[DISCUSS] Lifecycle of ShuffleMaster and its Relationship with JobMaster and PartitionTracker

Lifecycle of ShuffleMaster

Currently, the lifecycle of *ShuffleMaster* seems unclear. The *ShuffleServiceFactory* is loaded for each *JobMaster* instance and then *ShuffleServiceFactory#createShuffleMaster* will be called to create a *ShuffleMaser* instance. However, the default *NettyShuffleServiceFactory* always returns the same *ShuffleMaser* singleton instance for all jobs. Based on the current implementation, the lifecycle of *ShuffleMaster* seems open and depends on the shuffle plugin themselves. However, at the *TM* side, the *ShuffleEnvironment* is a part of the *TaskManagerServices* whose lifecycle is decoupled with jobs which is more like a service. It means there is also an inconsistency between the *TM* side and the *JM* side.

From my understanding, the reason for this is that the pluggable shuffle framework is still not completely finished yet, for example, there is a follow up umbrella ticket FLINK-19551 for the pluggable shuffle service framework and in its subtasks, there is one task (FLINK-12731) which aims to load shuffle plugin with the *PluginManager*. I think this can solve the issue mentioned above. After the corresponding factory loaded by the *PluginManager*, all *ShuffleMaster* instances can be stored in a map indexed by the corresponding factory class name which can be shared by all jobs. After that, the *ShuffleMaster* becomes a cluster level service which is consistent with the *ShuffleEnvironment* at the *TM* side.

As a summary, we propose to finish FLINK-12731 and make the shuffle service a real cluster level service first. Furthermore, we add two lifecycle methods to the *ShuffleMaster* interface, including *start* and *close* responding for initialization (for example, contacting the external system) and graceful shutdown (for example, releasing the resources) respectively (these methods already exist in the *ShuffleEnvironment* interface at the *TM* side). What do you think?

Relationship of ShuffleMaster & JobMaster

Currently, *JobMaster* holds the reference to the corresponding *ShuffleMaster* and it can register partitions (allocate *ShuffleDescriptor* from) to *ShuffleMaster* by the *registerPartitionWithProducer* method. To support use cases like allocating external resources when a job starts and releasing all allocated resources when a job terminates, we may also need some job level initialization and finalization. These job level initialization and finalization are also helpful when serving multiple jobs simultaneously.

As a summary, we propose to add two job level lifecycle methods *registerJob* and *unregisterJob* responding for job level shuffle initialization and finalization, for example, releasing all external resources occupied by the corresponding job. What do you think?

Relationship of ShuffleMaster & PartitionTracker

Currently, the *JobMasterPartitionTracker* can release external result partitions through the *releasePartitionExternally* method of *ShuffleMaster*. However, the shuffle plugin (*ShuffleMaster*) may also need the ability of stopping tracking some partitions depending on the status of the external services, for example, if the external storage node which stores some partitions crashes, we need to stop tracking all partitions in it to avoid reproducing the lost partitions one by one. By introducing something like *ShuffleContext* which delegates to the partition tracker, this requirement can be easily satisfied. Besides, for cluster partitions, we also need to have the ability to release them.

As a summary, we propose to add a releaseDataSetExternally method to the *ShuffleMaster* interface which is responsible for releasing cluster partitions. Besides, we propose to add a *ShuffleContext* which can delegate to the *PartitionTracker* and stop tracking partitions. For the cluster partitions and job partitions, two separated *ShuffleContext* abstracts are needed. What do you think?

Interface Change Summary

As discussed in the above sections, we propose to make some interface changes around the *ShuffleMaster* interface. The first change is to pass a *ShuffleMasterContex* instance to the *ShuffleServiceFactory* when creating the *ShuffleMaster* just like the *ShuffleEnvironment* creation at the *TM* side. Changes are marked with bold texts (the same below).

```
public interface ShuffleServiceFactory<
        SD extends ShuffleDescriptor, P extends ResultPartitionWriter, G extends
IndexedInputGate> {

    /**
     * Factory method to create a specific {@link ShuffleMaster} implementation.
     */
    ShuffleMaster<SD> createShuffleMaster(ShuffleMasterContext
shuffleMasterContext);

    /**
     * Factory method to create a specific local {@link ShuffleEnvironment}
implementation.
     */
    ShuffleEnvironment<P, G> createShuffleEnvironment(
            ShuffleEnvironmentContext shuffleEnvironmentContext);
}
```

The following is the *ShuffleMasterContext* interface. It will be implemented by the pluggable shuffle framework itself and can be used by the shuffle plugin. A context Interface is more friendly if we want to extend it in the future.

```
public interface ShuffleMasterContext {

    /** Gets the cluster configuration. */
    Configuration getConfiguration();

    /** Handles the fatal error if any. */
    void onFatalError(Throwable throwable);

    /**
     * Stops tracking the target dataset (cluster partitions), which means these
data can not be reused anymore.
     */
    CompletableFuture<Void> stopTrackingDataSet(IntermediateDataSetID
dataSetID);

    /** Returns IDs of all datasets (cluster partitions) being tracked by this
cluster currently. */
    CompletableFuture<List<IntermediateDataSetID>> listDataSets();
}
```

The second part to be enhanced is the *ShuffleMaster* interface. Methods to be added include *start*, *close*, *registerJob*, *unregisterJob* and *releaseDataSetExternally*. In addition, because each *ShuffleMaster* instance can serve multiple jobs simultaneously, when registering partitions, one should also provide the corresponding JobID. The following shows the updated *ShuffleMaster* interface:

```
public interface ShuffleMaster<T extends ShuffleDescriptor> extends
AutoCloseable {

    /**
     * Starts this shuffle master, for example getting the access and connecting
to the external
     * system.
     */
    void start() throws Exception;

    /** Closes this shuffle master which releases all resources. */
    void close() throws Exception;

    /** Registers the target job to this shuffle master. */
    void registerJob(JobShuffleContext context);

    /** Unregisters the target job from this shuffle master. */
    void unregisterJob(JobID jobID);

    /** Asynchronously register a partition and its producer with the shuffle
service. */
    CompletableFuture<T> registerPartitionWithProducer(
            JobID jobID,
```

```
            PartitionDescriptor partitionDescriptor,
            ProducerDescriptor producerDescriptor);

    /** Releases any external resources occupied by the given partition. */
    void releasePartitionExternally(ShuffleDescriptor shuffleDescriptor);

    /** Releases the target cluster partitions stored externally if any. */
    void releaseDataSetExternally(IntermediateDataSetID dataSetID);
}
```

The following is the *JobShuffleContext* interface. It will be implemented by the pluggable shuffle framework itself and can be used by the shuffle plugin.

```
public interface JobShuffleContext {

    /** Gets the corresponding job configuration. */
    Configuration getConfiguration();

    /** Gets the corresponding {@link JobID}. */
    JobID getJobID();

    /**
     * Stops tracking the target result partitions, which means these partitions
will be reproduced if used afterwards.
     */
    CompletableFuture<Void> stopTrackingPartitions(Collection<ResultPartitionID>
partitionIDS);

    /** Returns information of all partitions being tracked for the current job.
*/
    CompletableFuture<List<ResultPartitionDeploymentDescriptor>>
listPartitions();
}
```

What do you think of these changes? Any feedback is highly appreciated.