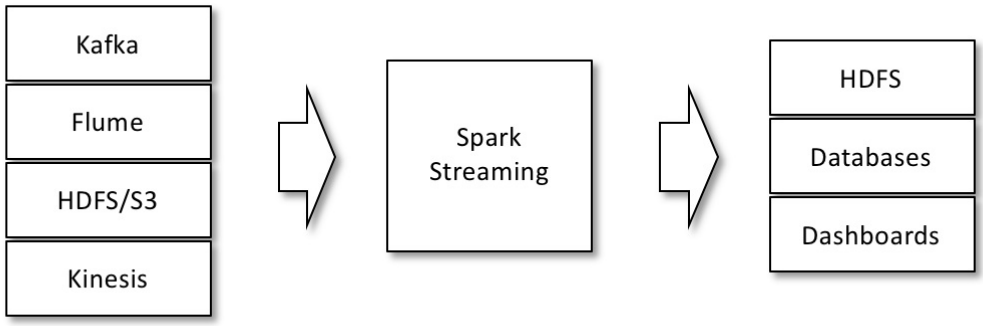
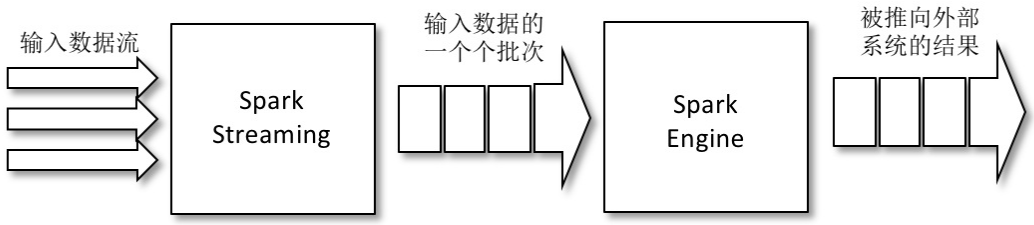
Spark Streaming简介

Spark Streaming 是 Spark 提供的对实时数据进行流式计算的组件。它是 Spark 核心 API 的一个扩展，具有吞吐量高、容错能力强的实时流数据处理系统，支持包括 Kafka、Flume、Kinesis 和 TCP 套接字等数据源，获取数据以后可以使用 map()、reduce()、join()、window() 等高级函数进行复杂算法的处理，处理结果可以存储到文件系统、数据库，或者展示到实时数据大盘等。另外，Sparking Streaming 也可以和其他组件，如 MLlib和Graphx 等结合，对实时数据进行更加复杂的处理。Spark Streaming 的数据处理流程如下图所示。



和 Spark 基于 RDD 的概念很相似，Spark Streaming 使用离散化流（Discretized Stream）作为抽象表示，叫作 DStream。DStream 是随着时间推移而收到的数据的序列。在内部，每个时间区间收到的数据都作为 RDD 存在，而 DStream 是由这些 RDD 组成的序列（因此得名“离散化”）。创建出来的 DStream 支持两种操作：一种是转换操作（Transformation），会生成一个新的 DStream；另一种是输出操作（Output Operation），可以把数据写入外部系统。



如上图所示，通俗一点讲，Spark Streaming 会把实时输入的数据流以时间片Δt（如1秒）为单位切分成块，每块数据代表一个 RDD。流数据的 DStream 可以看作一组 RDD 序列，通过调用 Spark 核心的作业处理这些批数据，最终得到处理后的一批批结果数据。

在开始讲解 Spark Streaming 的细节之前，让我们先来看一个简单的例子。Spark Streaming 对应的 Maven 依赖如下：

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.11</artifactId>
    <version>2.3.1</version>
</dependency>
```

下面就以 Spark Streaming 官方提供的单词统计代码为例来分析 Spark Streaming 的相关内容，具体的代码如代码清单34-1所示。

```
//代码清单34-1 Spark Streaming示例程序
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds,
StreamingContext}

object StreamingWordCount {
  def main(args:Array[String]): Unit ={
    val conf = new
SparkConf().setMaster("local[2]").setAppName("Wor
dCount")①
    val ssc = new StreamingContext(conf,
Seconds(1))②
    val lines = ssc.socketTextStream("localhost",
9999)③
    val words = lines.flatMap(_.split(" "))④
    val pairs = words.map(word => (word, 1))⑤
    val wordCounts = pairs.reduceByKey(_ + _)⑥
    wordCounts.print()⑦
    ssc.start()⑧
    ssc.awaitTermination()⑨
  }
}
```

示例代码首先从创建 StreamingContext 开始，它是流计算功能的主要入口。StreamingContext 会在底层创建出 SparkContext，用来处理数据。StreamingContext 的构造函数还接收用来指定多长时间处理一次新数据的批次间隔（Batch Duration）作为输入，这里把它设置为1秒。接着调用 socketTextStream() 来创建基于本地9999端口上收到的文本数据的 DStream。第④行至第⑥行的内容和前面单词统计代码如出一辙，这里就不过多解释，只不过这里针对的是 DStream 的处理。第⑦行使用输出操作来将结果打印出来。

到这里为止只是设定好了要进行的计算，系统收到数据时计算就会开始。要开始接收数据就必须如第⑧行一样显式调用StreamingContext的start() 方法。这样，Spark Streaming 就会开始把 Spark 作业不断交给下面的 SparkContext 去调度执行。执行会在另一个线程中进行，所以需要调用 awaitTermination() 方法来等待流计算完成，以防止应用退出。

示例代码中的内容是基于批次间隔的处理，这个也可以看作基于固定窗口（Fixed Window）的处理，每个窗口不会重合，固定窗口的大小就是批次间隔的大小。这里对应的转换操作也就可以看作基于固定窗口的转换操作。

Spark 安装包中自带了这个程序，所以可以直接使用如下的方式来启动这个程序：

```
[root@node1  spark]# bin/run-example
streaming.NetworkWordCount localhost 9999
2018-08-06 18:06:47 WARN  NativeCodeLoader:62 -
Unable to load native-hadoop
     library for your platform... using builtin-
java classes where applicable
2018-08-06 18:06:48 INFO  SparkContext:54 -
Running Spark version 2.3.1
2018-08-06 18:06:48 INFO  SparkContext:54 -
Submitted application:
     NetworkWordCount
2018-08-06 18:06:48 INFO  SecurityManager:54 -
Changing view acls to: root
2018-08-06 18:06:48 INFO  SecurityManager:54 -
Changing modify acls to: root
2018-08-06 18:06:48 INFO  SecurityManager:54 -
Changing view acls groups to:
(....省略若干信息)
```

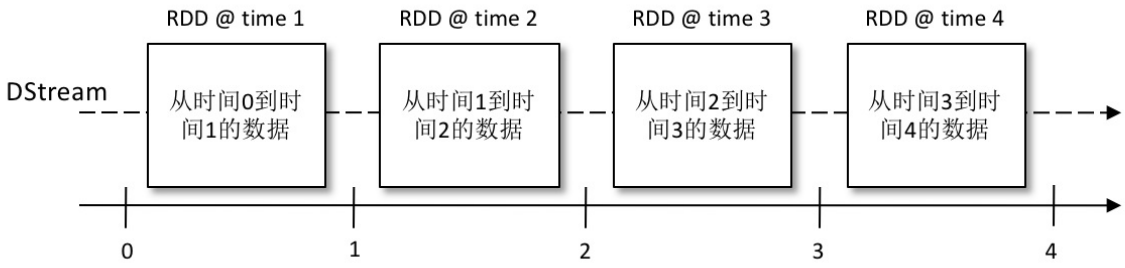
接着在另一个 shell 中使用 netcat 工具来输入一句“hello world”，示例如下：

```
[root@node1 spark]# nc -lk 9999
hello world
```

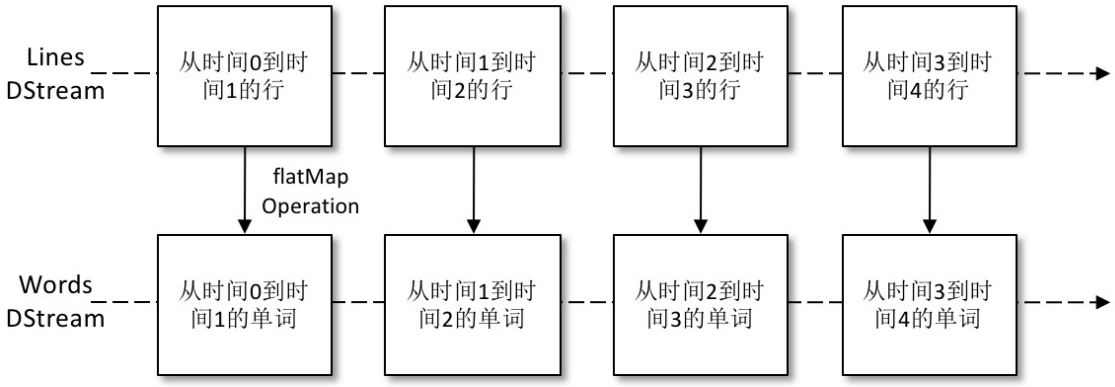
可以看到在 NetworkWordCount 程序中输出如下信息：

```
-------------------------------------------
Time: 1533549417000 ms
-------------------------------------------
(hello,1)
(world,1)
```

前面已经讲过，Spark Streaming 的编程抽象是离散化流，也就是 DStream，如下图所示。它是一个 RDD 序列，每个 RDD 代表数据流中一个时间片内的数据。



可以从外部输入源来创建 DStream，也可以对其他 DStream 应用进行转换操作得到新的 DStream。DStream 支持许多 RDD 支持的转换操作。以代码清单34-1为例，第④行代码中的 flatMap() 就是将行数据流（Lines DStream）中的 RDD 转换成单词数据流（Words DStream）中的 RDD，如下图所示。

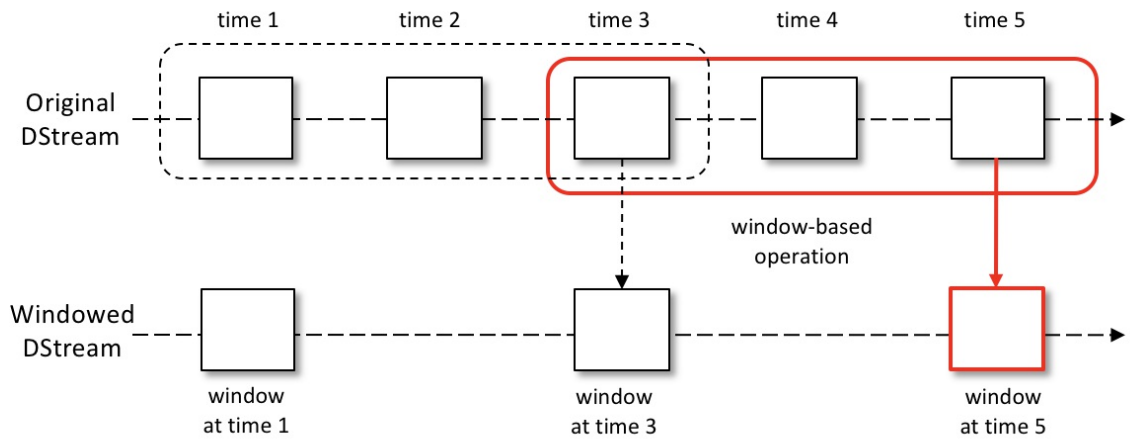


这些基础 RDD 的转换由 Spark 引擎计算。DStream 操作隐藏了大部分的细节，并为开发人员提供了更高级别的 API 以方便使用。

DStream 还支持输出操作，比如在示例中使用的 print()。输出操作和 RDD 的行动操作的概念类似。Spark 在行动操作中将数据写入外部系统，而 Spark Streaming 的输出操作在每个时间区间中周期性地执行，每个批次都生成输出。

除了上面提及的固定窗口的转换操作，Spark Streaming 还提供了基于滑动窗口（Sliding Window，相邻的窗口间会有重合部分）的转换操作，它会在一个比 StreamingContext 的批次间隔更长的时间范围内，通过整合多个批次的结果，计算出整个窗口的结果。

对滑动窗口操作而言，在其窗口内部会有N个批次数据，批次数据的个数由窗口间隔（Window Duration）决定，其为窗口持续的时间，在窗口操作中只有窗口间隔满足了才会触发批数据的处理。处理窗口的长度，另一个重要的参数就是滑动间隔（Slide Duration），它指的是经过多长时间窗口滑动一次形成新的窗口，滑动间隔默认情况下和批次间隔的相同，而窗口间隔一般设置得要比它们都大。需要注意的是，窗口间隔和滑动间隔的大小一定要设置为批次间隔的整数倍。



如上图所示，批次间隔是1个时间单位，窗口间隔是3个时间单位，滑动间隔是2个时间单位。对于初始的窗口 time1 至 time3，只有窗口间隔满足了才会触发数据的处理。这里需要注意的是，初始时有可能流入的数据没有撑满窗口，但是随着时间的推进，窗口最终会被撑满。每隔2个时间单位窗口滑动一次，会有新的数据流入窗口，这时窗口会移除最早的两个时间单位的数据，而与最新的两个时间单位的数据进行汇总形成新的窗口，即 time3 至 time5。

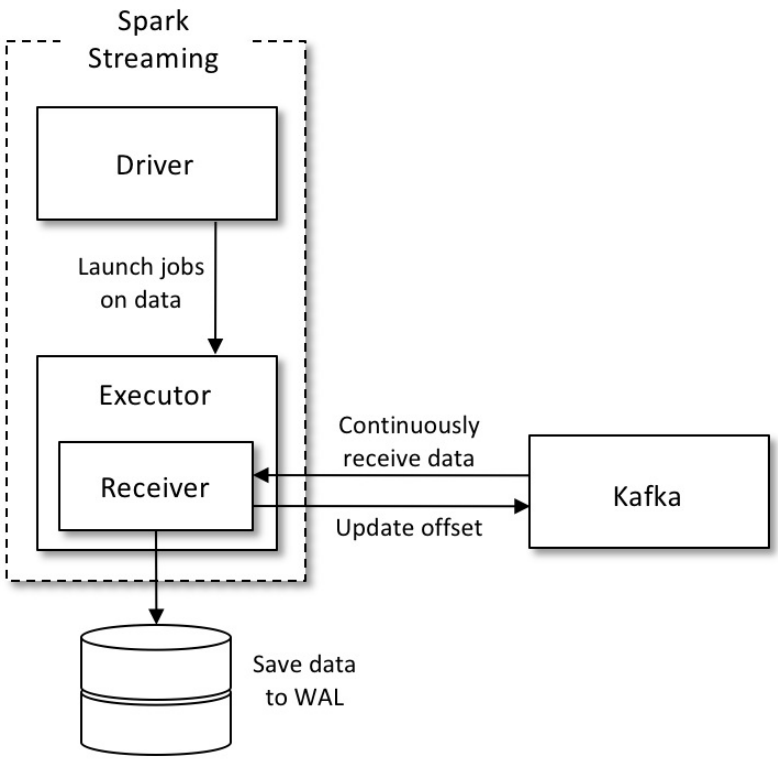
以代码清单34-1为例，如果我们想每隔10秒计算最近30秒的单词总数，那么可以将代码清单34-1中的第⑥行修改为如下语句：

```
val windowedWordCounts =
pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a +
b),
     Seconds(30), Seconds(10))
```

这里就涉及滑动窗口操作的两个参数：窗口间隔，也就是这里的30s；滑动间隔，也就是这里的10s。

Kafka与Spark Streaming的整合

采用 Spark Streaming 流式处理 Kafka 中的数据，首先需要把数据从 Kafka 中接收过来，然后转换为 Spark Streaming 中的 DStream。接收数据的方式一共有两种：利用接收器 Receiver 的方式接收数据和直接从 Kafka 中读取数据。

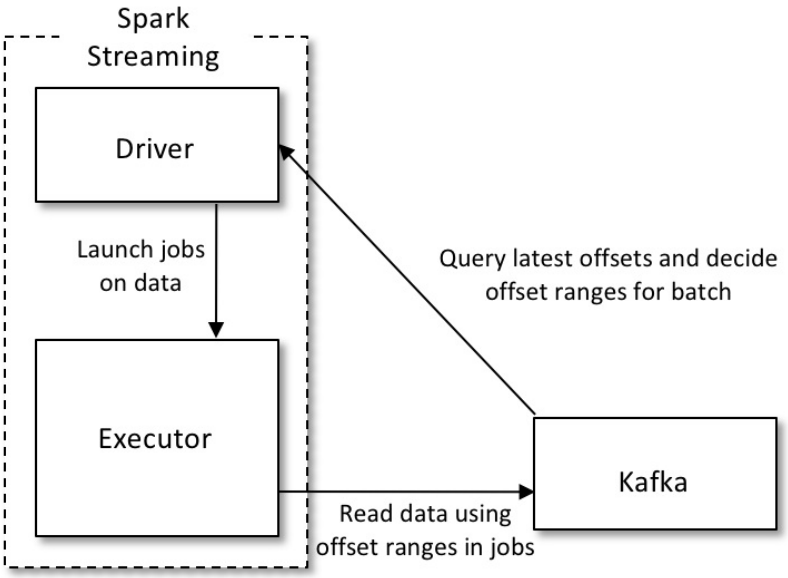


Receiver 方式通过 KafkaUtils.createStream() 方法来创建一个 DStream 对象，它不关注消费位移的处理，Receiver 方式的结构如上图所示。但这种方式在 Spark 任务执行异常时会导致数据丢失，如果要保证数据的可靠性，则需要开启预写式日志，简称 WAL（Write Ahead Logs），只有收到的数据被持久化到 WAL 之后才会更新 Kafka 中的消费位移。收到的数据和 WAL 存储位置信息被可靠地存储，如果期间出现故障，那么这些信息被用来从错误中恢复，并继续处理数据。

WAL 的方式可以保证从 Kafka 中接收的数据不被丢失。但是在某些异常情况下，一些数据被可靠地保存到了 WAL 中，但是还没有来得及更新消费位移，这样会造成 Kafka 中的数据被 Spark 拉取了不止

一次。同时在 Receiver 方式中，Spark 的 RDD 分区和 Kafka 的分区并不是相关的，因此增加 Kafka 中主题的分区数并不能增加 Spark 处理的并行度，仅仅增加了接收器接收数据的并行度。

Direct 方式是从 Spark 1.3 开始引入的，它通过 KafkaUtils.createDirectStream() 方法创建一个 DStream 对象，Direct 方式的结构如下图所示。该方式中 Kafka 的一个分区与 Spark RDD 对应，通过定期扫描所订阅的 Kafka 每个主题的每个分区的最新偏移量以确定当前批处理数据偏移范围。与 Receiver 方式相比，Direct 方式不需要维护一份 WAL 数据，由 Spark Streaming 程序自己控制位移的处理，通常通过检查点机制处理消费位移，这样可以保证 Kafka 中的数据只会被 Spark 拉取一次。



注意使用 Direct 的方式并不意味着实现了精确一次的语义（Exactly Once Semantics），如果要达到精确一次的语义标准，则还需要配合幂等性操作或事务性操作。

在 Spark 官网中，关于 Spark Streaming 与 Kafka 集成给出了两个依赖版本，一个是基于 Kafka 0.8 之后的版本（spark-streaming-kafka-0-8），另一个是基于 Kafka 0.10 及其之后的版本（spark-streaming-kafka-0-10）。spark-streaming-

kafka-0-8 版本的 Kafka 与 Spark Streaming 集成有 Receiver 方式和 Direct 方式这两种接收数据的方式，不过 spark-streaming-kafka-0-8 从 Spark 2.3.0 开始被标注为“弃用”。而 spark-streaming-kafka-0-10 版本只提供 Direct 方式，同时底层使用的是新消费者客户端 KafkaConsumer 而不是之前的旧消费者客户端，因此通过 KafkaUtils.createDirectStream() 方法构建的 DStream 数据集是 ConsumerRecord 类型。下表中给出了两个版本的更多细节对比。

兼容性比较	spark-streaming-kafka-0-8	spark-streaming-kafka-0-10
Kafka broker 版本	0.8.2.1或更高	0.10.0或更高
API 稳定性	弃用（Deprecated）	稳定（Stable）
语言支持	Scala、Java、Python	Scala、Java
Receiver DStream	Yes	No
Direct DStream	Yes	Yes
SSL/TLS 支持	No	Yes
Offset 提交 API	No	Yes
动态主题订阅	No	Yes

前面提及本节的内容是基于 Spark 2.3.1 版本的，因此下面的介绍也只基于 spark-streaming- kafka-0-10 版本做相应的陈述，更何况 spark-streaming-kafka-0-8 版本已经被弃用。spark-streaming-kafka-0-10 版本需要的 Maven 依赖如下：

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming-kafka-0-
10_2.11</artifactId>
    <version>2.3.1</version>
</dependency>
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.0.0</version>
</dependency>
```

下面使用一个简单的例子来演示 Spark Streaming 和 Kafka 的集成。在该示例中，每秒往 Kafka 写入一个0～9之间的随机数，通过 Spark Streaming 从 Kafka 中获取数据并实时计算批次间隔内的数据的数值之和。

往 Kafka 中写入随机数的主要代码如下：

```
Random random = new Random();
while (true) {
    String msg =
String.valueOf(random.nextInt(10));
    ProducerRecord<String, String> message =
            new ProducerRecord<>(topic, msg);
    producer.send(message).get();
    TimeUnit.SECONDS.sleep(1);
}
```

Kafka 与 Spark Streaming 的集成示例如代码清单34-2所示，代码中的批次间隔设置为2s。示例中的主题 topic-spark 包含4个分区。

```
//代码清单34-2 Kafka与Spark Streaming的集成示例
```

```
import
org.apache.kafka.clients.consumer.ConsumerConfig
import
org.apache.kafka.common.serialization.StringDeser
ializer
import org.apache.spark.SparkConf
import
org.apache.spark.streaming.kafka010.ConsumerStrat
egies._
import
org.apache.spark.streaming.kafka010.KafkaUtils
import
org.apache.spark.streaming.kafka010.LocationStrat
egies._
import org.apache.spark.streaming.{Seconds,
StreamingContext}

object StreamingWithKafka {
  private val brokers = "localhost:9092"
  private val topic = "topic-spark"
  private val group = "group-spark"
  private val checkpointDir =
"/opt/kafka/checkpoint"

  def main(args: Array[String]): Unit = {
    val sparkConf = new
SparkConf().setMaster("local")
      .setAppName("StreamingWithKafka")①
    val ssc = new StreamingContext(sparkConf,
Seconds(2))②
    ssc.checkpoint(checkpointDir)

    val kafkaParams = Map[String, Object](③
```

```
      ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG ->
brokers,

ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG ->
        classOf[StringDeserializer],

ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG ->
        classOf[StringDeserializer],
      ConsumerConfig.GROUP_ID_CONFIG -> group,
      ConsumerConfig.AUTO_OFFSET_RESET_CONFIG ->
"latest",
      ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG ->
(false:java.lang.Boolean)
    )

    val stream =
KafkaUtils.createDirectStream[String, String](
      ssc, PreferConsistent,
      Subscribe[String, String](List(topic),
kafkaParams))④

    val value = stream.map(record => {⑤
      val intVal =
Integer.valueOf(record.value())
      println(intVal)
      intVal
    }).reduce(_+_)
    value.print()⑥

    ssc.start
    ssc.awaitTermination
  }
}
```

第①和第②行代码在实例化 SparkConf 之后创建了 StreamingContext。创建 StreamingContext 后需要实例化一个 DStream，所以在第④行中通过 KafkaUtils.createDirectStream() 方法创建了一个。第⑤行只是简单地消费读取到的 ConsumerRecord，并执行简单的求和计算。

从 Kafka 中消费数据，这里的 Spark Streaming 本质上是一个消费者，因此 KafkaUtils.createDirectStream() 方法也需要指定 KafkaConsumer 的相关配置。KafkaUtils.createDirectStream() 方法的第一个参数好理解，方法中的第二个参数是 LocationStrategies 类型的，用来指定 Spark 执行器节点上 KafkaConsumer 的分区分配策略。

LocationStrategies 类型提供了3种策略：PerferBrokers 策略，必须保证执行器节点和 Kafka Broker 拥有相同的 host，即两者在相同的机器上，这样可以根据分区副本的 leader 节点来进行分区分配；PerferConsistent 策略，该策略将订阅主题的分区均匀地分配给所有可用的执行器，在绝大多数情况下都使用这种策略，本示例使用的也是这种策略；PerferFixed 策略，允许开发人员指定分区与 host 之间的映射关系。KafkaUtils.createDirectStream() 方法中的第三个参数是 ConsumerStrategies 类型的，用来指定 Spark 执行器节点的消费策略。

与 KafkaConsumer 订阅主题的方式对应，这里也有3种策略：Subscribe、SubscribePattern 和 Assign，分别代表通过指定集合、通过正则表达式和通过指定分区的方式进行订阅。

示例程序最直观的功能就是在每个批次间隔内（2s）读出数据（每秒1个）来进行求和，程序输出的部分结果如下所示。

```
3
4
-------------------------------------------
Time: 1533613594000 ms
-------------------------------------------
7
```

前面提到了执行器有3种消费策略，但是在代码清单34-2中只用到了 Subscribe 策略。如果要使用 SubscribePattern 策略，则可以将代码中的第④行代码修改为如下内容：

```
val stream =
KafkaUtils.createDirectStream[String,String](
  ssc, PreferConsistent,
  SubscribePattern[String,String]
(Pattern.compile("topic-.*"),kafkaParams)
)
```

如果要使用 Assign 策略，则可以将代码中的第④行代码修改为如下内容：

```
val partitions = List(new
TopicPartition(topic,0),
  new TopicPartition(topic,1),
  new TopicPartition(topic,2),
  new TopicPartition(topic,3))
val stream =
KafkaUtils.createDirectStream[String,String](
  ssc, PreferConsistent,
  Assign[String, String](partitions,
kafkaParams))
```

Spark Streaming 也支持从指定的位置处处理数据，前面演示的3种消费策略都可以支持，只需添加对应的参数即可。这里就以 Subscribe 策略为例来演示具体用法，可以用下面的代码替换代码清单34-2中的第④行代码，示例中的 fromOffsets 变量指定了每个分区的起始处理位置为5000：

```
val partitions = List(new
TopicPartition(topic,0),
  new TopicPartition(topic,1),
  new TopicPartition(topic,2),
  new TopicPartition(topic,3))
val fromOffsets = partitions.map(partition => {
  partition -> 5000L
}).toMap
val stream =
KafkaUtils.createDirectStream[String, String](
  ssc, PreferConsistent,
  Subscribe[String, String](List(topic),
kafkaParams, fromOffsets))
```

代码清单34-2中只是计算了批次间隔内的数据，这样只是简单的转换操作，如果需要使用滑动窗口操作，比如计算窗口间隔为20s、滑动间隔为2s的窗口内的数值之和，那么可以将第⑤行代码修改为如下内容：

```
val value = stream.map(record=>{
  Integer.valueOf(record.value())
}).reduceByWindow(_+_, _-
_,Seconds(20),Seconds(2))
```

前面说过在 Direct 方式下，Spark Streaming 会自己控制消费位移的处理，那么原本应该保存到 Kafka 中的消费位移就无法提供准确的信息了。但是在某些情况下，比如监控需求，我们又需要获取当前

Spark Streaming 正在处理的消费位移。Spark Streaming 也考虑到了这种情况，可以通过下面的程序来获取消费位移：

```
stream.foreachRDD(rdd=>{
  val offsetRanges =
rdd.asInstanceOf[HasOffsetRanges].offsetRanges
  rdd.foreachPartition{iter=>
    val o: OffsetRange =
offsetRanges(TaskContext.get.partitionId)
    println(s"${o.topic} ${o.partition}
${o.fromOffset} ${o.untilOffset}")
  }
})
```

注意需要将这段代码放在第④行之后，也就是需要在使用 KafkaUtils.createDirectStream() 方法创建 DStream 之后第一个调用，虽然 Kafka 的分区与 Spark RDD 一一对应，但是在混洗类型的方法（比如 reduceByKey()）执行之后这种对应关系就会丢失。

如果应用更加适合于批处理作业，那么在 Spark 中也可以使用 KafkaUtils.createRDD() 方法创建一个指定处理范围的 RDD。示例参考如下：

```
val offsetRanges = Array(
  OffsetRange(topic,0,0,100),
  OffsetRange(topic,1,0,100),
  OffsetRange(topic,2,0,100),
  OffsetRange(topic,3,0,100)
)
val rdd = KafkaUtils.createRDD(ssc,
  JavaConversions.mapAsJavaMap(kafkaParams),
  offsetRanges, PreferConsistent)
rdd.foreachPartition(records=>{
  records.foreach(record=>{

println(record.topic()+":"+record.partition()+":"
+ record.value())
  })
})
```

示例中的 OffsetRange 类型表示给定主题和分区中特定消息序列的下限和上限。OffsetRange(topic,0,0,100) 这行代码中标识从 topic 主题的第0个分区的偏移量0到偏移量100（不包括）的100条消息。