Rethink Flink Client API

Motivation

Flink jobs can be deployed in different cluster setups (local, standalone, YARN, Mesos, and K8s) with different modes (detached/blocking, session/per-job). But now users can only deploy jobs in distributed clusters with Flink CLI and the implementation of cluster deployment and job submission diverges in different settings:

* Local: Currently, this is the only mode in which users can execute their jobs without the involvement of Flink CLI. When deployed locally, jobs can only be executed in the blocking and per-job mode. ExecutionEnvironment will compile the JobGraph, create a MiniCluster, and submit the compiled JobGraph in the MiniCluster.

* Detached & Per-Job: In the cases where the job is submitted in the detached and per-job mode, CliFrontend will obtain the JobGraph by calling user's main method with OptimizerPlanEnvironment. When user's main method aborts on env.execute(), CliFrontend will create a cluster with obtained JobGraph. There is no explicit job submission in this case as JobGraph is coupled with the cluster deployment.

* Detached & Session: When deployed in the detached and session mode, CliFrontend will first retrieve existing cluster or deploy a new cluster. Then it will obtain the JobGraph by calling user's main method with DetachedEnvironment. When user's main method returns, CliFrontend will submit the JobGraph to the cluster.

* Blocking & Session, Blocking & Per-Job: In the two cases, CliFrontend first retrieves existing cluster or deploys a new cluster. Then it will call user's main method with ContextEnvironment. In the user's main method, ContextEnvironment will obtain the JobGraph and submit it to the cluster.

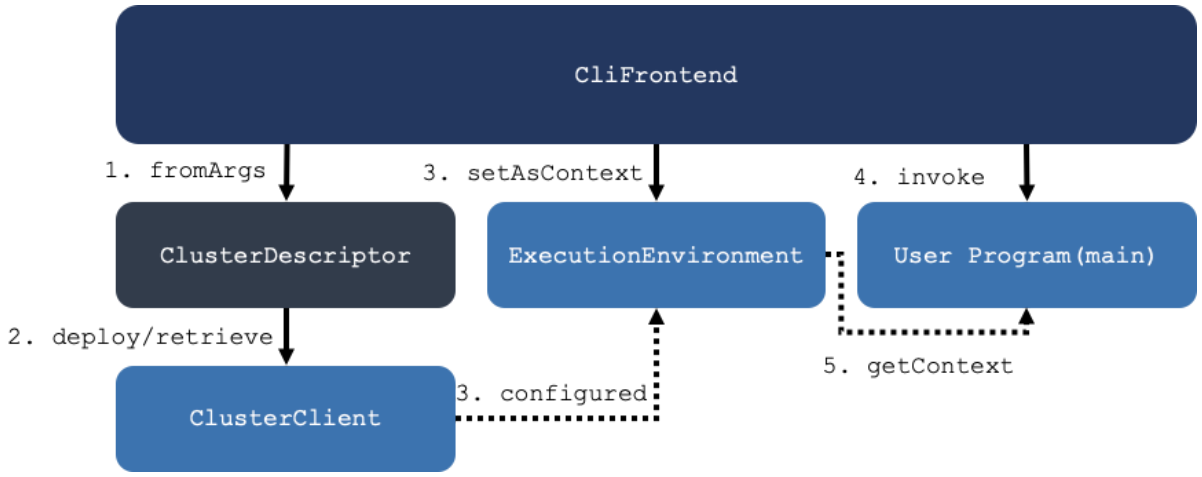
In this document, we propose to 1) unify the implementation of cluster deployment and job submission in Flink and 2) provide programmatic interfaces to allow flexible job and cluster management. When deploying a Flink job with command line:

1. CliFrontend first parses the parameters in the command line. It will deploy or retrieve the working cluster according to the parameters. It then creates a ClusterClient, puts it in the ExecutionEnvironment, sets the env as context, and calls user's main method. CliFrontend will not be responsible for job submission, hence no need to use DetachedEnvironment to obtain JobGraph.

2. ExecutionEnvironment is responsible for job compilation and submission. It will generate the JobGraph and submit it to the cluster with the ClusterClient set by CliFrontend. A completable future which contains a JobClient will be returned when the job is submitted. The future is completed when the job is successfully submitted and users can use returned JobClient to issue commands (e.g., savepoint, cancel) to the job and retrieve the result. Users do not need to distinguish between detached or blocking mode when deploying jobs. They can retrieve results with JobClient as they need.

Unified Implementation

We notice that although at the moment different execute modes have different codepaths, all of these modes fits in a unified process of job execution. That is, cluster deployment, job compilation and job submission. Thus we propose a unified view of job and cluster management as below.



CliFrontend parses the parameters in the command line and generates ClusterDescriptor as current CliFrontend does. ClusterDescriptor deploys or retrieves the working cluster according to the parameters. And then ClusterDescriptor creates a ClusterClient. CliFrontend configures the client in the ExecutionEnvironment, sets the env as context, and calls user's main method. Different mode results in different cluster. All cases are listed below.

1. In local mode[1], i.e., execute main method of user program directly, there is no context env, an env with default local mode configuration would be created and

[1] The phrase “local” is quite confusing because we removed local mode script in FLINK-6487 and “local” mode actually refers launching MiniCluster instead of launching standalone session locally.

used. A MiniCluster is launched by MiniClusterDescriptor, and the corresponding MiniClusterClient is configured in the context env.

2. In session mode,

2-1. In standalone session mode, a StandaloneClusterDescriptor is created. If the configured cluster is deployed previously, such as, by running jobmanager.sh and taskmanager.sh on machines, ClusterDescriptor retrieves the client from the existing session cluster. Otherwise ClusterDescriptor deploys the session cluster and gets the client as the return value. The client is configured in the context env.

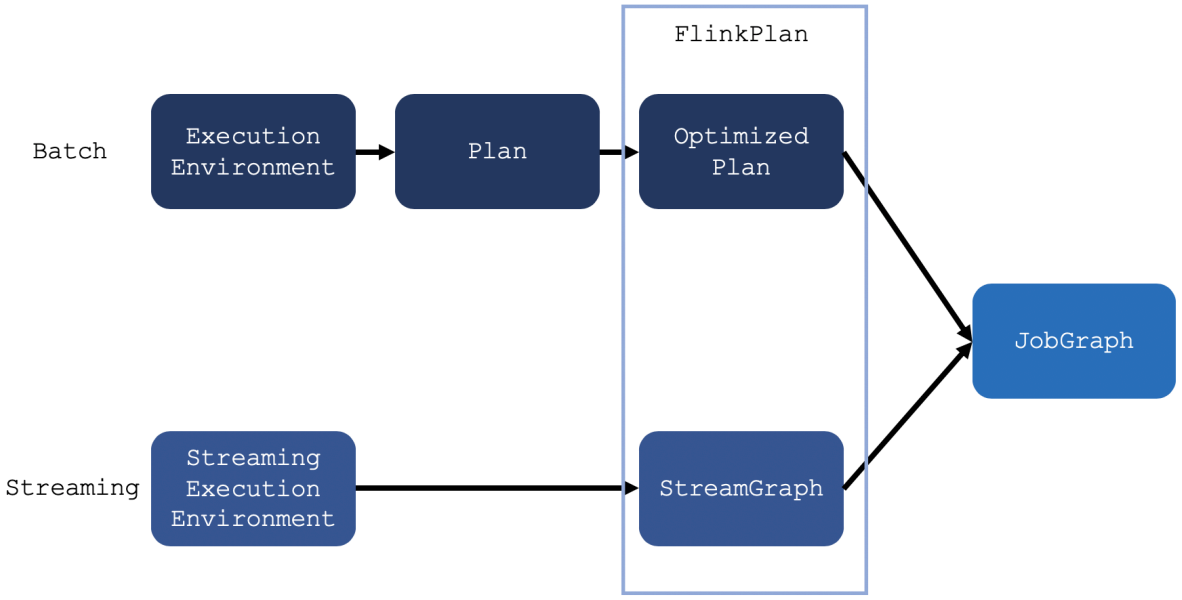
2-2. In mesos session mode, a MesosClusterDescriptor is created. If the configured cluster is deployed previously, such as, by running mesos-appmaster.sh, ClusterDescriptor retrieves the client from the existing session cluster. Otherwise ClusterDescriptor deploys the session cluster and gets the client as the return value. The client is configured in the context env.

2-3. In YARN session mode, a YarnClusterDescriptor is created. If the configured cluster is deployed previously, ClusterDescriptor retrieves the client from the existing session cluster. Otherwise ClusterDescriptor deploys the session cluster and gets the client as the return value. The client is configured in the context env.

3. In job mode, specially, standalone job mode, mesos job mode and YARN job mode, different from the existing process, we propose a similar process with session mode on cluster deployment, except when creating Dispatcher from configuration, in job mode, a MiniDispatcher is used which would exist once one job finishes. The rest such as cluster deployment, job compilation and job submission is completed and individual. This is quite different from current process which couples the JobGraph with cluster deployment and recover the JobGraph for execution instead of job submission.
(* An alternative is to remain job cluster as current but exclude the action “deploy job cluster” from client. Launching job cluster is always from entrypoint main method. *)

With context env set, CliFrontend invokes user’s main method and passes the parameters. In user’s main method, user gets env from context and defines its program. After that, env.execute() is called for job submission. Or for advanced users, a FlinkPlan or JobGraph would be retrieved from env and submitted by ClusterClient which is also retrieved from env.

In either way there is a stage that compiling a Flink program into FlinkPlan or JobGraph. Currently batch job and streaming job use different compilation path. See the graph below.



Class Plan is for legacy batch mode compatibility and the unified interface is FlinkPlan. For preview and job config modification, we propose exposing interfaces to get FlinkPlan(env.compile()) and JobGraph(env.optimize()).

Job submission now always fits in pattern client.submitJob(JobGraph). Especially, in job mode, we also deploy a cluster at first and then submit the JobGraph via cluster client.

A completable future which contains a JobClient will be returned when the job is submitted. The future is completed when the job is successfully submitted. JobClient is an encapsulation of ClusterClient which is associated with a specific job. Users can use returned JobClient to issue commands (e.g., savepoint, cancel) to the job and retrieve the result. The process is as follows.

```
CompletableFuture<JobClient> jobClientFuture =
    clusterClient.submitJob(jobGraph);

// this job client is associated with the job
// whose job id equals to jobGraph.getJobId()
JobClient jobClient = jobClientFuture.get();

CompletableFuture<JobResult> jobResultFuture =
    jobClient.getJobResult();

// wait for job result
JobResult jobResult = jobResultFuture.get();
```

Note that with the distinction between job submission and job execution, users do not need to specify detached or blocking mode when deploying jobs. They can retrieve results with JobClient as they need.

The whole process with the unified implementation is as above. For downstream project developer who want to make accurate control of cluster deployment, context environment settings and the overall process, he could implement his own CliFrontend. See also “Examples#Customize CliFrontend”.

Programmatic Interfaces

Providing programmatic interfaces for cluster description, job compilation and job submission helps advanced Flink users such as developers of Apache Zeppelin and Apache Beam integrate Flink with their customized submission process.

ClusterDescriptor

ClusterDescriptor is responsible for cluster management, that is, deploying a new cluster, retrieving an existing cluster, killing an existing cluster, and retrieving the cluster client. Common interfaces of ClusterDescriptor are listed below.

```
<init>(configuration)
retrieveCluster(clusterId): ClusterClient
deployCluster(clusterSpec): ClusterClient
killCluster(clusterId)
```

ClusterDescriptor varies between different resource management(local, standalone, YARN, Mesos, and K8s), and different descriptors might provides their specific interfaces, e.g, request specific resource management status.

ClusterClient

Different from ClusterDescriptor, ClusterClient serves same interfaces in any resource management. ClusterClient is responsible for job management, including

```
submitJob(JobGraph): CompletableFuture<SubmissionResult>
listJobs(): CompletableFuture<List<JobStatus>>
disposeSavepoint(savepoint path): CompletableFuture<Ack>
listTaskManagers(): CompletableFuture<List<TaskManagerStatus>>

getJobResult(JobID): CompletableFuture<JobResult>
```

```
getJobStatus(JobID): CompletableFuture<JobStatus>
cancel(JobID)
cancelWithSavepoint(JobID): savepoint path
stopWithSavepoint(JobID): savepoint path
triggerSavepoint(JobID): CompletableFuture<savepoint path>
getAccumulators(JobID): Map<acc name, acc value>
```

JobClient

A JobClient is an encapsulation of ClusterClient which is associated with a specific job.

```
getJobResult(): CompletableFuture<JobResult>
getJobStatus(): CompletableFuture<JobStatus>
cancel()
cancelWithSavepoint(): savepoint path
stopWithSavepoint(): savepoint path
triggerSavepoint(): CompletableFuture<savepoint path>
getAccumulators(): Map<acc name, acc value>
```

ExecutionEnvironment

ExecutionEnvironment is initialized with a cluster client that knows how to communicate with the cluster.

In user program, ExecutionEnvironment is get from context by statement “ExecutionEnvironemnt env = ExecutionEnvironemnt.get();”. Context env is set by CliFrontend. Otherwise an env with default local mode configuration would be created and used.

ExecutionEnvironment provides interfaces for job description, job compilation and job submission.

```
# initialized
<init>(configuration, ClusterClient) [in CliFrontend]
@classmethod setAsContext(factory) [setup factory]
@classmethod get(): ExecutionEnvironment [in user program]

# job description
createInput(...): DataSource
getConfig(...): ExecutionConfig
…
```

```
# job compilation
compile(): FlinkPlan
optimize(): JobGraph

# job execution
getClusterClient(): ClusterClient
execute(): CompletableFuture<JobClient>
```

Examples

1. High-level Interfaces with CliFrontend(flink)

```
// the program

public static void main(String[] args) {
  ExecutionEnvironment env = ExecutionEnvironment.get();

  // describe job topologies …

  CompletableFuture<JobClient> clientFuture = env.execute();
  JobClient client = clientFuture.get();
  CompletableFuture<JobResult> result = client.getJobResult();
  result.get(); // blocking for result
}

# execution
$ bin/flink run -c p.k.g.Program <args>
```

2. Low-level Interfaces with CliFrontend(flink)

```
// the program

public static void main(String[] args) {
  ExecutionEnvironment env = ExecutionEnvironment.get();

  // describe job topologies …

  ClusterClient client = env.getClusterClient();
```

```
  JobGraph jobGraph = env.optimize()
  CompletableFutrue<JobClient> jobClientFuture =
      client.submitJob(jobGraph);
  CompletableFuture<JobResult> result = client.getJobResult();
  result.thenAccept(...); // non-blocking
}

# execution
$ bin/flink run -c p.k.g.Program <args>
```

3. Customize Cli

```
/**
  * For downstream project developer, he's able to implement
  * his own CLI and customize cluster deployment,
  * context environment setting, and the overall process.
  */
public class MyCliFrontend {

  public MyCliFrontend(Configuration conf) { … }

  public void run(String[] args) {
    Configuration conf = Configuration.fromArgs(args);

    // …

    ClusterSpecification spec = …;
    ClusterDescriptor desc = …;
    ClusterClient client;
    client = desc.deployCluster(spec);
    ExecutionEnvironment env =
        new ExecutionEnvironment(conf, client);
    ExecutionEnvironmentFactory factory =
        new ExecutionEnvironmentFactory(conf, client);
    ExecutionEnvironment.setAsContext(factory);

    PackagedProgram program = …;
    program.invokeMain(args);

    // …
```

```
  }
}

# execution
1. directly used when programming in downstream project
2. wrap MyCliFrontend in a script and run as bin/myCli …
```