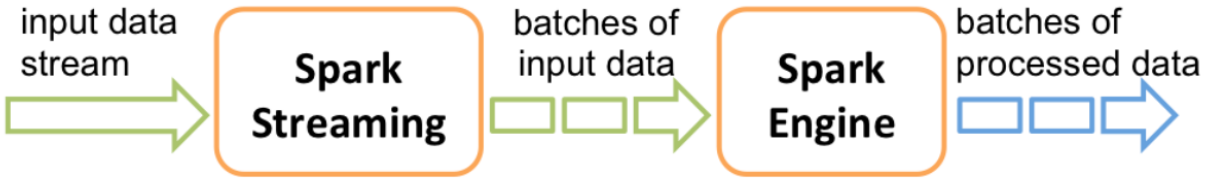
介绍

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like `map`, `reduce`, `join` and `window`. Finally, processed data can be pushed out to filesystems, databases, and live dashboards. In fact, you can apply Spark's machine learning and graph processing algorithms on data streams.

Internally, it works as follows. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.



示例代码

```
import org.apache.spark._
import org.apache.spark.streaming._

// master 是任务提交的地址
val conf = new SparkConf.setAppName(appName).setMaster(master)
val ssc = new StreamingContext(sc, Seconds(1))

```

```
// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)
// Split each line into words
val words = lines.flatMap(_.split(" "))
// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
// Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.print()
ssc.start() // Start the computation
ssc.awaitTermination() // Wait for the computation to terminate
```

构建sparkStreaming步骤：

1. Define the input sources by creating input DStreams.
2. Define the streaming computations by applying transformation and output operations to DStreams.
3. Start receiving data and processing it using `streamingContext.start()`.
4. Wait for the processing to be stopped (manually or due to any error) using `streamingContext.awaitTermination()`.
5. The processing can be manually stopped using `streamingContext.stop()`.

Points to remember

- Once a context has been started, no new streaming computations can be set up or added to it.
- Once a context has been stopped, it cannot be restarted.
- Only one StreamingContext can be active in a JVM at the same time.
- stop() on StreamingContext also stops the SparkContext. To stop only the StreamingContext, set the optional parameter of `stop()` called `stopSparkContext` to false.
- A SparkContext can be re-used to create multiple StreamingContexts, as long as the previous StreamingContext is stopped (without stopping the SparkContext) before the next StreamingContext is created.

Points to remember

- When running a Spark Streaming program locally, do not use “local” or “local[1]” as the master URL. Either of these means that only one thread will be used for running tasks locally. If you are using an input DStream based on a receiver (e.g. sockets, Kafka, Flume, etc.), then the single thread will be used to run the receiver, leaving no thread for processing the received data. Hence, when running locally, always use “local[*n*]” as the master URL, where *n* > number of receivers to run (see Spark Properties for information on how to set the master).

- Extending the logic to running on a cluster, the number of cores allocated to the Spark Streaming application must be more than the number of receivers. Otherwise the system will receive data, but not be able to process it.

Checkpointing

There are two types of data that are checkpointed.

- *Metadata checkpointing* - Saving of the information defining the streaming computation to fault-tolerant storage like HDFS. This is used to recover from failure of the node running the driver of the streaming application (discussed in detail later). Metadata includes:
  - *Configuration* - The configuration that was used to create the streaming application.
  - *DStream operations* - The set of DStream operations that define the streaming application.
  - *Incomplete batches* - Batches whose jobs are queued but have not completed yet.
- *Data checkpointing* - Saving of the generated RDDs to reliable storage. This is necessary in some *stateful* transformations that combine data across multiple batches. In such transformations, the generated RDDs depend on RDDs of previous batches, which causes the length of the dependency chain to keep increasing with time. To avoid such unbounded increases in recovery time (proportional to dependency chain), intermediate RDDs of stateful transformations are periodically *checkpointed* to reliable storage (e.g. HDFS) to cut off the dependency chains.

To summarize, metadata checkpointing is primarily needed for recovery from driver failures, whereas data or RDD checkpointing is necessary even for basic functioning if stateful transformations are used.

When to enable Checkpointing

Checkpointing must be enabled for applications with any of the following requirements:

- *Usage of stateful transformations* - If either `updateStateByKey` or `reduceByKeyAndWindow` (with inverse function) is used in the application, then the checkpoint directory must be provided to allow for periodic RDD checkpointing.
- *Recovering from failures of the driver running the application* - Metadata checkpoints are used to recover with progress information.

Note that simple streaming applications without the aforementioned stateful transformations can be run without enabling checkpointing. The recovery from driver failures will also be partial in that case (some received but unprocessed data may be lost). This is often acceptable and many run Spark Streaming applications in this way. Support for non-Hadoop environments is expected to improve in the future.

Accumulators, Broadcast Variables, and Checkpoints

Accumulators and Broadcast variables cannot be recovered from checkpoint in Spark Streaming. If you enable checkpointing and useAccumulators or Broadcast variables as well, you'll have to create lazily instantiated singleton instances for Accumulators and Broadcast variablesso that they can be re-instantiated after the driver restarts on failure.