Processor API

The Processor API allows developers to define and connect custom processors and to interact with state stores. With the Processor API, you can define arbitrary stream processors that process one received record at a time, and connect these processors with their associated state stores to compose the processor topology that represents a customized processing logic.

Overview

The Processor API can be used to implement both**stateless** as well as **stateful** operations, where the latter is achieved through the use of state stores.

Tip

Combining the DSL and the Processor API: You can combine the convenience of the DSL with the power and flexibility of the Processor API as described in the section Applying processors and transformers (Processor API integration).

For a complete list of available API functionality, see theKafka Streams API docs.

Defining a Stream Processor

A stream processor is a node in the processor topology that represents a single processing step. With the Processor API, you can define arbitrary stream processors that processes one received record at a time, and connect these processors with their associated state stores to compose the processor topology.

You can define a customized stream processor by implementing the `Processor` interface, which provides the `process()` API method. The `process()` method is called on each of the received records.

The `Processor` interface also has an `init()` method, which is called by the Kafka Streams library during task construction phase. Processor instances should perform any required initialization in this method. The `init()` method passes in a `ProcessorContext` instance, which provides access to the metadata of the currently processed record, including its source Apache Kafka® topic and partition, its corresponding message offset, and further such information. You can also use this context instance to schedule a punctuation function (via `ProcessorContext#schedule()`), to forward a new record as a key-value pair to the downstream processors (via `ProcessorContext#forward()`), and to commit the current processing progress (via `ProcessorContext#commit()`).

Specifically, `ProcessorContext#schedule()` accepts a user `Punctuator` callback interface, which triggers its `punctuate()` API method periodically based on the `PunctuationType`. The `PunctuationType` determines what notion of time is used for the punctuation scheduling: either event-time or wall-clock-time (by default, event-time is configured to represent event-time via `TimestampExtractor`). When event-time is used, `punctuate()` is triggered purely by data because event-time is determined (and advanced forward) by the timestamps derived from the input data. When there is no new input data arriving, event-time is not advanced and thus `punctuate()` is not called.

For example, if you schedule a `Punctuator` function every 10 seconds based on `PunctuationType.STREAM_TIME` and if you process a stream of 60 records with consecutive timestamps from 1 (first record) to 60 seconds (last record), then `punctuate()` would be called 6 times. This happens regardless of the time required to actually process those records. `punctuate()` would be called 6 times regardless of whether processing these 60 records takes a second, a minute, or an hour.

When wall-clock-time (i.e. `PunctuationType.WALL_CLOCK_TIME`) is used, `punctuate()` is triggered purely by the wall-clock time. Reusing the example above, if the `Punctuator` function is scheduled based on `PunctuationType.WALL_CLOCK_TIME`, and if these 60 records were p within 20 seconds, `punctuate()` is called 2 times (one time every 10 seconds). If these 60 records were processed within 5 seconds `punctuate()` is called at all. Note that you can schedule multiple `Punctuator` callbacks with different `PunctuationType` types within

Expand Content

processor by calling `ProcessorContext#schedule()` multiple times inside `init()` method.

Attention

Event-time is only advanced if all input partitions over all input topics have new data (with newer timestamps) available. If at least one partition does not have any new data available, event-time will not be advanced and thus `punctuate()` will not be triggered if `PunctuationType.STREAM_TIME` was specified. This behavior is independent of the configured timestamp extractor, i.e., using `WallclockTimestampExtractor` does not enable wall-clock triggering of `punctuate()`.

The following example `Processor` defines a simple word-count algorithm and the following actions are performed:

- In the `init()` method, schedule the punctuation every second (the minimum time unit supported is one millisecond) and retrieve the local state store by its name "Counts".
- In the `process()` method, upon each received record, split the value string into words, and update their counts into the state store (we will talk about this later in this section).

The `Punctuator` object scheduled in `init()` defines a `punctuate()` method, in which we iterate the local state store and send the aggregated counts to the downstream processor (we will talk about downstream processors later in this section), and commit the current stream state.

Attention

The code shows a simplified example that only works for single partition input topics. A generic Processor API word-count would require two processors: the first is stateless, splits each line into words, and sets the words as record keys. The result of the first processor is written back to an additional topic that is consumed by the second (stateful) processor that does the actual counting. Writing the words back to a topic is required to group the same words together so they go to the same instance of the second processor. This is similar in function to the shuffle phase of a Map-Reduce computation.

```java
public class WordCountProcessor implements Processor<String, String> {

  private ProcessorContext context;
  private KeyValueStore<String, Long> kvStore;

  @Override
  @SuppressWarnings("unchecked")
  public void init(ProcessorContext context) {
      // keep the processor context locally because we need it in punctuate() and commit()
      this.context = context;

      // retrieve the key-value store named "Counts"
      kvStore = (KeyValueStore) context.getStateStore("Counts");

      // schedule a punctuate() method every second based on event-time
      this.context.schedule(Duration.ofSeconds(1), PunctuationType.STREAM_TIME, (timestamp) -> {
          KeyValueIterator<String, Long> iter = this.kvStore.all();
          while (iter.hasNext()) {
              KeyValue<String, Long> entry = iter.next();
              context.forward(entry.key, entry.value.toString());
          }
          iter.close();

          // commit the current processing progress
          context.commit();
      });
  }

  @Override
  public void process(String dummy, String line) {
      String[] words = line.toLowerCase(Locale.getDefault()).split(" ");

      for (String word : words) {
          Integer oldValue = this.kvStore.get(word);

          if (oldValue == null) {
              this.kvStore.put(word, 1);
          } else {
              this.kvStore.put(word, oldValue + 1);
          }
      }
  }

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

}
```

Note

Stateful processing with state stores: The `WordCountProcessor` defined above can access the currently received record in its `process()` method, and it can leverage state stores to maintain processing states to, for example, remember recently arrived records for stateful processing needs like aggregations and joins. For more information, see the state stores documentation.

Accessing Processor Context

As we have mentioned above, a `ProcessorContext` controls the processing workflow such as scheduling a punctuation function, and committing the current processed state, etc. In fact, this object can also be used to access the metadata related with the application like `applicationId`, `taskId`, and `stateDir` located to store the task's state, and also the current processed record's metadata like `topic`, `partition`, `offset`, and `timestamp`.

The following example `process()` function enriches the record differently based on the record context:

```
public class EnrichProcessor implements Processor<String, String> {

  private ProcessorContext context;

  @Override
  @SuppressWarnings("unchecked")
  public void init(ProcessorContext context) {
      // keep the processor context locally because we need it in process()
      this.context = context;
  }

  @Override
  public void process(String key, String value) {
      switch(context.topic()) {
          case "alerts":
              context.forward(key, decorateWithHighPriority(value));
          case "notifications":
              context.forward(key, decorateWithMediumPriority(value));
          default:
              context.forward(key, decorateWithLowPriority(value));
      }
  }
}
```

Note

Record context metadata: The metadata of the currently processing record may not always be available. For example, if the current processing record is not piped from any source topic, but is generated from a punctuation function, then its metadata field `topic` will be empty, and `partition` `offset` will be a sentinel value (`-1` in this case), while its `timestamp` field will be the triggering time of the punctuation function that generated this record.

State Stores

To implement a **stateful** `Processor` or `Transformer`, you must provide one or more state stores to the processor or transformer (*stateless* processors or transformers do not need state stores). State stores can be used to remember recently received input records, to track rolling aggregates, to de-duplicate input records, and more. Another feature of state stores is that they can be interactively queried from other applications, such as a NodeJS-based dashboard or a microservice implemented in Scala or Go.

The available state store types in Kafka Streams have fault tolerance enabled by default.

Defining and creating a State Store

You can either use one of the available store types orimplement your own custom store type It's common practice to leverage an existing store type via the `Stores` factory.

Note that, when using Kafka Streams, you normally don't create or instantiate state stores directly in your code. Rather, you define state stores indirectly by creating a so-called `StoreBuilder`. This builder is used by Kafka Streams as a factory to instantiate the actual state stores locally in application instances when and where needed.

The following store types are available out of the box.

Store Type	Storage Engine	Fault-tolerant?	Description

Persistent `KeyValueStore<K, V>`	RocksDB	Yes (enabled by default)	• **The recommended store type for most use cases.**
			• Stores its data on local disk.
			• Storage capacity: managed local state can be larger than the mem space) of an application instance, but must fit into the available lo space.
			• RocksDB settings can be fine-tuned, see RocksDB configuration.
			• Available store variants: time window key-value store, session win value store
			
			`// Creating a persistent key-value store:`
			`// here, we create a `KeyValueStore<String, Long>` named "persiste`
			
			`import org.apache.kafka.streams.processor.StateStoreSupplier;`
			`import org.apache.kafka.streams.state.StoreBuilder;`
			`import org.apache.kafka.streams.state.Stores;`
			
			`StoreBuilder countStoreBuilder =`
			`  Stores.keyValueStoreBuilder(`
			`    Stores.persistentKeyValueStore("persistent-counts"),`
			`    Serdes.String(),`
			`    Serdes.Long()`
			`  );`
			
			See PersistentKeyValueStore for detailed factory options.

In-memory `KeyValueStore<K, V>`	-	Yes (enabled by default)	• Stores its data in memory.
			• Storage capacity: managed local state must fit into memory (heap an application instance.
			• Useful when application instances run in an environment where lo space is either not available or local disk space is wiped in-betwee instance restarts.
			• Available store variants: time window key-value store, session win value store
			
			`// Creating an in-memory key-value store:`
			`// here, we create a `KeyValueStore<String, Long>` named "inmemory`
			`import org.apache.kafka.streams.processor.StateStoreSupplier;`
			`import org.apache.kafka.streams.state.StoreBuilder;`
			`import org.apache.kafka.streams.state.Stores;`
			
			`StateStoreBuilder countStoreBuilder =`
			`  Stores.keyValueStoreBuilder(`
			`    Stores.inMemoryKeyValueStore("inmemory-counts"),`
			`    Serdes.String(),`
			`    Serdes.Long()`
			`  );`
			
			See InMemoryKeyValueStore for detailed factory options.

Fault-tolerant State Stores

To make state stores fault-tolerant and to allow for state store migration without data loss, a state store can be continuously backed up to a Kafka topic behind the scenes. For example, to migrate a stateful stream task from one machine to another when elastically adding or removing capacity from your application. This topic is sometimes referred to as the state store's associated*changelog topic*, or its *changelog*. For example, if you experience machine failure, the state store and the application's state can be fully restored from its changelog. You can enable or disable this backup feature for a state store.

By default, persistent key-value stores are fault-tolerant. They are backed by a compacted changelog topic. The purpose of compacting this topic is to prevent the topic from growing indefinitely, to reduce the storage consumed in the associated Kafka cluster, and to minimize recovery time if a state store needs to be restored from its changelog topic.

Similarly, persistent window stores are fault-tolerant. They are backed by a topic that uses both compaction and deletion. Because of the structure of the message keys that are being sent to the changelog topics, this combination of deletion and compaction is required for the changelog topics of window stores. For window stores, the message keys are composite keys that include the "normal" key and window timestamps. For these types of composite keys it would not be sufficient to only enable compaction to prevent a changelog topic from growing out of bounds. With deletion enabled, old windows that have expired will be cleaned up by Kafka's log cleaner as the log segments expire. The default retention setting is `Materialized#withRetention()` + 1 day. You can override this setting by specifying `StreamsConfig.WINDOW_STORE_CHANGE_LOG_ADDITIONAL_RETENTION_MS_CONFIG` in the `StreamsConfig`.

When you open an `Iterator` from a state store you must call `close()` on the iterator when you are done working with it to reclaim resources; or you can use the iterator from within a try-with-resources statement. If you do not close an iterator, you may encounter an OOM error.

Enable or Disable Fault Tolerance of State Stores (Store Changelogs)

You can enable or disable fault tolerance for a state store by enabling or disabling the change logging of the store through `withLoggingEnabled()` and `withLoggingDisabled()`. You can also fine-tune the associated topic’s configuration if needed.

Example for disabling fault-tolerance:

```
import org.apache.kafka.streams.state.StoreBuilder;
import org.apache.kafka.streams.state.Stores;

StoreBuilder<KeyValueStore<String, Long>> countStoreSupplier = Stores.keyValueStoreBuilder(
  Stores.persistentKeyValueStore("Counts"),
    Serdes.String(),
    Serdes.Long())
  .withLoggingDisabled(); // disable backing up the store to a changelog topic
```

Attention

If the changelog is disabled then the attached state store is no longer fault tolerant and it can't have anystandby replicas.

Here is an example for enabling fault tolerance, with additional changelog-topic configuration: You can add any log config from kafka.log.LogConfig. Unrecognized configs will be ignored.

```
import org.apache.kafka.streams.state.StoreBuilder;
import org.apache.kafka.streams.state.Stores;

Map<String, String> changelogConfig = new HashMap<>();
// override min.insync.replicas
changelogConfig.put("min.insync.replicas", "1");

StoreBuilder<KeyValueStore<String, Long>> countStoreSupplier = Stores.keyValueStoreBuilder(
  Stores.persistentKeyValueStore("Counts"),
    Serdes.String(),
    Serdes.Long())
  .withLoggingEnabled(changelogConfig); // enable changelogging, with custom changelog settings
```

Implementing Custom State Stores

You can use thebuilt-in state store types or implement your own. The primary interface to implement for the store is `org.apache.kafka.streams.processor.StateStore`. Kafka Streams also has a few extended interfaces such as `KeyValueStore`.

You also need to provide a "factory" for the store by implementing the `org.apache.kafka.streams.state.StoreBuilder` interface, which Kafka Streams uses to create instances of your store.

There is an example state store implementation in Scala, which can serve as a starting point for your own stores:

- CMSStore (Scala) -- an in-memory, fault-tolerant store that leverages a Count-Min Sketch data structure for probabilistic counting of items in an input stream. The state store supplier is implemented in CMSStoreSupplier. The backup and restore functionality of the state store and its fault tolerance can be enabled and disabled through configuration. The changelogging of the store is performed through CMSStoreChangeLogger.

Connecting Processors and State Stores

Now that a processor (WordCountProcessor) and the state stores have been defined, you can construct the processor topology by connecting these processors and state stores together by using the `Topology` instance. In addition, you can add source processors with the specified Kafka topics to generate input data streams into the topology, and sink processors with the specified Kafka topics to generate output data streams out of the topology.

Here is an example implementation:

```
Topology builder = new Topology();

// add the source processor node that takes Kafka topic "source-topic" as input
builder.addSource("Source", "source-topic")

    // add the WordCountProcessor node which takes the source processor as its upstream processor
    .addProcessor("Process", () -> new WordCountProcessor(), "Source")

    // add the count store associated with the WordCountProcessor processor
    .addStateStore(countStoreSupplier, "Process")

    // add the sink processor node that takes Kafka topic "sink-topic" as output
    // and the WordCountProcessor node as its upstream processor
    .addSink("Sink", "sink-topic", "Process");
```

Here is a quick explanation of this example:

- A source processor node named `"Source"` is added to the topology using the `addSource` method, with one Kafka topic `"source-topic"` fed to it. You can specify optional key and value deserializers to read the source, like Confluent GenericAvroSerde and SpecificAvroSerde.
- A processor node named `"Process"` with the pre-defined `WordCountProcessor` logic is then added as the downstream processor of the `"Source"` node using the `addProcessor` method.
- A predefined persistent key-value state store is created and associated with the `"Process"` node, using `countStoreSupplier`.
- A sink processor node is then added to complete the topology using the `addSink` method, taking the `"Process"` node as its upstream processor and writing to a separate `"sink-topic"` Kafka topic. Note that users can also use another overloaded variant of `addSink` to dynamically determine the Kafka topic to write to for each received record from the upstream processor.

In this topology, the `"Process"` stream processor node is considered a downstream processor of the `"Source"` node, and an upstream processor of the `"Sink"` node. As a result, whenever the `"Source"` node forwards a newly fetched record from Kafka to its downstream `"Process"` node, the `WordCountProcessor#process()` method is triggered to process the record and update the associated state store. Whenever `context#forward()` is called in the `WordCountProcessor#punctuate()` method, the aggregate key-value pair will be sent via the `"Sink"` processor node to the Kafka topic `"sink-topic"`. Note that in the `WordCountProcessor` implementation, you must refer to the same store name `"Counts"` when accessing the key-value store, otherwise an exception will be thrown at runtime, indicating that the state store cannot be found. If the state store is not associated with the processor in the `Topology` code, accessing it in the processor's `init()` method will also throw an exception at runtime, indicating the state store is not accessible from this processor.

Now that you have fully defined your processor topology in your application, you can proceed to running the Kafka Streams application.

Describing a Topology

After a `Topology` is specified, it is possible to retrieve a description of the corresponding DAG via `#describe()` that returns a `TopologyDescription`. A `TopologyDescription` contains all added source, processor, and sink nodes as well as all attached stores. You can access the specified input and output topic names and patterns for source and sink nodes. For processor nodes, the attached stores are added to the description. Additionally, all nodes have a list to all their connected successor and predecessor nodes. Thus, `TopologyDescription` allows to retrieve the `DAG` structure of the specified topology. Note that global stores are listed explicitly because they are accessible by all nodes without the need to explicitly connect them. Furthermore, nodes are grouped by `SubTopology`, where each `SubTopology` is a group of processor nodes that are directly connected to each other (i.e., either by a direct connection--but not a topic--or by sharing a store). During execution, each `SubTopology` will be processed by one or multiple tasks. Thus, each `SubTopology` describes an independent unit of works that can be executed by different threads in parallel. Describing a `Topology` before starting your streams application with the specified topology is helpful to reason about tasks and thus maximum parallelism (we will talk about how to execute your written application later in this section). It is also helpful to get insight into a `Topology` if it is not specified directly as described above but via Kafka Streams DSL.

Last updated on Sep 10, 2019.