轮询所需的时间（以毫秒为单位）。 如果为0，则立即返
回所有可用的记录
        protected final long pollTimeout;
        /**
         * 创建一个kafka的consumer source
         * @param topic                     消费的主题名称
         * @param valueDeserializer         反序列化类型，用于将kafka的字节消息
转换为Flink的对象
         * @param props                     用户传入的kafka参数
         */
        public FlinkKafkaConsumer(String topic, DeserializationSchema<T>
 valueDeserializer, Properties props) {
                this(Collections.singletonList(topic), valueDeserializer
, props);
        }
        /**
         * 创建一个kafka的consumer source
         * 该构造方法允许传入KafkaDeserializationSchema，该反序列化类支持访问ka
fka消费的额外信息
         * 比如：key/value对，offsets(偏移量)，topic(主题名称)
         * @param topic                 消费的主题名称
         * @param deserializer          反序列化类型，用于将kafka的字节消息转换
为Flink的对象
         * @param props                 用户传入的kafka参数
         */
        public FlinkKafkaConsumer(String topic, KafkaDeserializationSche
ma<T> deserializer, Properties props) {
                this(Collections.singletonList(topic), deserializer, pro
ps);
```

```
        }
        /**
         * 创建一个kafka的consumer source
         * 该构造方法允许传入多个topic(主题)，支持消费多个主题
         * @param topics           消费的主题名称，多个主题为List集合
         * @param deserializer     反序列化类型，用于将kafka的字节消息转换为Fli
nk的对象
         * @param props            用户传入的kafka参数
         */
        public FlinkKafkaConsumer(List<String> topics, DeserializationSc
hema<T> deserializer, Properties props) {
                this(topics, new KafkaDeserializationSchemaWrapper<>(des
erializer), props);
        }
        /**
         * 创建一个kafka的consumer source
         * 该构造方法允许传入多个topic(主题)，支持消费多个主题,
         * @param topics          消费的主题名称，多个主题为List集合
         * @param deserializer    反序列化类型，用于将kafka的字节消息转换为Flin
k的对象,支持获取额外信息
         * @param props           用户传入的kafka参数
         */
        public FlinkKafkaConsumer(List<String> topics, KafkaDeserializat
ionSchema<T> deserializer, Properties props) {
                this(topics, null, deserializer, props);
        }
        /**
         * 基于正则表达式订阅多个topic
         * 如果开启了分区发现，即FlinkKafkaConsumer.KEY_PARTITION_DISCOVERY_
INTERVAL_MILLIS值为非负数
         * 只要是能够正则匹配上，主题一旦被创建就会立即被订阅
         * @param subscriptionPattern    主题的正则表达式
         * @param valueDeserializer    反序列化类型，用于将kafka的字节消息转换
为Flink的对象,支持获取额外信息
         * @param props                 用户传入的kafka参数
         */
        public FlinkKafkaConsumer(Pattern subscriptionPattern, Deseriali
zationSchema<T> valueDeserializer, Properties props) {
                this(null, subscriptionPattern, new KafkaDeserialization
SchemaWrapper<>(valueDeserializer), props);
        }
        /**
         * 基于正则表达式订阅多个topic
         * 如果开启了分区发现，即FlinkKafkaConsumer.KEY_PARTITION_DISCOVERY_
INTERVAL_MILLIS值为非负数
         * 只要是能够正则匹配上，主题一旦被创建就会立即被订阅
         * @param subscriptionPattern    主题的正则表达式
         * @param deserializer           该反序列化类支持访问kafka消费的额外信
息,比如：key/value对，offsets(偏移量)，topic(主题名称)
```

```
90          * @param props                    用户传入的kafka参数
91          */
92         public FlinkKafkaConsumer(Pattern subscriptionPattern, KafkaDese
93 rializationSchema<T> deserializer, Properties props) {
94                 this(null, subscriptionPattern, deserializer, props);
95         }
96         private FlinkKafkaConsumer(
97                 List<String> topics,
98                 Pattern subscriptionPattern,
99                 KafkaDeserializationSchema<T> deserializer,
100                Properties props) {
101                // 调用父类(FlinkKafkaConsumerBase)构造方法，PropertiesUtil
102 .getLong方法第一个参数为Properties，第二个参数为key，第三个参数为value默认值
103                super(
104                        topics,
105                        subscriptionPattern,
106                        deserializer,
107                        getLong(
108                                checkNotNull(props, "props"),
109                                KEY_PARTITION_DISCOVERY_INTERVAL_MILLIS,
110  PARTITION_DISCOVERY_DISABLED),
111                        !getBoolean(props, KEY_DISABLE_METRICS, false));
112
113                this.properties = props;
114                setDeserializer(this.properties);
115
116                // 配置轮询超时时间，如果在properties中配置了KEY_POLL_TIMEOUT
117 参数，则返回具体的配置值，否则返回默认值DEFAULT_POLL_TIMEOUT
118                try {
119                        if (properties.containsKey(KEY_POLL_TIMEOUT)) {
120                                this.pollTimeout = Long.parseLong(proper
121 ties.getProperty(KEY_POLL_TIMEOUT));
122                        } else {
123                                this.pollTimeout = DEFAULT_POLL_TIMEOUT;
124                        }
125                }
126                catch (Exception e) {
127                        throw new IllegalArgumentException("Cannot parse
128  poll timeout for '" + KEY_POLL_TIMEOUT + '\'', e);
129                }
130        }
131    // 父类(FlinkKafkaConsumerBase)方法重写，该方法的作用是返回一个fetcher实例，
132        // fetcher的作用是连接kafka的broker，拉去数据并进行反序列化，然后将数据输
133 出为数据流(data stream)
134        @Override
135        protected AbstractFetcher<T, ?> createFetcher(
136                SourceContext<T> sourceContext,
137                Map<KafkaTopicPartition, Long> assignedPartitionsWithIni
138 tialOffsets,
```

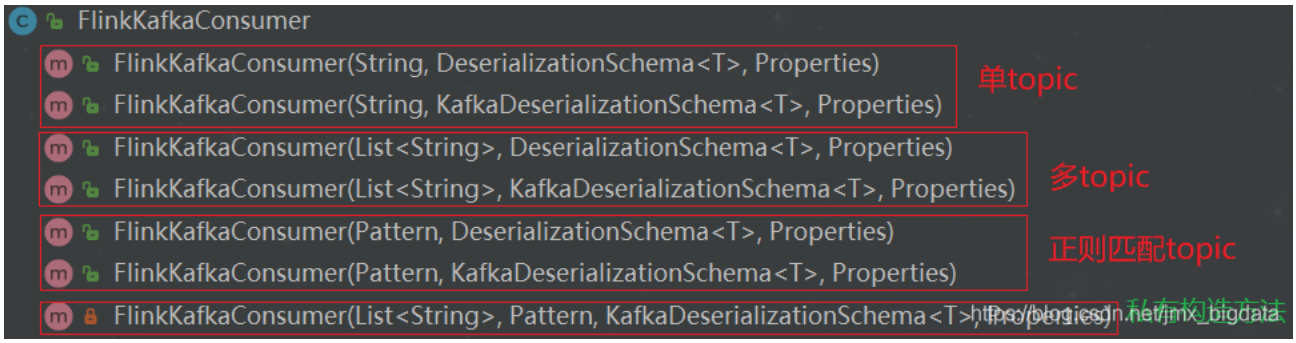
```
                SerializedValue<AssignerWithPeriodicWatermarks<T>> water
marksPeriodic,
                SerializedValue<AssignerWithPunctuatedWatermarks<T>> wat
ermarksPunctuated,
                StreamingRuntimeContext runtimeContext,
                OffsetCommitMode offsetCommitMode,
                MetricGroup consumerMetricGroup,
                boolean useMetrics) throws Exception {
        // 确保当偏移量的提交模式为ON_CHECKPOINTS(条件1：开启checkpoint，条件2
：consumer.setCommitOffsetsOnCheckpoints(true))时，禁用自动提交
                // 该方法为父类(FlinkKafkaConsumerBase)的静态方法
                // 这将覆盖用户在properties中配置的任何设置
                // 当offset的模式为ON_CHECKPOINTS，或者为DISABLED时，会将用户
配置的properties属性进行覆盖
                // 具体是将ENABLE_AUTO_COMMIT_CONFIG = "enable.auto.commit
"的值重置为"false
        // 可以理解为：如果开启了checkpoint，并且设置了consumer.setCommitOffse
tsOnCheckpoints(true)，默认为true，
                // 就会将kafka properties的enable.auto.commit强制置为false
                adjustAutoCommitConfig(properties, offsetCommitMode);
                return new KafkaFetcher<>(
                        sourceContext,
                        assignedPartitionsWithInitialOffsets,
                        watermarksPeriodic,
                        watermarksPunctuated,
                        runtimeContext.getProcessingTimeService(),
                        runtimeContext.getExecutionConfig().getAutoWaterm
arkInterval(),
                        runtimeContext.getUserCodeClassLoader(),
                        runtimeContext.getTaskNameWithSubtasks(),
                        deserializer,
                        properties,
                        pollTimeout,
                        runtimeContext.getMetricGroup(),
                        consumerMetricGroup,
                        useMetrics);
        }
        //父类(FlinkKafkaConsumerBase)方法重写
        // 返回一个分区发现类，分区发现可以使用kafka broker的高级consumer API发
现topic和partition的元数据
        @Override
        protected AbstractPartitionDiscoverer createPartitionDiscoverer(
                KafkaTopicsDescriptor topicsDescriptor,
                int indexOfThisSubtask,
                int numParallelSubtasks) {

                return new KafkaPartitionDiscoverer(topicsDescriptor, in
dexOfThisSubtask, numParallelSubtasks, properties);
        }
```

```
        /**
         *判断是否在kafka的参数开启了自动提交，即enable.auto.commit=true,
         * 并且auto.commit.interval.ms>0,
         * 注意：如果没有没有设置enable.auto.commit的参数，则默认为true
         *       如果没有设置auto.commit.interval.ms的参数，则默认为5000毫秒
         * @return
         */
        @Override
        protected boolean getIsAutoCommitEnabled() {
                //
                return getBoolean(properties, ConsumerConfig.ENABLE_AUTO_
COMMIT_CONFIG, true) &&
                        PropertiesUtil.getLong(properties, ConsumerConfig
.AUTO_COMMIT_INTERVAL_MS_CONFIG, 5000) > 0;
        }

        /**
         * 确保配置了kafka消息的key与value的反序列化方式，
         * 如果没有配置，则使用ByteArrayDeserializer序列化器，
         * 该类的deserialize方法是直接将数据进行return，未做任何处理
         * @param props
         */
        private static void setDeserializer(Properties props) {
                final String deSerName = ByteArrayDeserializer.class.getN
ame();

                Object keyDeSer = props.get(ConsumerConfig.KEY_DESERIALIZ
ER_CLASS_CONFIG);
                Object valDeSer = props.get(ConsumerConfig.VALUE_DESERIAL
IZER_CLASS_CONFIG);

                if (keyDeSer != null && !keyDeSer.equals(deSerName)) {
                        LOG.warn("Ignoring configured key DeSerializer ({
})", ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG);
                }
                if (valDeSer != null && !valDeSer.equals(deSerName)) {
                        LOG.warn("Ignoring configured value DeSerializer
({})", ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG);
                }
                props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, d
eSerName);
                props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
 deSerName);
        }
}
```

分析

上面的代码已经给出了非常详细的注释，下面将对比较关键的部分进行分析。

- 构造方法分析



FlinkKakfaConsumer提供了7种构造方法，如上图所示。不同的构造方法分别具有不同的功能，通过传递的参数也可以大致分析出每种构造方法特有的功能，为了方便理解，本文将对其进行分组讨论，具体如下：

单topic

```
/**
         * 创建一个kafka的consumer source
         * @param topic                 消费的主题名称
         * @param valueDeserializer     反序列化类型，用于将kafka的字节消息转换为Flink的对象
         * @param props                 用户传入的kafka参数
         */
        public FlinkKafkaConsumer(String topic, DeserializationSchema<T> valueDeserializer, Properties props) {
                this(Collections.singletonList(topic), valueDeserializer, props);
        }

/**
         * 创建一个kafka的consumer source
         * 该构造方法允许传入KafkaDeserializationSchema，该反序列化类支持访问kafka消费的额外信息
         * 比如：key/value对，offsets(偏移量)，topic(主题名称)
         * @param topic                 消费的主题名称
         * @param deserializer          反序列化类型，用于将kafka的字节消息转换为Flink的对象
         * @param props                 用户传入的kafka参数
         */
        public FlinkKafkaConsumer(String topic, KafkaDeserializationSchema<T> deserializer, Properties props) {
                this(Collections.singletonList(topic), deserializer, props);
        }
```

上面两种构造方法只支持单个topic，区别在于反序列化的方式不一样。第一种使用的是DeserializationSchema，第二种使用的是KafkaDeserializationSchema，其中使用带有KafkaDeserializationSchema参数的构造方法可以获取更多的附属信息，比如在某些场景下需要获取key/value对，offsets(偏移量)，topic(主题名称)等信息，可以选择使用此方式的构造方法。以上两种方法都调用了私有的构造方法，私有构造方法的分析见下面。

多topic

```
/**
         * 创建一个kafka的consumer source
         * 该构造方法允许传入多个topic(主题)，支持消费多个主题
         * @param topics           消费的主题名称，多个主题为List集合
         * @param deserializer     反序列化类型，用于将kafka的字节消息转换为Fli
nk的对象
         * @param props            用户传入的kafka参数
         */
        public FlinkKafkaConsumer(List<String> topics, DeserializationSc
hema<T> deserializer, Properties props) {
                this(topics, new KafkaDeserializationSchemaWrapper<>(des
erializer), props);
        }
        /**
         * 创建一个kafka的consumer source
         * 该构造方法允许传入多个topic(主题)，支持消费多个主题，
         * @param topics          消费的主题名称，多个主题为List集合
         * @param deserializer    反序列化类型，用于将kafka的字节消息转换为Flin
k的对象,支持获取额外信息
         * @param props           用户传入的kafka参数
         */
        public FlinkKafkaConsumer(List<String> topics, KafkaDeserializat
ionSchema<T> deserializer, Properties props) {
                this(topics, null, deserializer, props);
        }
```

上面的两种多topic的构造方法，可以使用一个list集合接收多个topic进行消费，区别在于反序列化的方式不一样。第一种使用的是DeserializationSchema，第二种使用的是KafkaDeserializationSchema，其中使用带有KafkaDeserializationSchema参数的构造方法可以获取更多的附属信息，比如在某些场景下需要获取key/value对，offsets(偏移量)，topic(主题名称)等信息，可以选择使用此方式的构造方法。以上两种方法都调用了私有的构造方法，私有构造方法的分析见下面。

正则匹配topic

```
/**
         * 基于正则表达式订阅多个topic
         * 如果开启了分区发现，即FlinkKafkaConsumer.KEY_PARTITION_DISCOVERY_
INTERVAL_MILLIS值为非负数
         * 只要是能够正则匹配上，主题一旦被创建就会立即被订阅
         * @param subscriptionPattern    主题的正则表达式
         * @param valueDeserializer    反序列化类型，用于将kafka的字节消息转换
为Flink的对象,支持获取额外信息
         * @param props                用户传入的kafka参数
         */
        public FlinkKafkaConsumer(Pattern subscriptionPattern, Deseriali
zationSchema<T> valueDeserializer, Properties props) {
                this(null, subscriptionPattern, new KafkaDeserialization
SchemaWrapper<>(valueDeserializer), props);
        }
        /**
         * 基于正则表达式订阅多个topic
         * 如果开启了分区发现，即FlinkKafkaConsumer.KEY_PARTITION_DISCOVERY_
INTERVAL_MILLIS值为非负数
         * 只要是能够正则匹配上，主题一旦被创建就会立即被订阅
         * @param subscriptionPattern    主题的正则表达式
         * @param deserializer           该反序列化类支持访问kafka消费的额外信
息,比如：key/value对，offsets(偏移量)，topic(主题名称)
         * @param props                  用户传入的kafka参数
         */
        public FlinkKafkaConsumer(Pattern subscriptionPattern, KafkaDese
rializationSchema<T> deserializer, Properties props) {
                this(null, subscriptionPattern, deserializer, props);
        }
```

实际的生产环境中可能有这样一些需求，比如有一个flink作业需要将多种不同的数据聚合到一起，而这些数据对应着不同的kafka topic，随着业务增长，新增一类数据，同时新增了一个kafka topic，如何在不重启作业的情况下作业自动感知新的topic。首先需要在构建FlinkKafkaConsumer时的properties中设置flink.partition-discovery.interval-millis参数为非负值，表示开启动态发现的开关，以及设置的时间间隔。此时FLinkKafkaConsumer内部会启动一个单独的线程定期去kafka获取最新的meta信息。具体的调用执行信息，参见下面的私有构造方法

私有构造方法

```
        private FlinkKafkaConsumer(
                List<String> topics,
                Pattern subscriptionPattern,
                KafkaDeserializationSchema<T> deserializer,
                Properties props) {

```

```
                // 调用父类(FlinkKafkaConsumerBase)构造方法，PropertiesUtil
.getLong方法第一个参数为Properties，第二个参数为key，第三个参数为value默认值。KEY
_PARTITION_DISCOVERY_INTERVAL_MILLIS值是开启分区发现的配置参数，在properties里
面配置flink.partition-discovery.interval-millis=5000(大于0的数),如果没有配置则
使用PARTITION_DISCOVERY_DISABLED=Long.MIN_VALUE(表示禁用分区发现)
                super(
                        topics,
                        subscriptionPattern,
                        deserializer,
                        getLong(
                                checkNotNull(props, "props"),
                                KEY_PARTITION_DISCOVERY_INTERVAL_MILLIS,
 PARTITION_DISCOVERY_DISABLED),
                        !getBoolean(props, KEY_DISABLE_METRICS, false));

                this.properties = props;
                setDeserializer(this.properties);

                // 配置轮询超时时间，如果在properties中配置了KEY_POLL_TIMEOUT
参数，则返回具体的配置值，否则返回默认值DEFAULT_POLL_TIMEOUT
                try {
                        if (properties.containsKey(KEY_POLL_TIMEOUT)) {
                                this.pollTimeout = Long.parseLong(proper
ties.getProperty(KEY_POLL_TIMEOUT));
                        } else {
                                this.pollTimeout = DEFAULT_POLL_TIMEOUT;
                        }
                }
                catch (Exception e) {
                        throw new IllegalArgumentException("Cannot parse
 poll timeout for '" + KEY_POLL_TIMEOUT + '\'', e);
                }
        }
```

- 其他方法分析

KafkaFetcher对象创建

```
   // 父类(FlinkKafkaConsumerBase)方法重写，该方法的作用是返回一个fetcher实例，
        // fetcher的作用是连接kafka的broker，拉去数据并进行反序列化，然后将数据输
出为数据流(data stream)
        @Override
        protected AbstractFetcher<T, ?> createFetcher(
                SourceContext<T> sourceContext,
                Map<KafkaTopicPartition, Long> assignedPartitionsWithIni
tialOffsets,
                SerializedValue<AssignerWithPeriodicWatermarks<T>> water
```

```
marksPeriodic,
                SerializedValue<AssignerWithPunctuatedWatermarks<T>> wat
ermarksPunctuated,
                StreamingRuntimeContext runtimeContext,
                OffsetCommitMode offsetCommitMode,
                MetricGroup consumerMetricGroup,
                boolean useMetrics) throws Exception {
        // 确保当偏移量的提交模式为ON_CHECKPOINTS(条件1：开启checkpoint，条件2
：consumer.setCommitOffsetsOnCheckpoints(true))时，禁用自动提交
                // 该方法为父类(FlinkKafkaConsumerBase)的静态方法
                // 这将覆盖用户在properties中配置的任何设置
                // 当offset的模式为ON_CHECKPOINTS，或者为DISABLED时，会将用户
配置的properties属性进行覆盖
                // 具体是将ENABLE_AUTO_COMMIT_CONFIG = "enable.auto.commit
"的值重置为"false
        // 可以理解为：如果开启了checkpoint，并且设置了consumer.setCommitOffse
tsOnCheckpoints(true)，默认为true，
                // 就会将kafka properties的enable.auto.commit强制置为false
                adjustAutoCommitConfig(properties, offsetCommitMode);
                return new KafkaFetcher<>(
                        sourceContext,
                        assignedPartitionsWithInitialOffsets,
                        watermarksPeriodic,
                        watermarksPunctuated,
                        runtimeContext.getProcessingTimeService(),
                        runtimeContext.getExecutionConfig().getAutoWaterm
arkInterval(),
                        runtimeContext.getUserCodeClassLoader(),
                        runtimeContext.getTaskNameWithSubtasks(),
                        deserializer,
                        properties,
                        pollTimeout,
                        runtimeContext.getMetricGroup(),
                        consumerMetricGroup,
                        useMetrics);
        }
```

该方法的作用是返回一个fetcher实例，fetcher的作用是连接kafka的broker，拉去数据并进行反序列化，然后将数据输出为数据流(data stream)，在这里对自动偏移量提交模式进行了强制调整，即确保当偏移量的提交模式为ON_CHECKPOINTS(条件1：开启checkpoint，条件2：consumer.setCommitOffsetsOnCheckpoints(true))时，禁用自动提交。这将覆盖用户在properties中配置的任何设置，简单可以理解为：如果开启了checkpoint，并且设置了consumer.setCommitOffsetsOnCheckpoints(true)，默认为true，就会将kafka properties的enable.auto.commit强制置为false。关于offset的提交模式，见下文的偏移量提交模式分析。

判断是否设置了自动提交

```
   @Override
        protected boolean getIsAutoCommitEnabled() {
                //
                return getBoolean(properties, ConsumerConfig.ENABLE_AUTO_
COMMIT_CONFIG, true) &&
                        PropertiesUtil.getLong(properties, ConsumerConfig
.AUTO_COMMIT_INTERVAL_MS_CONFIG, 5000) > 0;
        }
```

判断是否在kafka的参数开启了自动提交，即enable.auto.commit=true，并且auto.commit.interval.ms>0, 注意：如果没有没有设置enable.auto.commit的参数，则默认为true, 如果没有设置auto.commit.interval.ms的参数，则默认为5000毫秒。该方法会在FlinkKafkaConsumerBase的open方法进行初始化的时候调用。

反序列化

```
private static void setDeserializer(Properties props) {
         // 默认的反序列化方式
                final String deSerName = ByteArrayDeserializer.class.getN
ame();
         //获取用户配置的properties关于key与value的反序列化模式
                Object keyDeSer = props.get(ConsumerConfig.KEY_DESERIALIZ
ER_CLASS_CONFIG);
                Object valDeSer = props.get(ConsumerConfig.VALUE_DESERIAL
IZER_CLASS_CONFIG);
         // 如果配置了，则使用用户配置的值
                if (keyDeSer != null && !keyDeSer.equals(deSerName)) {
                        LOG.warn("Ignoring configured key DeSerializer ({
})", ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG);
                }
                if (valDeSer != null && !valDeSer.equals(deSerName)) {
                        LOG.warn("Ignoring configured value DeSerializer
({})", ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG);
                }
        // 没有配置，则使用ByteArrayDeserializer进行反序列化
                props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, d
eSerName);
                props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
 deSerName);
        }
```

确保配置了kafka消息的key与value的反序列化方式，如果没有配置，则使用ByteArrayDeserializer序列化器，

ByteArrayDeserializer类的deserialize方法是直接将数据进行return，未做任何处理。

FlinkKafkaConsumerBase源码

```
@Internal
public abstract class FlinkKafkaConsumerBase<T> extends RichParallelSourceFunction<T> implements
                CheckpointListener,
                ResultTypeQueryable<T>,
                CheckpointedFunction {

        public static final int MAX_NUM_PENDING_CHECKPOINTS = 100;
        public static final long PARTITION_DISCOVERY_DISABLED = Long.MIN_VALUE;
        public static final String KEY_DISABLE_METRICS = "flink.disable-metrics";
        public static final String KEY_PARTITION_DISCOVERY_INTERVAL_MILLIS = "flink.partition-discovery.interval-millis";
        private static final String OFFSETS_STATE_NAME = "topic-partition-offset-states";
        private boolean enableCommitOnCheckpoints = true;
        /**
         * 偏移量的提交模式，仅能通过在FlinkKafkaConsumerBase#open(Configuration)进行配置
         * 该值取决于用户是否开启了checkpoint

         */
        private OffsetCommitMode offsetCommitMode;
        /**
         * 配置从哪个位置开始消费kafka的消息，
         * 默认为StartupMode#GROUP_OFFSETS，即从当前提交的偏移量开始消费
         */
        private StartupMode startupMode = StartupMode.GROUP_OFFSETS;
        private Map<KafkaTopicPartition, Long> specificStartupOffsets;
        private Long startupOffsetsTimestamp;

        /**
         * 确保当偏移量的提交模式为ON_CHECKPOINTS时，禁用自动提交，
         * 这将覆盖用户在properties中配置的任何设置。
         * 当offset的模式为ON_CHECKPOINTS，或者为DISABLED时，会将用户配置的properties属性进行覆盖
         * 具体是将ENABLE_AUTO_COMMIT_CONFIG = "enable.auto.commit"的值重置为"false，即禁用自动提交
         * @param properties       kafka配置的properties，会通过该方法进行覆盖
         * @param offsetCommitMode    offset提交模式
         */
        static void adjustAutoCommitConfig(Properties properties, OffsetC
```

```
ommitMode offsetCommitMode) {
                if (offsetCommitMode == OffsetCommitMode.ON_CHECKPOINTS
|| offsetCommitMode == OffsetCommitMode.DISABLED) {
                        properties.setProperty(ConsumerConfig.ENABLE_AUTO
_COMMIT_CONFIG, "false");
                }
        }

        /**
         * 决定是否在开启checkpoint时，在checkpoin之后提交偏移量，
         * 只有用户配置了启用checkpoint，该参数才会其作用
         * 如果没有开启checkpoint，则使用kafka的配置参数：enable.auto.commit
         * @param commitOnCheckpoints
         * @return
         */
        public FlinkKafkaConsumerBase<T> setCommitOffsetsOnCheckpoints(b
oolean commitOnCheckpoints) {
                this.enableCommitOnCheckpoints = commitOnCheckpoints;
                return this;
        }
        /**
         * 从最早的偏移量开始消费，
         *该模式下，Kafka 中的已经提交的偏移量将被忽略，不会用作起始位置。
         *可以通过consumer1.setStartFromEarliest()进行设置
         */
        public FlinkKafkaConsumerBase<T> setStartFromEarliest() {
                this.startupMode = StartupMode.EARLIEST;
                this.startupOffsetsTimestamp = null;
                this.specificStartupOffsets = null;
                return this;
        }

        /**
         * 从最新的数据开始消费，
         *  该模式下，Kafka 中的 已提交的偏移量将被忽略，不会用作起始位置。
         *
         */
        public FlinkKafkaConsumerBase<T> setStartFromLatest() {
                this.startupMode = StartupMode.LATEST;
                this.startupOffsetsTimestamp = null;
                this.specificStartupOffsets = null;
                return this;
        }

        /**
         *指定具体的偏移量时间戳,毫秒
         *对于每个分区，其时间戳大于或等于指定时间戳的记录将用作起始位置。
         * 如果一个分区的最新记录早于指定的时间戳，则只从最新记录读取该分区数据。
         * 在这种模式下，Kafka 中的已提交 offset 将被忽略，不会用作起始位置。
```

```
94          */
95         protected FlinkKafkaConsumerBase<T> setStartFromTimestamp(long s
96 tartupOffsetsTimestamp) {
97                 checkArgument(startupOffsetsTimestamp >= 0, "The provide
98 d value for the startup offsets timestamp is invalid.");
99 
100                long currentTimestamp = System.currentTimeMillis();
101                checkArgument(startupOffsetsTimestamp <= currentTimestam
102 p,
103                        "Startup time[%s] must be before current time[%s
104 ].", startupOffsetsTimestamp, currentTimestamp);
105 
106                this.startupMode = StartupMode.TIMESTAMP;
107                this.startupOffsetsTimestamp = startupOffsetsTimestamp;
108                this.specificStartupOffsets = null;
109                return this;
110        }
111 
112        /**
113         *
114         * 从具体的消费者组最近提交的偏移量开始消费，为默认方式
115         * 如果没有发现分区的偏移量，使用auto.offset.reset参数配置的值
116         * @return
117         */
118        public FlinkKafkaConsumerBase<T> setStartFromGroupOffsets() {
119                this.startupMode = StartupMode.GROUP_OFFSETS;
120                this.startupOffsetsTimestamp = null;
121                this.specificStartupOffsets = null;
122                return this;
123        }
124 
125        /**
126         *为每个分区指定偏移量进行消费
127         */
128        public FlinkKafkaConsumerBase<T> setStartFromSpecificOffsets(Map<
129 KafkaTopicPartition, Long> specificStartupOffsets) {
130                this.startupMode = StartupMode.SPECIFIC_OFFSETS;
131                this.startupOffsetsTimestamp = null;
132                this.specificStartupOffsets = checkNotNull(specificStartup
133 Offsets);
134                return this;
135        }
136        @Override
137        public void open(Configuration configuration) throws Exception {
138                // determine the offset commit mode
139                // 决定偏移量的提交模式，
140                // 第一个参数为是否开启了自动提交，
141                // 第二个参数为是否开启了CommitOnCheckpoint模式
142                // 第三个参数为是否开启了checkpoint
```

```
                this.offsetCommitMode = OffsetCommitModes.fromConfigurati
on(
                                getIsAutoCommitEnabled(),
                                enableCommitOnCheckpoints,
                                ((StreamingRuntimeContext) getRuntimeCon
text()).isCheckpointingEnabled());

            // 省略的代码
        }

// 省略的代码
        /**
         * 创建一个fetcher用于连接kafka的broker，拉去数据并进行反序列化，然后将数
据输出为数据流(data stream)
         * @param sourceContext    数据输出的上下文
         * @param subscribedPartitionsToStartOffsets  当前sub task需要处理
的topic分区集合，即topic的partition与offset的Map集合
         * @param watermarksPeriodic     可选,一个序列化的时间戳提取器，生成per
iodic类型的 watermark
         * @param watermarksPunctuated  可选,一个序列化的时间戳提取器，生成pun
ctuated类型的 watermark
         * @param runtimeContext        task的runtime context上下文
         * @param offsetCommitMode      offset的提交模式,有三种，分别为：DISA
BLED(禁用偏移量自动提交),ON_CHECKPOINTS(仅仅当checkpoints完成之后，才提交偏移量
给kafka)
         * KAFKA_PERIODIC(使用kafka自动提交函数，周期性自动提交偏移量)
         * @param kafkaMetricGroup   Flink的Metric
         * @param useMetrics         是否使用Metric
         * @return                   返回一个fetcher实例
         * @throws Exception
         */
        protected abstract AbstractFetcher<T, ?> createFetcher(
                        SourceContext<T> sourceContext,
                        Map<KafkaTopicPartition, Long> subscribedPartiti
onsToStartOffsets,
                        SerializedValue<AssignerWithPeriodicWatermarks<T
>> watermarksPeriodic,
                        SerializedValue<AssignerWithPunctuatedWatermarks
<T>> watermarksPunctuated,
                        StreamingRuntimeContext runtimeContext,
                        OffsetCommitMode offsetCommitMode,
                        MetricGroup kafkaMetricGroup,
                        boolean useMetrics) throws Exception;
        protected abstract boolean getIsAutoCommitEnabled();
        // 省略的代码
}
```

上述代码是FlinkKafkaConsumerBase的部分代码片段，基本上对其做了详细注释，里面的有些方法是FlinkKafkaConsumer继承的，有些是重写的。之所以在这里给出，可以对照FlinkKafkaConsumer的源码，从而方便理解。

偏移量提交模式分析

Flink Kafka Consumer 允许有配置如何将 offset 提交回 Kafka broker（或 0.8 版本的 Zookeeper）的行为。请注意：Flink Kafka Consumer 不依赖于提交的 offset 来实现容错保证。提交的 offset 只是一种方法，用于公开 consumer 的进度以便进行监控。

配置 offset 提交行为的方法是否相同，取决于是否为 job 启用了 checkpointing。在这里先给出提交模式的具体结论，下面会对两种方式进行具体的分析。基本的结论为：

- 开启checkpoint
  - 情况1：用户通过调用 consumer 上的 setCommitOffsetsOnCheckpoints(true) 方法来启用 offset 的提交(默认情况下为 true)
    那么当 checkpointing 完成时，Flink Kafka Consumer 将提交的 offset 存储在 checkpoint 状态中。
    这确保 Kafka broker 中提交的 offset 与 checkpoint 状态中的 offset 一致。
    注意，在这个场景中，Properties 中的自动定期 offset 提交设置会被完全忽略。
    此情况使用的是ON_CHECKPOINTS
  - 情况2：用户通过调用 consumer 上的 setCommitOffsetsOnCheckpoints(“false”) 方法来禁用 offset 的提交，则使用DISABLED模式提交offset
- 未开启checkpoint
  Flink Kafka Consumer 依赖于内部使用的 Kafka client 自动定期 offset 提交功能，因此，要禁用或启用 offset 的提交
- 情况1：配置了Kafka properties的参数配置了"enable.auto.commit" = "true"或者 Kafka 0.8 的 auto.commit.enable=true，使用KAFKA_PERIODIC模式提交offset，即自动提交offset
  - 情况2：没有配置enable.auto.commit参数，使用DISABLED模式提交offset，这意味着kafka不知道当前的消费者组的消费者每次消费的偏移量。

提交模式源码分析

- offset的提交模式

```
public enum OffsetCommitMode {
        // 禁用偏移量自动提交
        DISABLED,
        // 仅仅当checkpoints完成之后，才提交偏移量给kafka
```

```
        ON_CHECKPOINTS,
        // 使用kafka自动提交函数，周期性自动提交偏移量
        KAFKA_PERIODIC;
}

```

- 提交模式的调用

```
public class OffsetCommitModes {
        public static OffsetCommitMode fromConfiguration(
                        boolean enableAutoCommit,
                        boolean enableCommitOnCheckpoint,
                        boolean enableCheckpointing) {
                // 如果开启了checkinpoint，执行下面判断
                if (enableCheckpointing) {
                        // 如果开启了checkpoint，进一步判断是否在checkpoin启用时提交(setCommitOffsetsOnCheckpoints(true))，如果是则使用ON_CHECKPOINTS模式
                        // 否则使用DISABLED模式
                        return (enableCommitOnCheckpoint) ? OffsetCommitMode.ON_CHECKPOINTS : OffsetCommitMode.DISABLED;
                } else {
                        // 若Kafka properties的参数配置了"enable.auto.commit" = "true"，则使用KAFKA_PERIODIC模式提交offset
                        // 否则使用DISABLED模式
                        return (enableAutoCommit) ? OffsetCommitMode.KAFKA_PERIODIC : OffsetCommitMode.DISABLED;
                }
        }
}
```

小结

本文主要介绍了Flink Kafka Consumer，首先对FlinkKafkaConsumer的不同版本进行了对比，然后给出了一个完整的Demo案例，并对案例的配置参数进行了详细解释，接着分析了FlinkKafkaConsumer的继承关系，并分别对FlinkKafkaConsumer以及其父类FlinkKafkaConsumerBase的源码进行了解读，最后从源码层面分析了Flink Kafka Consumer的偏移量提交模式，并对每一种提交模式进行了梳理。