HadoopMapRedClasses

JobInProgress

JobInProgress represents a job as it is being tracked by JobTracker.

JobInProgress contains two sets of TaskInProgress instances:

- TaskInProgress maps[], an array holding one tip per split
- TaskInProgress reduces[], an array holding one tip per partition

Method initTask() fills out those tips. initTask also creates:

- hostMaps[], which cotains a mapping from a host to file split id's that the host is holding on DFS.

JobInProgress also offers two methods for creating a !Task instance to run, out of maps[] and reduces[]:

- obtainNewMapTask() returns a map task for a given TaskTracker to run.
- obtainNewReduceTask() returns a reduce task.

Both delegate implementation to findNewTask(), which searches for a tip in order below:

- cachedTasks for a tip whose input is hosted on the given host. still skip if the input is already processed by other task or the input has failed once on the host.
- regular task, which means a tip with an input that hasn't run.
- speculative tasks, which means a tip with an input that's already being run by other task, but the progress is slow.
- a failed tip, which means a tip with a input that was run by a failed task.

To randomize it a bit, search after cachedTasks starts with firstMapsToTry or firstReduceToTry, which is set to recently failed task's input + 1.

!Task status is updated normally when JobTracker calls JobInProgress.updateTaskStatus() as JobTracker.emitHeartbeat() processes a TaskTracker's heartbeat. updateTaskStatus then calls TaskInProgress.updateStatus(). If the tip says success or failure, updateTaskStatus() calls completedTask() or failedTask(). completeTask() checks if the tip is last partition, and if so declares job finish. failedTask() sees if the tip wants to give up after too many failed attempts, then declares job failure.

failedTask() can be directly called by JobTracker when a launching task is timed out without any heartbeat from task tracker, or when a task tracker is found to be lost. (There must be a third event where a map output is found to be corrupt or impossible to copy for whatever reason)

A job's schedule status is updated in:

- JobStatus status

TaskInProgress

TaskInProgress represents a set of tasks for a given unique input, where input is a split for map task or a partition for reduce task.

Usually there would be only one task per tip, but speculative execution and retry mechanism can create multiple tasks for a given input.

TaskInProgress has two constructor, one that takes input FileSplit (for map task), and another that takes partition number (for reduce task). These two are invoked by JobInProgress.initTask().

TaskInProgress contains:

- recentTasks, instance(s) of !Task in flight for the given input

method getTaskToRun() populates recentTasks with either MapTask or ReduceTask instance and then returns the !Task to caller. !Task ids are obtained from a field usableTaskids.

Note there is also:

- taskStatuses, a set of TaskStatus instances that contains status per task. keyed by task id.

updateStatus() puts latest status of a task into taskStatues when a tasktracer emits heartbeat containing the task status.

method completed() or failedSubTask() wraps up the life of a task by removing it from recentTasks. Each is called by JobInProgress.completeTask() or failedTask() respectively.

Note there is a class TaskTracker.TaskInProgress, which is internal to TaskTracker. TaskTracker.TaskInProgress, unlike the other one, has 1:1 mapping with a !Task. i.e. It represents only one !Task.

TaskTrackerStatus

TaskTrackerStatus represents a status of a task tracker daemon as it's monitored inside JobTracker.

It also is used as a parameter to certain methods as representative of task tracker node. Example is obtainNewMapTask() or obtainNewReduceTask() of TaskInProgress.

It contains a set of task reports in:

- Vector taskReports, a vector of TaskStatus instances.

, when JobTracker.emitHeartBeat() is invoked with a TaskTrackerStatus instance.

TaskTracker

TaskTracker is a daemon process running on each of worker node.

TaskTracker communicates with JobTracker via two RPC agents:

- taskReportServer, for reporting to JobTracker's requests for task status
- jobClient, for getting new task from and sending heartbeats to JobTracker.

TaskTracker contains instances of TaskTracker.TaskInProgress in two fields:

- tasks, for all tasks in consideration indexed by taskid strings.
- runningTasks, for subset of tasks that are being run.

A new instance of !Task is obtained by calling jobClient.pollForNewTasks(), which will drive JobTracker to eventually call TaskInProgress.getTaskToRun(). Note again that an instance of TaskTracker.TaskInProgress is created, containing the new !Task, then gets added into tasks and runningTasks.

method startNewTask() takes the new !Task then starts the actual sequence for running user code for the task. Eventually, the user code is executed in a child process. Child process communicates with the parent TaskTracker daemon via RPC. The interface for the RPC is TaskUmblicalProtocal. TaskTracker inherts and implements this protocol.

TaskTracker.TaskInProgress

TaskTracker.TaskInProgress represents a single task as it's being tracked by TaskTracker daemon.

It contains an instance of a !Task. It also contains an instance of TaskRunner, a shadow thread that monitors the !Task as it's run in child process.

TaskRunner

TaskRunner has two subclasses, MapTaskRunner and ReduceTaskRunner, which are instantiated by !Task.createRunner, when MapTask or ReduceTask is entity under the !Task interface.

TaskRunner.run(), which is the thread entry point, first invokes prepare(). ReduceTaskRunner heavily implements prepare() to fetch all input files for the reduce work. Note they are produced by map tasks, so come from many different nodes.

Then TaskRunner.run() loads user code Jar files, and calls runChild(), which creates child process. In child java process, TaskTracker.Child.main() is passed as process entry point, along with taskid string, and a port for TaskTracker's TaskUmblicalProtocal.

TaskTracker.Child.main()

TaskTracker.Child.main() offers entry point for child java process so that user code for map or reduce work can be called.

TaskTracker.Child.main() first gets an instance of !Task and job config via RPC using the TaskUmblicalProtocol. Then it calls !Task.run().

MapTask

MapTask offers method run() that calls MapRunner.run(), which in turn calls the user-supplied Mapper.map().

ReduceTask

Similarly to MapTask, ReduceTask offers run() that sorts input files using SequenceFile.Sorter.sort(), and then calls user-supplied Reducer.reduce().

ReduceTaskRunner

Key contribution is prepare() that copies all input files from many hosts. Note this code is not part of child process. only !Task.run()() is run by child.

InputFormat

InputFormat is an interface for instantiating file splits and the readers for map task.

- FileSplit[] getSplits(), creates an array of FileSplit for the job. A FileSplit is a span of a given file.
- RecordReader getRecordReader(), creates a reader for a given split.

Note the resuling split count can be more than what the config says, as a split boundary can be drawn around DFS block. This is actually implementation offered by InputFormatBase, which then gets specialized into SequenceFileInputFormat.

OutputFormat

OutputFormat is an interface for instatiating writer for consuming output of reduce task.

- RecordWriter getRecordWriter(), creates a writer for a given job.

io.SequenceFile.Sorter

Key method is sort() which calls mergePass() as 1st pass, then repeatedly calls sortPass() until it has one output file.

One thing to note is that this class uses the output file in unoffical way. A pysical file contains multiple segments that becomes input for merge for next phase. Each segment is in SequenceFile format.

The container file is named with numeric suffix that represents pass id. sortPass() thus creates file.0 that contains a large number of segments. Then each of next pass reduces the segments by merging a subset of segments from the input file.

JobTracker

JobTracker is a daemon per pool that administers all aspects of mapred activities.

jobs

JobTracker keeps all the current jobs by containing instances of JobInProgress in:

- jobs: map from jobid string to JobInProgress
- jobsByArrival: same set of JobInProgress ordered by arrival time

Main methods that access the two above are:

- submitJob(): creates/adds a JobInProgress to jobs and jobsByArrival
- pollForNewTask(): takes a task tracker id and finds a task by going through jobsByArrival. A job is selected only when its predecessor is running.

task tracker and tasks:

- taskTrackers: map from task tracker id string to TaskTrackerStatus
- trackerToTaskMap: map from task tracker id string to a set of task id strings
- taskidToTrackerMap: reverse of the above. map from taskid to tracker id string.

Note there is no map from taskid string to !Task instances.

TaskInProgress:

- taskidToTIPMap: map from task id string to TIP instance

Note JobInProgress also has all tip's.

Asynchronous works

JobTracker is event-driven. It maintains a couple of active entities:

- interTrackerServer: an RPC agent for driving JobTracker's implementation of InterTrackerProtocol and JobSubmissionProtocol
- initJobsThread, initJobs: thread for initializing JobInProgress as arriving at jobInitQueue.

- retireJobsThread, retireJobs: to scan jobs and to remove ones that finished long time ago. This is the only way that a job gets removed from jobs and jobsByArrival.
- expireTrackersThread, expireTracker, trackerExpiryQueue: to remove task trackers that seem dead
- expireLaunchingTasksThread, expireLaunchingTasks: to timeout a task that has never reported back.

id strings

job id

Job id is a string that uses format below:

- jobId: "job_" uniqueId

uniqueId is a unique numeric sequence id, which is basically JobTracker.nextJobId++. The code is in JobTracker.createUniqueId().

uniqueId is then set to JobInProgress.uniqueString, and then passed to TaskInProgress's constructors.

TaskInProgress id string

Tip id is a string that uses format below:

- map tip: "tip_" JobInProgress.uniqueString "*m*" splitId
- reduce tip: "tip_" JobInProgress.uniqueString "*r*" partitionId

Tip id is set to TaskInProgress.id. The code is in TaskInProgress.init().

task id string

!Task id is a string:

- map task id: "task_" JobInProgress.uniqueString "*m*" splitId "_" i
- reduce task id: "task_" JobInProgress.uniqueString "*r*" partitionId "_" i

Above, i is 0 to max-1, where max is TaskInProgress.MAX_TASK_EXECS + MAX_TASK_FAILURES.

TaskInProgress.init() also creates all these task ids, and set them to TaskInProgress.totalTaskIds[] and usuableTaskIds.

work dirs and path

JobTracker

JobTracker has a home dir called "jobTracker" on local file system, which is value of JobTracker.SUBDIR.

Under the dir, each JobInProgress copies two files from user-submitted location:

- job file: "jobTracker/" jobId ".xml"
- jar file: "jobTracker/" jobid ".jar"

These two paths are then set to JobInProgress.localJobFile and localJobFile, respectively.

TaskTracker

Similarly, TaskTracker has a home dir called "taskTracker". Under there, each task creates a subdir:

- task dir: "taskTracker/" taskId

Under task dir, jobFile and jarFile are created as:

- job file: "taskTracker/" taskId "/job.xml"
- jar file: "taskTracker/" taskId "/job.jar"

Jar file is just a verbatim copy. But job file is a localized one to the task, as it's modified by !Task.localizeConfiguration(). Also the local jar file's path is set in the config as the only location of jar. Then !Task.jobFile and !Task.conf all are set to this localized one.

Code sequence for above starts with TaskTracker.TaskInProgress.launchTask().

Each task also has workdir at top level. Map output files go under this dir as:

- map output for a partition: taskId "/part-" partitionId ".out"

Above, taskId has "*m*" in it as its map task id.

When the map outputs for a partition is copied to the reduce task, the local files are created as:

- reduce input from a split: taskId "/map_" splitId ".out"

Above, taskId has "*r*" in it, as it's reduce task id. Near the end, reduce creates a single sort file named:

- sort output: taskId "/all.2"

Interlim merge pass file for the sort will be:

- pass file: taskId "/all.2." passId

, where passId starts with 0.