DONG DAI

HBASE CODE CRACK

Contents

HBase Introduction

According to the official website of HBase: ***Apache HBase is the Hadoop database, a distributed, scalable, big data store***. Comparing with HDFS, programmers should use HBase when you need random, realtime read/write access to your Big Data. All the data in HBase is represented as data stored in very large tables, which include billions of rows X millions of columns.

HBase is also a type of NoSQL database. More precisely, a type of NoSQL datastore instead of database because it lacks of features you find in RDBMS, such as typed columns, secondary indexes, triggers, and advanced query languages, etc.

Overview

Project Introduction

Hadoop Relevant

MapReduce Job

Jar Submission

As we have said again and again in the whole book, HDFS component in Hadoop play a critical roll in supporting our HBase persistent storage. However, At the same time, the MapReduce framework also leverages the productivity a lot. So, in this section, we will briefly discuss how the Hadoop framework supports the job submission and job/task execution.

We all know that, if developers want to run their classes or jars in Hadoop cluster, they need to issue command **Hadoop**. There are several commands which can be used to submit jobs or classes: *hadoop jar <jar>*, *hadoop CLASSNAME*, or *hadoop job*. Among them, *hadoop jar <jar>* is the most common use case because a jar file usually contains all the necessary data to run a standalone job, so it would be much easier to manipulate. Check out the source code of **hadoop** command under the directory *HADOOP_HOME/bin/hadoop*. This executable file is actually a bash script which process all the java path stuff and execute different main classes according to users commands. For example, if users issued *'hadoop jar 1.jar'*, then it will actually run *'java org.apache.hadoop.util.RunJar 1.jar'* with lots of parameters added automatically. So, in most cases, the class 'org.apache.hadoop.util.RunJar' was called to run MapReduce jobs. To process next stage, we need to check the source code of this class.

'RunJar.java' is stayed under *'util'* directory, it is so simple that it only contains two functions: **main** and **unJar**. The main function was called with one argument containing the jar file path. This main function does nearly nothing in fact, it firstly check whether the arguments number is correct which means whether users have provided their jar file paths, then it tried to get the main class by reading its manifest file. Secondly, it called **unJar** function to unzip the jar file into local temporary directory which have been configured by 'hadoop.tmp.dir'. After unzipping the jar file, we will get a correct directory arrangement, then we need add the needed classes and libs into CLASSPATH variable. ClassLoader instance will be initialized using the all the paths which we have added into the CLASSPATH variable. Set current thread's class loader, construct the argument, and finally invoke the main class. After these tree steps, we dynamically call the submitted java class in jar file. All the code of this procedure is list bellow(Listing.1). After class RunJar runs, we are gonna to run the main class of the submitted jar files.

```
ClassLoader loader = new URLClassLoader(cp.toArray(new URL[0]));
Thread.currentThread().setContextClassLoader(loader);
Class<?> mainClass = Class.forName(mainClassName, true, loader);
Method main = mainClass.getMethod('main', new Class[] {
 Array.newInstance(String.class, 0).getClass()
});
String[] newArgs = Arrays.asList(args).
  subList(firstArg, args.length).toArray(new String[0]);
main.invoke(null, new Object[] { newArgs });
```

Listing 1: RunJar main procedure

Jar Execution

Most MapReduce application's main function includes these steps: 1) build a Job instance according to the JobConf instance; 2) set up the values in Job instance like the Mapper Class and Reducer Class; 3) call **waitForCompletion** waiting for the completion of MR jobs.

The **waitForCompletion** function will call *submit* function in Job Class to build a complete Job instance, and after that call the job instance's waitForCompletion function. Listing.2 shows the core of **submit** function in Job class.

```
//Connect to the JobTracker and submit the job
connect();
info = jobClient.submitJobInternal(conf);
super.setJobID(info.getID());
state = JobState.RUNNING;
```

Listing 2: submit function in Job class

The connect() was called to build a complete JobClient instance. The JobClient constructor will set the brief configuration according to the JobConf instance firstly, and then call init() function to build a RPCProxy to connect to JobTracker as Listing.3 has shown.

```
setConf(conf);
ugi = UserGroupInformation.getCurrentUser();
this.rpcJobSubmitClient =
  createRPCProxy(JobTracker.getAddress(conf), conf);
this.jobSubmitClient = createProxy(this.rpcJobSubmitClient, conf);
```

Listing 3: JobClient init

After buiding the JobClient instance, we will call its **submitJobInternal** method to really do the submit stuff. This function will return a RunningJob instance (**'info'** in Listing.2) which contains the new job's id. Until then, we finally get the job submitted and get a global id. The local Job instance is fulfilled then. The **submitJobInternal** just contains one statement: a ugi.doAs() method call. Inside this function, we new a **PrivilegedExceptionAction** which contains a **run** method doing the real stuff as Listing.4 shown.

```
JobConf jobCopy = job;
//JobSubmissionFiles is a class delegate to obtain the job staging
//storage area, it also implemented by call jobSubmissionClient.getStagingAreaDir()
Path jobStagingArea =
  JobSubmissionFiles.getStagingDir(JobClient.this, jobCopy);
JobID jobId = jobSubmitClient.getNewJobId();
```

```
Path submitJobDir = new Path(jobStagingArea, jobId.toString());
...
copyAndConfigureFiles(jobCopy, submitJobDir);
Path submitJobFile =
  JobSubmissionFiles.getJobConfPath(submitJobDir);
...
```

Listing 4: submitJobInternal

copyAndConfigureFiles takes charge of uploading executable files and configuring current Job instance. Firstly, from the JobConf instance we can get current 'tmpfiles', 'tmpjars', and 'tmparchieves', all these parameters are set by the command line arguments. After this, we need to figure out what filesystem current JobTracker is using by calling getFileSystem function of **submitJobDir** (created in Listing.4). Since then, the real submitJobDir was created and some relevant directories are also created, like **filesDir, archievesDir,** and **libjarsDir**.

```
//Firstly, get the local files of files, jars, archives
String files = job.get("tmpfiles");
String libjars = job.get("tmpjars");
String archives = job.get("tmparchives");
//if any of them exists, we need to copy from local to dist
//filesystem
if (files != null){
  FileSystem.mkdirs(fs, filesDir, mapredSysPerms);
  String[] filesArr = files.split(',');
  for(String tmpFile: filesArr){
    tmpURL = new URL(tmpFile);
    Path tmp = new Path(tmpURL);
    Path newPath = copyRemoteFiles(fs, filesDir, tmp, job, replication);
    URL pathURI = getPathURL(newPath, tmpURI.getFragment());
    DistributedCache.addCacheFile(pathURI, job);
  }
  DistributedCache.createSymlink(job);
}
if (libjars != null) {
  FileSystem.mkdirs(fs, libjarsDir, mapredSysPerms);
  String[] libjarsArr = libjars.split(",");
  for (String tmpjars: libjarsArr) {
    Path tmp = new Path(tmpjars);
    Path newPath = copyRemoteFiles(fs, libjarsDir, tmp, job, replication);
    DistributedCache.addArchiveToClassPath
      (new Path(newPath.toUri().getPath()), job, fs);
  }
}
if (archives != null) {
 FileSystem.mkdirs(fs, archivesDir, mapredSysPerms);
 String[] archivesArr = archives.split(",");
```

```
  for (String tmpArchives: archivesArr) {
    URI tmpURI = new URI(tmpArchives);
    Path tmp = new Path(tmpURI);
    Path newPath = copyRemoteFiles(fs, archivesDir, tmp, job, replication);
    URI pathURI = getPathURI(newPath, tmpURI.getFragment());
    DistributedCache.addCacheArchive(pathURI, job);
    DistributedCache.createSymlink(job);
  }
  String originalJarPath = job.getJar();

 if (originalJarPath != null) { // copy jar to JobTracker's fs
  Path submitJarFile = JobSubmissionFiles.getJobJar(submitJobDir);
   job.setJar(submitJarFile.toString());
   fs.copyFromLocalFile(new Path(originalJarPath), submitJarFile);
   fs.setReplication(submitJarFile, replication);
   fs.setPermission(submitJarFile,
       new FsPermission(JobSubmissionFiles.JOB_FILE_PERMISSION));
 }
}
```

Listing 5: copyAndConfigureFiles

Inside this function, **copyRemoteFiles** did the file uploading work:

```
Path newPath = new Path(parentDir, originalPath.getName());
FileUtil.copy(remoteFs, originalPath, jtFs, newPath, false, job);
```

OK, from now on, we have uploaded **jar, archives, libs** files into JobTracker's filesystem. Back to the **submitJobInternal** function, we will continuously run our job. Step 1, initialize JobContext instance according to current JobConf instance and jobId we have got from JobTracker. Step 2, create the splits for the job and write the map task number into current JobConf instance. Step 3, write queue admins of the queue. Step 4, finally, write the newest JobConf instance into JobTracker's filesystem under the same directory of jars, libs, etc. Step 5, actually submit the job:

```
//jobSubmitClient is a proxy of the submission proxy.
status = jobSubmitClient.submitJob(
         jobId, submitJobDir.toString(),
         jobCopy.getCredentials());
```

When we execute here, everything in client side has finished, the rest task would be executed at JobTracker.

JobTracker submitJob

JobTracker implements the **JobSubmissionProtocol** which includes several important functions, one of them is **submitJob**. Submitting a job in JobTracker contain serval steps:

- construct a JobInfo(*jobInfo*) instance according to current jobId, jobSubmitDir etc.
- construct a JobInProgress(*job*) instance according to JobInfo instance

- check whether the job can run in the cluster.
- Store the information (jobInfo) in a file so that the job can be recovered later
- call this.taskScheduler.checkJobSubmit(job) to predict whether current job can be submitted.
- really submit the job by calling **JobTracker.addJob()** method.

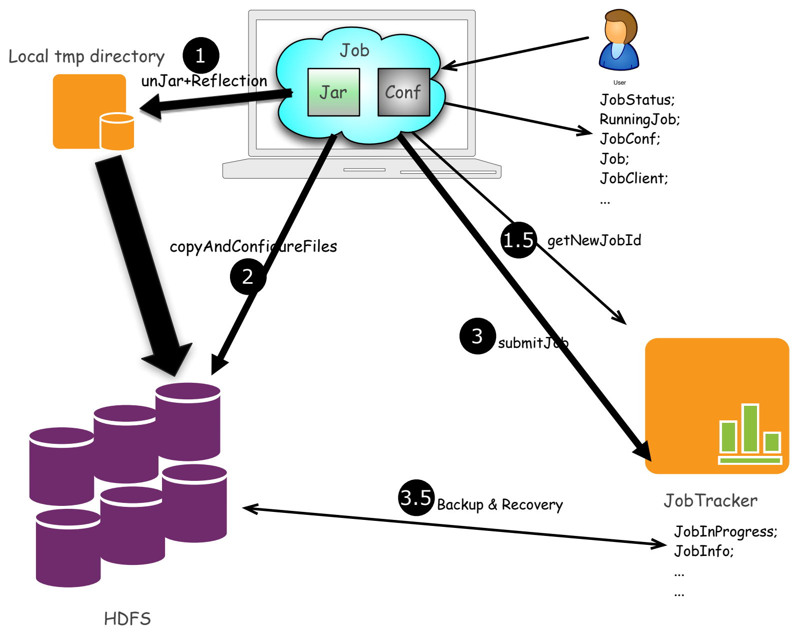
JobTracker.addJob() method is very short, it synchronizes on the **jobs** map and **taskscheduler** and add the newly submitted job into the **jobs** map as Listing.6 shows.

```
synchronized (jobs) {
 synchronized (taskScheduler) {
  jobs.put(job.getProfile().getJobID(), job);
  for (JobInProgressListener listener : jobInProgressListeners) {
   listener.jobAdded(job);
  }
 }
}
...
return job.getStatus();
```

Listing 6: addJob of JobTracker

Conclusion

Until here, we have briefly introduced the execution flow of submitting MapReduce jobs in Hadoop. This procedure is kind of common in any distributed programming framework. So, we would like to spend a little more time to summarize it.



Above Figure shows the major steps in submitting a MapReduce job in a jar file:

1. RunJar class will unjar the jar file into local temporary directory
2. Call JobTracker's RPC interface to obtain a new jobId, and the remote directory
3. copy these jar file and associate files into HDFS
4. call JobTracker's RPC interface 'submitJob' to do the real work
5. backup current jobinfo in JobTracker to do recovery in future