KIP-52: Connector Control APIs

Status

Current state: *Under Discussion*

Discussion thread:

JIRA: Pause/resume: KAFKA-2370 - Getting issue details... STATUS , Restart: KAFKA-3506 - Getting issue details... STATUS

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

Motivation

After a connector is submitted to the Connect framework, users have limited control over its runtime operation. They can change its configuration and they can remove it, but there is no direct way to restart a failed connector or one of its tasks, nor to temporarily suspend processing while an upstream /downstream system is undergoing maintenance. In this KIP, we propose to add several control APIs to address this gap.

Public Interfaces

New APIs

Pause connector

Method: PUT

Path: /connectors/{connector}/pause

Description: This API asynchronously causes the connector and its tasks to suspend processing. If the connector is already paused, this is a no-op. The paused state is persistent, which means that the connector will stay paused even after cluster rebalances or is restarted.

Response Codes: 202 (Accepted) on successful pause initiation or if the command is a no-op, 404 if the connector doesn't exist

Resume Connector

Method: PUT

Path: /connectors/{connector}/resume

Description: This API asynchronously causes the connector and its tasks to resume processing. If the connector is already started, this is a no-op.

Response Codes: 202 (Accepted) on successful resume initiation or if the command is a no-op, 404 if the connector doesn't exist

Restart Connector

Method: POST

Path: /connectors/{connector}/restart

Description: This API synchronously restarts the connector task.

Response Codes: 204 on successful restart, 404 if the connector doesn't exist, and 409 if the task is already being restarted

Query parameter: forward: indicates whether the restart can be forwarded

Restart Task

Method: POST

Path: /connectors/{connector}/tasks/{task}/restart

Description: This API synchronously restarts a specific worker task. This is a no-op if the cluster is rebalancing.

Response Codes: 204 on successful restart, 404 if the connector doesn't exist, and 409 if the task is already being restarted

Query parameter: forward: indicates whether the restart can be forwarded

Proposed Changes

Pause/resume Connector

The pause/resume functionality is designed to address the use case in which the remote system is undergoing maintenance. Rather than having Connect continue to pull records from or push records to the connector, which could result in many reported errors, it would be preferable for an administrator to simply pause the connector until the maintenance has been completed. One of the requirements that follows from this use case is that the paused state must be persistent.

We have considered several alternatives for storing this state (see below), but ultimately we decided to use the Connect compacted config topic. The main reasons are the following:

- We already have a mechanism to ensure consensus among a connector's tasks on the position within the config topic.
- Although we do not do so in this KIP (see explanation below), we may take into account the pause state when assigning partitions for the group, since it would allow a better balance of the currently running tasks.

So using the config topic basically ensures that the information is in the right place to leverage the information in the future. Additionally, we expect that the frequency of calls to the pause/resume API should be the same order of magnitude as config updates themselves, so the risk that the additional load on the config topic will make consensus among the tasks on the current offset more difficult appears to be small.

Currently, the configuration records stored in the config topic use simple delimited strings. For example, connector configs are stored using the key "connector-{name}" (where "{name}" indicates the name of the connector). We propose to follow this convention and use the key "connector-state-{name}." The record value will contain a simple object with a "state" property indicating the state of the connector and will be serialized using the internal value converter (the same as we do for config objects themselves).

Usage: Since this API is asynchronous, users of this endpoint will need to poll the connector's /status endpoint to verify that the expected state transition has completed. Pausing the connector requires all of the connector's tasks to transition and there may be an observable delay between the individual task transitions. Additionally, if the cluster is rebalancing at the time of the command, the transition won't take effect until after the rebalance has completed.

Restart Tasks

The restart API addresses the problem of restarting a failed or defunct connector. Task restarts are one-time commands which either complete or fail at the time of the request, so there is no need to persist them indefinitely. If the cluster is already in the process of rebalancing or if another user has already initiated a restart, then we return an error to the user so that the restart can be tried again. If we later change the rebalance behavior to restart tasks selectively (e.g. if we used a sticky partitioning approach), then we can also alter the behavior of this API to restart while the rebalance is in progress.

Since there is no persistence needed, we propose to use the existing HTTP forwarding mechanism to send the restart to the current worker which is hosting the task. This generally requires two hops: one to the leader since it is the only worker which knows the full task assignments, and one to the worker which hosts the task at the moment. When the restart request is received on the worker hosting the task, it responds to the request and begins the restart. Note that there is a risk of creating a request loop since a rebalance might cause the task to be reassigned before a pending request can be handled, but we can break the loop by including a flag in the request to indicate whether the request is from the current leader. If the request is sent by the leader, then the worker receiving it will not bother forwarding the request back to the leader and instead return a 404, which will be handled by the leader.

Compatibility, Deprecation, and Migration Plan

This change does not affect the behavior of any existing endpoints, so no migration plan should be needed. We treat the absence of a target state stored in the config topic as "started" by default. Hence when a Connect cluster is upgraded, all connectors will begin in the "started" state. If a connector is paused during a rolling upgrade (which will cause the pause state to be written to the config topic), an old version of Connect will receive the newly written state record and attempt to handle it. However, since the key pattern does not conflict with any existing keys, the connector will simply log a warning and ignore the record. After the rolling upgrade has completed, all workers will consume the correct state and transition accordingly.

Obviously any tooling which depends on these APIs will not work on older versions of Connect.

Rejected Alternatives

Alternatives to pause/resume: Currently there is no good option for pausing a connector. The only option would be to first delete the connector to pause it and resubmit it in order to resume it. This is error-prone since the user has to be careful to preserve the current configuration before deletion.

Alternatives to restart: When a task fails, the only way currently to restart it is to trigger a cluster rebalance. The user could achieve this by either submitting a fake configuration update or deleting and resubmitting the failed connector. Both options feel clumsy and error-prone.

Pause Persistence: Kafka Connect stores runtime task states in a separate status topic. We considered using this topic to store the pause states, but rejected it because it would have made it more difficult to leverage this information in the rebalance protocol. In particular, runtime status updates occur at a higher frequency than config updates, so it would be much more challenging achieving consensus among the workers.

Restart Routing: Instead of depending on two hops to route the restart requests, we could distribute the full task assignments to all workers on every rebalance. Then each worker would know exactly where to route the restart request. Unfortunately, this makes the overhead of the rebalance protocol excessive as the number of workers increases (the order of the message size is O(n^2) where n is the number of workers).