Checkpoint Failure process improvement

Vino yang, Tencent

Goals

Currently, the checkpoint's failure handling logic is somewhat confusing (not focused), which makes some functions on existing code passive. This design document primarily describes how to improve checkpoint failure handling logic and make it more clear. Based on this, we introduce a *CheckpointFailureManager*, which makes the checkpoint failure processing more flexible. This mainly comes from the following appeals:

- FLINK-4810[1]: Checkpoint Coordinator should fail ExecutionGraph after "n" unsuccessful checkpoints
- FLINK-10074[3]: Allowable number of checkpoint failure
- FLINK-10724[2]: Refactor failure handling in check point coordinator

Next, we will explain the original implementation and the existing problems.

Prior Art

[A1] The checkpoint trigger and execute response have a large difference in failure handling.

The logic triggered by the checkpoint is mainly in the *triggerCheckpoint* method, which returns a *CheckpointTriggerResult* object that represents the result of the triggerring. The checkpoint response is asynchronous and message-driven, and in the *CheckpointCoordinator* it evolves into a callback mode for some particular processing methods. Since the callback response methods need to deal with various situations, it subdivides more processing submethods and is scattered throughout the *CheckpointCoordinator*. But in the end they all boil down to *PendingCheckpoint#*abortXXX. These methods list below:

- abortExpired
- abortSubsumed
- abortDeclined
- abortError
- abortWithCause
- failUnacknowledgedPendingCheckpointsFor

- discardCheckpoint
- completePendingCheckpoint
- …...

[A2] The fault-tolerant logic for the checkpoint snapshot failure is on the TM side.

FLINK-4809[4] introduces a *setFailOnCheckpointingErrors* API, which is passed to *ExecutionConfig* and brought to the TM. When the snapshot of the task fails, according to this configuration either an exception is thrown and the entire job fails or only a decline message is reported. This design moves the responsibility for failure handling from the central point on JM side (*CheckpointCoordinator*) to the TM side. It causes trouble for the scalability of the checkpoint's fault-tolerance.

Design

Before explaining a clear design, let's clearly define the two parts of the checkpoint:

1. Trigger: periodic or on-demand triggering of state snapshotting in tasks by *CheckpointCoordinator* (pending checkpoint)
2. Execute: asynchronous state snapshotting in tasks on TM side, *CheckpointCoordinator* collects the results from all tasks to complete or fail the pending checkpoint.

The design below includes these three parts.

[T1] Refactoring checkpoint failure handling.

Before we introduce the checkpoint failure manager, we need to refactor the *CheckpointCoordinator* and reorganize the failure handling. Flink has implemented work in the checkpoint trigger section. But the some design is incompleted and there is still some work to be done for checkpoint execution stage. In this section, the work would be split into two steps.

[T1-1] Redefine failure processing specific classes.

We should refactor these classes:

- Remove the exists *CheckpointTriggerResult*;
- Unifiy the exists *CheckpointTriggerException* and *CheckpointException*;
- Rename the exists *CheckpointDeclineReason* to *CheckpointFailureReason* and add more failure enum values;

To explain the design, we could split the checkpoint’s full behavior into two stages:

- Trigger stage
- Execution stage

For trigger stage, we can use *PendingCheckpoint* to represent a successful case and use *CheckpointException* which carries the *CheckpointFailureReason* to represent an exceptional case.

For execution stage, we can use CompletedCheckpoint to represent a successful case and use *CheckpointException* which carries the *CheckpointFailureReason* to represent an exceptional case.

More detail about the CheckpointException and CheckpointFailureReason, please see "[T1-2]" step 3.

[T1-2] Refactor based on these Redefined classes

Step 1: remove *CheckpointTriggerResult*. There is a key method : *CheckpointCoordinator#triggerCheckpoint* returns an instance of this type. After removing this class, we can refactor this method with this signature:

```
public PendingCheckpoint triggerCheckpoint(long timestamp...) throws CheckpointException {
   //....
}
```

Step 2: merge *CheckpointDeclineReason* into *CheckpointFailureReason*

The *CheckpointDeclineReason* has defined some enum values to represent those declining case. However, to cover more cases for the whole checkpoint lifecycle, we should introduce more enum values for the new *CheckpointFailureReason*, they are listed below:

For now, it has the following enumeration values:

- CheckpointExpired("Checkpoint expired before completing")
- CheckpointSubsumed("Checkpoint has been subsumed")
- CheckpointDeclined("Checkpoint was declined (tasks not ready)")
- CheckpointCoordinatorShutdown("CheckpointCoordinator shutdown")
- CheckpointCoordinatorSuspend("Checkpoint Coordinator is suspending")
- JobFailure("The job has failed.")
- TaskCheckpointFailure("Task local checkpoint failure.")
- FinalizeCheckpointFailure("Failure to finalize checkpoint.")
- TriggerCheckpointFailure("Trigger checkpoint failure")

Step 3: introduce *CheckpointFailureReason* into *CheckpointExecption*, the constructors could be :

```
public CheckpointException(CheckpointFailureReason reason) {

}

public CheckpointException(CheckpointFailureReason reason, Throwable throwable) {

}
```

It still also provides a getter for *CheckpointFailureReason*. The original places where used *CheckpointTriggerException* would use *CheckpointExecption#getCheckpointFailureReason* to replace.

[T1-3] Refactor PendingCheckpoint#abortXXX.

There are a lot of ways to start with abort in *PendingCheckpoint*. Since we have *CheckpointAbortReason*, we can refactor it. The reconstructed signature is as follows:

```
public void abort(CheckpointFailureReason reason, Throwable cause) {
    try {
        onCompletionPromise.completeExceptionally(new CheckpointException(reason, cause));
        reportFailedCheckpoint(cause);
        assertAbortSubsumedForced(reason);
    } finally {
        dispose(true);
    }
}
```

When we provide this method, the original abortXXX methods can be removed.This will unify the processing behavior of abort.

[T2] Introduce CheckpointFailureManager.

Clarification: The term “failure” here refers to both “trigger failure” and “failure during execution”.

The purpose of introducing *CheckpointFailureManager* is to separate the checkpoint failure processing from the *CheckpointCoordinator*. It mainly manages the behavior of checkpoint failures and makes decisions. There are two main decisions at present:

- Tolerate checkpoint failures and let the job run normally;
- Do not tolerate checkpoint failure and trigger job failure;

The reference factor for decision making depends on two configurations:

- CheckpointConfig#failOnCheckpointingErrors (present)
- CheckpointConfig#tolerableFailureNumber (newly introduced)

Next, we'll show you how to make decisions for checkpoint failures.

[T2-1] Introduce an atomic failure counter.

In order to satisfy logic such as "Allow checkpoint failure consecutively n times", we need to count the number of consecutive checkpoint failures.

Ideally there are 2 public methods that will operate the counter:

- handleCheckpointException: counter +1 if failure
- handleCheckpointSuccess: counter reset to 0 if completed

After the counter is changed, it will be compared with the number of tolerated failures. When the maximum threshold of tolerance is reached, the configured job failure handler is called.

[T2-2] How to react on specific failure reasons?

It needs to be affirmed here. The failure of our concern is mainly the failure of the checkpoint "**self**". Of course, it is worth discussing.

For *CheckpointFailureReason*:

- PERIODIC_SCHEDULER_SHUTDOWN: ignore
- ALREADY_QUEUED: ignore
- TOO_MANY_CONCURRENT_CHECKPOINTS: ignore
- MINIMUM_TIME_BETWEEN_CHECKPOINTS: ignore
- NOT_ALL_REQUIRED_TASKS_RUNNING: ignore
- EXCEPTION: counter +1
- CheckpointExpired: ignore
- CheckpointSubsumed: ignore
- CheckpointDeclined: counter +1
- CheckpointCoordinatorShutdown: ignore
- CheckpointCoordinatorSuspend: ignore
- CheckpointUncertainError: counter +1

[T2-3] Add a callback to checkpoint failure handling.

In “Design[T1]” we have refactored the *CheckpointCoordinator*, which makes the logic of failure processing become more focused from being scattered in multiple places.

So we only need to add a callback to *CheckpointFailureManager* for the *handleCheckpointException* method.

So, the pseudo code of the *handleCheckpointResult* method (called after *triggerCheckpoint* in *triggerSavepoint* and periodically) is now like:

```
private CompletableFuture<CompletedCheckpoint> handleCheckpoint(PendingCheckpoint checkpoint,
CheckpointException exception) {

    If (exception != null) {
       checkpointFailureManager.handleCheckpointException(exception);
   } else {
     return checkpoint.getCompletionFuture();
   }
}
```

Next, let's talk about the test.

[T3] Refactor failure handling with CheckpointFailureManager

[T3-1] Let CheckpointCoordinator decide whether to tolerate checkpoint failure of task

As described in "Prior Art [A2]", some of the current *CheckpointCoordinator* responsibilities are separated into the TM side. This design will cause the *CheckpointCoordinator* to lose some control over the checkpoint. We need to refactor it.

The general steps are as follows:

1. CheckpointConfig#failOnCheckpointingErrors should be passed to the *ExecutionGraph* instead of passing it to the Task via ExecutionConfig.(After we proceeded to the T2 step and started to introduce the CheckpointFailureManager. We will create a CheckpointFailureManager first in the ExecutionGraph. ExecutionGraph would tell to CheckpointFailureManager: CheckpointConfig#failOnCheckpointingErrors and how to fail.)
2. Only retain the logic of DecliningCheckpointExceptionHandler and send a DeclineCheckpoint message to JM
3. CheckpointCoordinator determines whether to tolerate failure based on DeclineCheckpoint and failOnCheckpointingErrors

The advantage is that the *CheckpointCoordinator* will gain unique control and will facilitate the failure management of the checkpoint.

[T3-2] Improve checkpoint failure processing combine CheckpointFailureManager

Since we have introduced *CheckpointFailureManager* in step [T2]. Now we could do the final refactor for checkpoint failure processing.

Test Considerations

Test Cases

Based on the design description, the implementation is mainly divided into two parts:

- Refactor existing failure handling logic
- Introducing a CheckpointFailureManager

For the first part, most of our code changes did not involve the public interface. At present, the various scenarios for *CheckpointCoordinator* have been thoroughly tested. We can verify the correctness of the refactoring based on the existing tests. But maybe we need to make changes to existing tests, such as *CheckpointExceptionHandler*.

For the second part, we need to test the logic introduced in the CheckpointFailureManager independently, and we need to write the integration test in conjunction with the *CheckpointCoordinator*.

Challenges

Future

References

[1]: https://issues.apache.org/jira/browse/FLINK-4810
[2]: https://issues.apache.org/jira/browse/FLINK-10724
[3]: https://issues.apache.org/jira/browse/FLINK-10074
[4]: https://issues.apache.org/jira/browse/FLINK-4809
[5]: https://github.com/apache/flink/pull/6567
[6]: https://issues.apache.org/jira/browse/FLINK-10724?focusedCommentId=16715194&page=com.atlassian.jira.plugin.system.issuetabpanels%3Acomment-tabpanel#comment-16715194

Change Log

2018-12-15	Vino Yang (tencent)	Draft 1
2018-12-27	Vino Yang (tencent)	Draft 2 - based on suggestion from Andrey

2018-12-29	Vino Yang(tencent)	Draft 3 - based on suggestion from Andrey
2019-01-08	Vino Yang(tencent)	Draft 3 - based on suggestion from Andrey
2019-04-16	Vino Yang(tencent)	Draft 4 - based on suggestion from Stefan and Andrey
2019-04-18	Vino Yang(tencent)	Draft 5 - based on suggestion from Stefan