

KIP-478 - Strongly typed Processor API

- Status
- Motivation
- Proposed Changes
- Public Interfaces
 - (deprecation) `org.apache.kafka.streams.processor.{Processor, ProcessorSupplier, ProcessorContext}`
 - (new class) `org.apache.kafka.streams.processor.api.Processor`
 - (new class) `org.apache.kafka.streams.processor.api.ProcessorSupplier`
 - (alter class) `org.apache.kafka.streams.processor.ProcessorContext`
 - (new class) `org.apache.kafka.streams.processor.api.ProcessorContext`
 - (new class) `org.apache.kafka.streams.processor.api.Record`
 - (new class) `org.apache.kafka.streams.processor.api.RecordMetadata`
 - (new class) `org.apache.kafka.streams.processor.StateStoreContext`
 - (deprecation and new method) `org.apache.kafka.streams.processor.StateStore`
 - (new method) `org.apache.kafka.streams.StreamsBuilder`
 - (new method) `org.apache.kafka.streams.Topology`
 - (deprecation and new method) `org.apache.kafka.streams.kstream.KStream.process`
 - (unchanged) `org.apache.kafka.streams.kstream.{Transformer, ValueTransformer, ValueTransformerWithKey}`
 - (new class) (test-utils) `org.apache.kafka.streams.processor.api.MockProcessorContext`
- Compatibility, Deprecation, and Migration Plan
- Rejected Alternatives


Status

Current state: Accepted

Discussion thread: <https://lists.apache.org/thread.html/13f306454761ef7318fb9a658b902fb1663a73e3dde542a2c2b29ab4@%3Cdev.kafka.apache.org%3E>

Vote Thread: <https://lists.apache.org/thread.html/6d3b4f6d286f3f88db3e7f9eebe2f0d361152b84f2021aaf9ba56f8e%40%3Cdev.kafka.apache.org%3E>

JIRA:

 Unable to render Jira issues macro, execution error.

POC: <https://github.com/apache/kafka/pull/6856>

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

The current Processor API is designed with flexibility in mind, but it overlooks an opportunity for type safety that is still compatible with the flexibility objective.

In particular, the Processor interface has ``<K,V>`` generic parameters, but these only bound the *input* types of the processor. This is a very reasonable decision, since the `process` method is actually `void`, and any result from the processing is instead propagated via `ProcessorContext#forward`. In principle, any type can be forwarded, so the context doesn't bound the forward call at all (it's effectively `Object key, Object value`).

However, in practice, most processors will produce keys or values or both of a predictable type, and authors would benefit from the parity-check that type safety provides. This can be achieved by adding generic parameters to `ProcessorContext`, and also adding "output type" bounds to the Processor interface itself. This does *not* restrict the range of use cases at all, though, since heterogeneous-output Processors, which wish to forward variable types, can still bound the "output types" to `<?,?>`.

Some examples of use cases that would benefit from this change:

- Any processor can benefit. Imagine a pure user of the ProcessorAPI who has very complex processing logic. I have seen several processor implementations that are hundreds of lines long and call `context.forward` in many different locations and branches. In such an implementation, it would be very easy to have a bug in a rarely used branch that forwards the wrong kind of value. This would structurally prevent that from happening.
- Also, anyone who heavily uses the ProcessorAPI would likely have developed helper methods to wire together processors, just as we have in the DSL implementation. This change would enable them to ensure at compile time that they are actually wiring together compatible types. This was actually *my* original motivation, since I found it very difficult and time consuming to follow the Streams DSL internal builders.

For completeness, it's worth mentioning that direct and simple usage of the ProcessorAPI would *not* benefit from this KIP. In other words,

- if your processors are short and simple (like you call `context.forward()` once), then your implementation is "obviously correct" and the type safety doesn't buy much.
- if you directly use `Topology.addProcessor(Processor myProcessor,..., String parentProcessorName)`, then we can't enforce that `myProcessor`'s input type is the same as the parent processor's output type (since we only have its name). Adding such safety would be possible as follow-on work after KIP-478 is implemented.

Proposed Changes

The high-level approach is to bound the *output* types of Processor, which we can enforce by bounding the types that ProcessorContext accepts for *forward*.

Fortunately, ProcessorContext does not have any generic parameters, so we can just add a new parameter list `<K, V>` without breaking any code compatibility. Un-parameterized usages will just start to get a "rawtypes" warning.

Unfortunately, Processor already has two generic parameters (the input key and value types), and there is no backward compatible way to add two more (the output key and value types), so we have to create a new Processor interface. The new interface of course needs a different fully qualified name, which can be achieved with:

1. A new class name (like `processor.Processor2` or `processor.TypedProcessor`)
2. A new package name (like `processor.api.Processor` or `processor2.Processor`)

The current interface is `processor.Processor`, for reference. The class name seems clunkier, since even after we deprecate and remove Processor, the new names would continue to be visible in source code. This wouldn't be bad if there were some obvious good name for the new interface, but unfortunately Processor seems like the perfect name.

The package name has the drawback that, in Java, you can't import a package, so any references to the new interfaces would need to be fully qualified as long as they co-exist with the old interfaces in source code. On the plus side, once we ultimately remove the old interface, we can then just `import org.apache.kafka.streams.processor.api.Processor`, and then get back to source code that references only Processor. A relevant precedent would be the `nio` package in the standard library, which is the "new" version of the `io` package.

These changes will help users write safer code in their custom processors, and get a proof from the type system that they're forwarding only the types they think they are.

This change also positions us to make other public API improvements, like KAFKA-8396, to collapse transformer types that become redundant after this KIP.

Public Interfaces

(deprecation) `org.apache.kafka.streams.processor.{Processor, ProcessorSupplier, ProcessorContext}`

- these classes are deprecated, which would be propagated to any public APIs that reference them.

(new class) `org.apache.kafka.streams.processor.api.Processor`

- Similar to `org.apache.kafka.streams.processor.Processor`, but adds output generic type parameters
- Bounds to the forwarding types allowed on the ProcessorContext
- Add `init` and `close` are defaulted to no-op for convenience
- Javadocs are similar to existing Processor interface
- updates the `process` method to use a complex Record type and pass the record metadata to `process` (only when it's defined)

```
public interface Processor<KIn, VIn, KOut, VOut> {
    default void init(ProcessorContext<KOut, VOut> context) {}
    void process(Record<KIn, VIn> record);
    default void close() {}
}
```

(new class) `org.apache.kafka.streams.processor.api.ProcessorSupplier`

- Just a Supplier for the new Processor type

```
public interface ProcessorSupplier<KIn, VIn, KOut, VOut> {
    Processor<KIn, VIn, KOut, VOut> get();
}
```

(alter class) org.apache.kafka.streams.processor.ProcessorContext

- Alter `getStateStore` so that callers will no longer have to cast to the concrete store of their choice (although a cast is still done internally)
 - This change is backward compatible

```
public interface ProcessorContext {

- StateStore getStateStore(final String name);
+ <S extends StateStore> S getStateStore(final String name);

}
```

(new class) org.apache.kafka.streams.processor.api.ProcessorContext

- Copy of `processor.ProcessorContext` with added generic parameters `<K, V>`
 - code snippet below shows how the new API compares to `processor.ProcessorContext`
- Alter `forward` to take `Record` and optional `childName`
- Drop the deprecated members of `processor.ProcessorContext`
- Alter `getStateStore` so that callers will no longer have to cast to the concrete store of their choice (although a cast is still done internally)
- Drop `register(StateStore, StateRestoreCallback)`, which will be moved to `StateStoreContext`
- Drop the "record context" methods, which will be moved to `Record` and `RecordMetadata`

```
public interface ProcessorContext<K, V> {
    ...
- <KForward,          VForward>          void forward(final K key, final V value);
- <KForward,          VForward>          void forward(final K key, final V value, final To to);
+ <K extends KForward, V extends VForward> void forward(Record<K, V> record);
+ <K extends KForward, V extends VForward> void forward(Record<K, V> record, String childName);

- StateStore getStateStore(final String name);
+ <S extends StateStore> S getStateStore(final String name);

- void register(StateStore store, StateRestoreCallback stateRestoreCallback);

- String topic();
- int partition();
- long offset();
- Headers headers();
- long timestamp();
+ Optional<RecordMetadata> recordMetadata();
}
```

(new class) org.apache.kafka.streams.processor.api.Record

- encapsulates all the data attributes of a record for processing: key, value, timestamp, and headers
- can be used both to receive a record for processing in `Processor` and to forward a record downstream in `ProcessorContext`
- includes a constructor for creating a new `Record` from scratch as well as builder-style methods for making a shallow copy of a `Record` with an attribute changed

Record

```
public class Record<K, V> {
    public Record(final K key, final V value, final long timestamp, final Headers headers);
    public Record(final K key, final V value, final long timestamp);

    public K key();
    public V value();
    public long timestamp();
    public Headers headers();

    public <NewK> Record<NewK, V> withKey(final NewK key);
    public <NewV> Record<K, NewV> withValue(final NewV value);
    public Record<K, V> withTimestamp(final long timestamp);
    public Record<K, V> withHeaders(final Headers headers);
}
```

(new class) org.apache.kafka.streams.processor.api.RecordMetadata

- interface that offers a view onto the "record context"
- not settable nor forwardable
- only available when a consumer record is being processed (i.e., it's wrapped with Optional in Processor.process)

```
public interface RecordMetadata {
    String topic();
    int partition();
    long offset();
}
```

(new class) org.apache.kafka.streams.processor.StateStoreContext

- Extraction of only the members of ProcessorContext that need to be provided to state stores (via `StateStore#init`)
- It includes `register(StateStore, StateRestoreCallback)`, which only needs to be called by stores (so it's dropped from the new ProcessorContext)
- It includes all the "general context" members (app id, config, etc), which are all still in ProcessorContext as well
- It does *not* include anything processor- or record- specific, like `forward()` or any information about the "current" record, which is only a well-defined in the context of the Processor. Processors process one record at a time, but state stores may be used to store and fetch many records, so there is no "current record".

StateStoreContext

```
package org.apache.kafka.streams.processor;

import org.apache.kafka.common.serialization.Serde;
import org.apache.kafka.streams.StreamsMetrics;
import org.apache.kafka.streams.errors.StreamsException;

import java.io.File;
import java.util.Map;

/**
 * State store context interface.
 */
public interface StateStoreContext {

    /**
     * Returns the application id.
     *
     * @return the application id
     */
    String applicationId();

    /**
     * Returns the task id.
     */
}
```

```

 *
 * @return the task id
 */
TaskId taskId();

/**
 * Returns the default key serde.
 *
 * @return the key serializer
 */
Serde<?> keySerde();

/**
 * Returns the default value serde.
 *
 * @return the value serializer
 */
Serde<?> valueSerde();

/**
 * Returns the state directory for the partition.
 *
 * @return the state directory
 */
File stateDir();

/**
 * Returns Metrics instance.
 *
 * @return StreamsMetrics
 */
StreamsMetrics metrics();

/**
 * Registers and possibly restores the specified storage engine.
 *
 * @param store the storage engine
 * @param stateRestoreCallback the restoration callback logic for log-backed state stores upon restart
 *
 * @throws IllegalStateException If store gets registered after initialized is already finished
 * @throws StreamsException if the store's change log does not contain the partition
 */
void register(final StateStore store,
              final StateRestoreCallback stateRestoreCallback);

/**
 * Returns all the application config properties as key/value pairs.
 *
 * <p> The config properties are defined in the {@link org.apache.kafka.streams.StreamsConfig}
 * object and associated to the StateStoreContext.
 *
 * <p> The type of the values is dependent on the {@link org.apache.kafka.common.config.ConfigDef.Type
type} of the property
 * (e.g. the value of {@link org.apache.kafka.streams.StreamsConfig#DEFAULT_KEY_SERDE_CLASS_CONFIG
DEFAULT_KEY_SERDE_CLASS_CONFIG}
 * will be of type {@link Class}, even if it was specified as a String to
 * {@link org.apache.kafka.streams.StreamsConfig#StreamsConfig(Map) StreamsConfig(Map)}).
 *
 * @return all the key/values from the StreamsConfig properties
 */
Map<String, Object> appConfigs();

/**
 * Returns all the application config properties with the given key prefix, as key/value pairs
 * stripping the prefix.
 *
 * <p> The config properties are defined in the {@link org.apache.kafka.streams.StreamsConfig}
 * object and associated to the StateStoreContext.
 *
 * @param prefix the properties prefix
 * @return the key/values matching the given prefix from the StreamsConfig properties.

```

```

    */
    Map<String, Object> appConfigsWithPrefix(final String prefix);
}

```

(deprecation and new method) org.apache.kafka.streams.processor.StateStore

- Deprecate the existing `init(ProcessorContext)` method and replace it with `init(StateStoreContext)`
- The new method will have a default implementation that calls the old method, preserving backward compatibility
- In a major-versioned release, we will *delete* the deprecated init method and strip off the `default` keyword from the new method, resulting in a fully compatible transition to the desired end-state in which we only have `init(StateStoreContext)`. This note about the future is informational, we would actually propose this move in a separate KIP.

StateStore

```

@Deprecated
void init(org.apache.kafka.streams.processor.ProcessorContext context, StateStore root);

+ /**
+  * Initializes this state store.
+  * <p>
+  * The implementation of this function must register the root store in the context via the
+  * {@link StateStoreContext#register(StateStore, StateRestoreCallback)} function, where the
+  * first {@link StateStore} parameter should always be the passed-in {@code root} object, and
+  * the second parameter should be an object of user's implementation
+  * of the {@link StateRestoreCallback} interface used for restoring the state store from the changelog.
+  * <p>
+  * Note that if the state store engine itself supports bulk writes, users can implement another
+  * interface {@link BatchingStateRestoreCallback} which extends {@link StateRestoreCallback} to
+  * let users implement bulk-load restoration logic instead of restoring one record at a time.
+  *
+  * @throws IllegalStateException If store gets registered after initialized is already finished
+  * @throws StreamsException if the store's change log does not contain the partition
+  */
+ default void init(final StateStoreContext context, final StateStore root) {
+     // delegate to init(ProcessorContext, StateStore)
+ }

```

(new method) org.apache.kafka.streams.StreamsBuilder

- These changes are fully backward compatible

```

public synchronized <KIn, VIn> StreamsBuilder addGlobalStore(
    final StoreBuilder storeBuilder,
    final String topic,
    final Consumed<KIn, VIn> consumed,
    final processor.api.ProcessorSupplier<KIn, VIn, Void, Void> stateUpdateSupplier
);

```

(new method) org.apache.kafka.streams.Topology

- These changes are fully backward compatible

```

public synchronized <KIn, VIn, KOut, VOut> Topology addProcessor(
    final String name,
    final processor.api.ProcessorSupplier<KIn, VIn, KOut, VOut> supplier,
    final String... parentNames
);

public synchronized <KIn, VIn, KOut, VOut> Topology addGlobalStore(
    final StoreBuilder storeBuilder,
    final String sourceName,
    final Deserializer<KIn> keyDeserializer,
    final Deserializer<VIn> valueDeserializer,
    final String topic,
    final String processorName,
    final processor.api.ProcessorSupplier<KIn, VIn, KOut, VOut> stateUpdateSupplier,
);

public synchronized <KIn, VIn, KOut, VOut> Topology addGlobalStore(
    final StoreBuilder storeBuilder,
    final String sourceName,
    final TimestampExtractor timestampExtractor,
    final Deserializer<KIn> keyDeserializer,
    final Deserializer<VIn> valueDeserializer,
    final String topic,
    final String processorName,
    final processor.api.ProcessorSupplier<KIn, VIn, KOut, VOut> stateUpdateSupplier,
);

```

(deprecation and new method) org.apache.kafka.streams.kstream.KStream.process

Note that this API is a candidate for change in the future as a part of



Unable to render Jira issues macro, execution error.

In the mean time, we will provide a migration path to the new PAPI. Since the KStreams.process currently does not allow forwarding, we will set the KOut and VOut parameters to Void, Void.

```
// DEPRECATIONS:
/*
...
* @deprecated Since 3.0. Use {@link KStream#process(org.apache.kafka.streams.processor.api.ProcessorSupplier,
java.lang.String...)} instead.
*/
@Deprecated
void process(
    org.apache.kafka.streams.processor.ProcessorSupplier<? super K, ? super V> processorSupplier,
    final String... stateStoreNames
);

*/
...
* @deprecated Since 3.0. Use {@link KStream#process(org.apache.kafka.streams.processor.api.ProcessorSupplier,
org.apache.kafka.streams.kstream.Named, java.lang.String...)} instead.
*/
@Deprecated
void process(
    org.apache.kafka.streams.processor.ProcessorSupplier<? super K, ? super V> processorSupplier,
    Named named,
    String... stateStoreNames
);

// NEW METHODS:
void process(
    ProcessorSupplier<? super K, ? super V, Void, Void> processorSupplier,
    String... stateStoreNames
);

void process(
    ProcessorSupplier<? super K, ? super V, Void, Void> processorSupplier,
    Named named,
    String... stateStoreNames
);
```

We will also do the same with the Scala API. Note that we depart from the typical scala-api pattern for suppliers (`()=>Processor`) and take a `ProcessorSupplier`, because otherwise the new and old methods will clash after type erasure.

Also, we are taking the forwarding type as `Void` instead of `Unit` because it is not possible for the scala API implementation to convert a `ProcessorSupplier [K, V, Unit, Unit]` parameter to a `ProcessorSupplier[K, V, Void, Void]` argument to the java API. The only impact of this is that implementers would have to call forward with `forward(null, null)` instead of `forward((),())`. Since the actual intent is for implementers not to call forward at all, this seems like an inconsequential incongruity.

```
// DEPRECATIONS:

@deprecated(since = "3.0", message = "Use process(ProcessorSupplier, String*) instead.")
def process(
    processorSupplier: () => org.apache.kafka.streams.processor.Processor[K, V],
    stateStoreNames: String*
): Unit

@deprecated(since = "3.0", message = "Use process(ProcessorSupplier, String*) instead.")
def process(
    processorSupplier: () => org.apache.kafka.streams.processor.Processor[K, V],
    named: Named,
    stateStoreNames: String*
): Unit

// NEW METHODS
def process(processorSupplier: ProcessorSupplier[K, V, Void, Void], stateStoreNames: String*): Unit

def process(processorSupplier: ProcessorSupplier[K, V, Void, Void], named: Named, stateStoreNames: String*):
Unit
```


(unchanged) org.apache.kafka.streams.kstream.{Transformer, ValueTransformer, ValueTransformerWithKey}

Just explicitly stating that the Transformer interfaces would not be changed at all. The generics case for Transformer is a little more complicated, and I'd like to give it the consideration it really deserves within the scope of <https://issues.apache.org/jira/browse/KAFKA-8396>.

This future work is tracked as



Unable to render Jira issues macro, execution error.

(new class) (test-utils) org.apache.kafka.streams.processor.api.MockProcessorContext

- copy of the current test-util MockProcessorContext to implement the new interface
- most members are identical to the current MockProcessorContext
- add forwarded generic types to match the forward key/value bounds that are now available in the ProcessorContext
- add a new method `StateStoreContext getStateStoreContext()` to get a StateStoreContext view of the MockProcessorContext for use with initializing state stores in user testing code

MockProcessorContext

```
package org.apache.kafka.streams.processor.api;

/**
 * {@link MockProcessorContext} is a mock of {@link ProcessorContext} for users to test their {@link Processor}
 * implementations.
 * <p>
 * The tests for this class (MockProcessorContextTest) include several behavioral
 * tests that serve as example usage.
 * <p>
 * Note that this class does not take any automated actions (such as firing scheduled punctuators).
 * It simply captures any data it witnesses.
 * If you require more automated tests, we recommend wrapping your {@link Processor} in a minimal source-
 * processor-sink
 * {@link Topology} and using the {@link TopologyTestDriver}.
 */
public class MockProcessorContext<KForward, VForward> implements ProcessorContext<KForward, VForward>,
RecordCollector.Supplier {
    /**
     * {@link CapturedPunctuator} holds captured punctuators, along with their scheduling information.
     */
    public static final class CapturedPunctuator {
        public long getIntervalMs();

        public PunctuationType getType();

        public Punctuator getPunctuator();

        public void cancel();

        public boolean cancelled();
    }

    public static final class CapturedForward<KForward, VForward> {
        /**
         * The child this data was forwarded to.
         *
         * @return The child name, or {@code null} if it was broadcast.
         */
        public String childName();

        /**
         * The timestamp attached to the forwarded record.
         */
    }
}
```

```

        * @return A timestamp, or {@code -1} if none was forwarded.
        */
        public long timestamp();

        /**
         * The data forwarded.
         *
         * @return A key/value pair. Not null.
         */
        public KeyValue<KForward, VForward> keyValue();
    }

    /**
     * Create a {@link MockProcessorContext} with dummy {@code config} and {@code taskId} and {@code null}
     {@code stateDir}.
     * Most unit tests using this mock won't need to know the taskId,
     * and most unit tests should be able to get by with the
     * {@link InMemoryKeyValueStore}, so the stateDir won't matter.
     */
    public MockProcessorContext();

    /**
     * Create a {@link MockProcessorContext} with dummy {@code taskId} and {@code null} {@code stateDir}.
     * Most unit tests using this mock won't need to know the taskId,
     * and most unit tests should be able to get by with the
     * {@link InMemoryKeyValueStore}, so the stateDir won't matter.
     *
     * @param config a Properties object, used to configure the context and the processor.
     */
    public MockProcessorContext(final Properties config);

    /**
     * Create a {@link MockProcessorContext} with a specified taskId and null stateDir.
     *
     * @param config a {@link Properties} object, used to configure the context and the processor.
     * @param taskId a {@link TaskId}, which the context makes available via {@link
    MockProcessorContext#taskId()}.
     * @param stateDir a {@link File}, which the context makes available via {@link
    MockProcessorContext#stateDir()}.
     */
    public MockProcessorContext(final Properties config, final TaskId taskId, final File stateDir);

    @Override
    public String applicationId();

    @Override
    public TaskId taskId();

    @Override
    public Map<String, Object> appConfigs();

    @Override
    public Map<String, Object> appConfigsWithPrefix(final String prefix);

    @Override
    public Serde<?> keySerde();

    @Override
    public Serde<?> valueSerde();

    @Override
    public File stateDir();

    @Override
    public StreamsMetrics metrics();

    /**
     * The context exposes these metadata for use in the processor. Normally, they are set by the Kafka Streams
     framework,
     * but for the purpose of driving unit tests, you can set them directly.
     *

```

```

    * @param topic      A topic name
    * @param partition A partition number
    * @param offset     A record offset
    * @param timestamp A record timestamp
    */
    public void setRecordMetadata(final String topic,
                                final int partition,
                                final long offset,
                                final Headers headers,
                                final long timestamp);

    /**
     * The context exposes this metadata for use in the processor. Normally, they are set by the Kafka Streams
     * framework,
     * but for the purpose of driving unit tests, you can set it directly. Setting this attribute doesn't
     * affect the others.
     */
    * @param topic A topic name
    */
    public void setTopic(final String topic);

    /**
     * The context exposes this metadata for use in the processor. Normally, they are set by the Kafka Streams
     * framework,
     * but for the purpose of driving unit tests, you can set it directly. Setting this attribute doesn't
     * affect the others.
     */
    * @param partition A partition number
    */
    public void setPartition(final int partition);

    /**
     * The context exposes this metadata for use in the processor. Normally, they are set by the Kafka Streams
     * framework,
     * but for the purpose of driving unit tests, you can set it directly. Setting this attribute doesn't
     * affect the others.
     */
    * @param offset A record offset
    */
    public void setOffset(final long offset);

    /**
     * The context exposes this metadata for use in the processor. Normally, they are set by the Kafka Streams
     * framework,
     * but for the purpose of driving unit tests, you can set it directly. Setting this attribute doesn't
     * affect the others.
     */
    * @param headers Record headers
    */
    public void setHeaders(final Headers headers);

    /**
     * The context exposes this metadata for use in the processor. Normally, they are set by the Kafka Streams
     * framework,
     * but for the purpose of driving unit tests, you can set it directly. Setting this attribute doesn't
     * affect the others.
     */
    * @param timestamp A record timestamp
    */
    public void setTimestamp(final long timestamp);

    @Override
    public String topic();

    @Override
    public int partition();

    @Override
    public long offset();

    @Override

```

```

public Headers headers();

@Override
public long timestamp();

@SuppressWarnings("unchecked")
@Override
public <S extends StateStore> S getStateStore(final String name);

@Override
public Cancellable schedule(final Duration interval,
                           final PunctuationType type,
                           final Punctuator callback);

/**
 * Get the punctuators scheduled so far. The returned list is not affected by subsequent calls to {@code
schedule(...)}.
 *
 * @return A list of captured punctuators.
 */
public List<CapturedPunctuator> scheduledPunctuators();

@Override
public <K extends KForward, V extends VForward> void forward(final K key, final V value);

@Override
public <K extends KForward, V extends VForward> void forward(final K key, final V value, final To to);

/**
 * Get all the forwarded data this context has observed. The returned list will not be
 * affected by subsequent interactions with the context. The data in the list is in the same order as the
calls to
 * {@code forward(...)}.
 *
 * @return A list of key/value pairs that were previously passed to the context.
 */
public List<CapturedForward<KForward, VForward>> forwarded();

/**
 * Get all the forwarded data this context has observed for a specific child by name.
 * The returned list will not be affected by subsequent interactions with the context.
 * The data in the list is in the same order as the calls to {@code forward(...)}.
 *
 * @param childName The child name to retrieve forwards for
 * @return A list of key/value pairs that were previously passed to the context.
 */
public List<CapturedForward<KForward, VForward>> forwarded(final String childName);

/**
 * Clear the captured forwarded data.
 */
public void resetForwards();

@Override
public void commit();

/**
 * Whether {@link ProcessorContext#commit()} has been called in this context.
 *
 * @return {@code true} iff {@link ProcessorContext#commit()} has been called in this context since
construction or reset.
 */
public boolean committed();

/**
 * Reset the commit capture to {@code false} (whether or not it was previously {@code true}).
 */
public void resetCommit();

@Override
public RecordCollector recordCollector() {

```

```
    // This interface is assumed by state stores that add change-logging.
    // Rather than risk a mysterious ClassCastException during unit tests, throw an explanatory exception.
}

/**
 * Used to get a {@link StateStoreContext} for use with
 * {@link StateStore#init(StateStoreContext, StateStore)}
 * if you need to initialize a store for your tests.
 * @return a {@link StateStoreContext} that delegates to this ProcessorContext.
 */
public StateStoreContext getStateStoreContext();
}
```

Compatibility, Deprecation, and Migration Plan

We'd deprecate the existing Processor class and any public APIs that depend on it. In future major releases, we'll discuss whether it's been deprecated long enough to be removed.

The most impacted usages would be usages of the Processor API itself. Code would need to be migrated to the new interfaces. Since all the methods are the same, this is a simple process of swapping out the package name and then adding the new generic type arguments.

Rejected Alternatives