KIP-988: Streams Standby Update Listener

Status

Current state: *"Adopted"*

Discussion thread: *here*

JIRA: *here*

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

Co-Authored by Eduwer Camacaro.

Motivation

Currently, Kafka Streams allows you to provide a `StateRestoreListener` which executes callbacks upon the start and end of a state restoration, and also after each batch is restored in the restoration. This is useful for monitoring any Active Task that is undergoing restoration.

KIP-869 adds metrics which further improve visibility of Active Task Restorations. However, as of now it is difficult to get real-time updates (via callbacks) which tell you about the state of Standby Tasks on your Kafka Streams instance. This would be most useful from the operational perspective, for example when implementing a rolling restart or a smooth scale down. In these situations, I as an operator of Kafka Streams want to be able to know where each Active and Standby Task lives and also how "caught up" they are so that I know when it's safe to remove or bounce a certain Streams instance.

In additional to the operational use-case described above, knowing when a Standby Task is created and destroyed (either by promotion or migration) would help operators test various configurations of `session.timeout.ms` during rolling upgrade scenarios with regards to how noisy the shuffling of Standby Tasks is.

Public Interfaces

We will add the `StandbyUpdateListener` interface as follows:

```
public interface StandbyUpdateListener {

    enum SuspendReason {
        MIGRATED,
        PROMOTED
    }

    /**
     * A callback that will be invoked after registering the changelogs for each state store in a standby
     * task. It is guaranteed to always be invoked before any records are loaded into the standby store.
     *
     * @param topicPartition the changelog TopicPartition for this standby task
     * @param storeName the name of the store being loaded
     * @param startingOffset the offset from which the standby task begins consuming from the changelog
     */
    void onUpdateStart(final TopicPartition topicPartition,
                       final String storeName,
                       final long startingOffset);

    /**
     * Method called after loading a batch of records. In this case the maximum size of the batch is whatever
     * the value of the MAX_POLL_RECORDS is set to.
     * <n>
     * This method is called after loading each batch and it is advised to keep processing to a minimum.
     * Any heavy processing will block the state updater thread and slow down the rate of standby task
     * loading. Therefore, if you need to do any extended processing or connect to an external service,
     * consider doing so asynchronously.
     *
     * @param topicPartition the TopicPartition containing the values to restore
     * @param storeName the name of the store undergoing restoration
     * @param batchEndOffset batchEndOffset the changelog end offset (inclusive) of the batch that was just
loaded
     * @param batchSize the total number of records in the batch that was just loaded
     * @param currentEndOffset the current end offset of the changelog topic partition.
     */
    void onBatchLoaded(final TopicPartition topicPartition,
                       final String storeName,
                       final TaskId taskId,
                       final long batchEndOffset,
                       final long batchSize,
                       final long currentEndOffset);

    /**
     * This method is called when the corresponding standby task stops updating, for the provided reason.
     * <p>
     * If the task was {@code MIGRATED} to another instance, this callback will be invoked after this
     * state store (and the task itself) are closed (in which case the data will be cleaned up after
     * state.cleanup.delay.ms).
     * If the task was {@code PROMOTED} to an active task, the state store will not be closed, and the
     * callback will be invoked after unregistering it as a standby task but before re-registering it as an
active task
     * and beginning restoration. In other words, this will always called before the corresponding
     * {@link StateRestoreListener#onRestoreStart} call is made.
     *
     * @param topicPartition the TopicPartition containing the values to restore
     * @param storeName the name of the store undergoing restoration
     * @param storeOffset is the offset of the last changelog record that was read and put into the store at
the time
     * of suspension.
     * @param currentEndOffset the current end offset of the changelog topic partition.
     * @param reason is the reason why the standby task was suspended.
     */
    void onUpdateSuspended(final TopicPartition topicPartition,
                           final String storeName,
                           final long storeOffset,
                           final long currentEndOffset,
                           final SuspendReason reason);
}
```

We will add a method to the `KafkaStreams` object:

```
public void setStandbyUpdateListener(StandbyUpdateListener otterStandbyUpdateListener);
```

Proposed Changes

We propose to create a `StandbyUpdateListener` interface, and allow users to supply one to their Kafka Streams Topology via `KafkaStreams#setStandbyUpdateListener(...)` in a manner similar to the `StateRestoreListener`.

Compatibility, Deprecation, and Migration Plan

We are adding a new method without changing any existing API's. Existing users will not need to know about the `StandbyUpdateListener` functionality, and existing code will be unaffected.

Test Plan

This is a small change that can be tested sufficiently via unit tests. We at LittleHorse have implemented a draft PR and will test internally as well.

Rejected Alternatives

Do Nothing

The first alternative is "Do Nothing." In theory, we can use the `KafkaStreams#metrics()` method to get a handle on the Restore Consumer, and look at the lag metrics of that consumer to determine how "caught up" the Standby Task is. This has a few problems:

1. It feels "clunky" to go through JMX metrics within an application to change things that we want to handle in code path. Additionally, the updates are not guaranteed to be as precise, and it is a polling mechanism rather than a push-based (callback) mechanism.
2. We have no easy way of determining why the Standby Task was migrated away to a different instance (whether it was `PROMOTED` to an Active Task, or `MIGRATED` to run on another Streams Instance).

Use StateRestoreListener

The second alternative is to use the `StateRestoreListener` for Standby Tasks as well. However, that quickly falls apart upon further examination because the API of the `StateRestoreListener` is semantically different from the `StandbyUpdateListener`. Most crucially, a State Restoration has a definitive "finish line", which is the last offset of the Changelog TopicPartition at the time that the State Restoration begins. This is because that offset will not increase during restoration, since the Active Task is down (we know this because the Active Task itself is undergoing restoration!). In contrast, with Standby Tasks, the finish line is constantly advancing.