Report of MapReduce Facility
Project 3 of 15-640
Kailiang Chen(kailianc) & Yang Pan(yangpan)

Design

Hadoop Backgroud

In this project, our goal is to design and implement a **MapReduce Facility**, similar to Hadoop, but with certain design constraints aimed at enabling it to work more efficiently in our computing environment with smaller data sets. It is capable of dispatching parallel maps and reduces across multiple hosts, as well as recovering from worker failure.
The whole MapReduce framework in Hadoop consists of four independent entities:
(1) The Client, which submits the MapReduce job.
(2) The Jobtracker, which coordinates the job run.
(3) The Tasktrackers, which run the tasks that the job has been split into.
(4) The Distributed Filesystem(HDFS), which is used for sharing job files between the other entities.

Hadoop MapReduce provides a clear interface for Application Programmer to customize specific Mapper and Reducer by overriding map and reduce functions to accomplish different jobs. When the job is submitted from application to the framework , it is passed from JobClient to the JobTracker, and then divided into tasks to pass to the TaskTracker. By RPC and HTTP transmission, each TaskTracker sends heartbeat to inform its status to the JobTracker. MapTask and ReduceTask are launched concurrently in a Child process to accomplish each small tasks, which consists of split, map, sort, partition, shuffle, and reduce. Figure1 and Figure2 describe the MapReduce workflow.
All of these work are based on a local/remote filesystem, which the latter one is called Hadoop Distributed File System(HDFS), which is made up of a NameNode and several DataNodes. Figure 3 and Figure 4 describe the HDFS read/write flow.

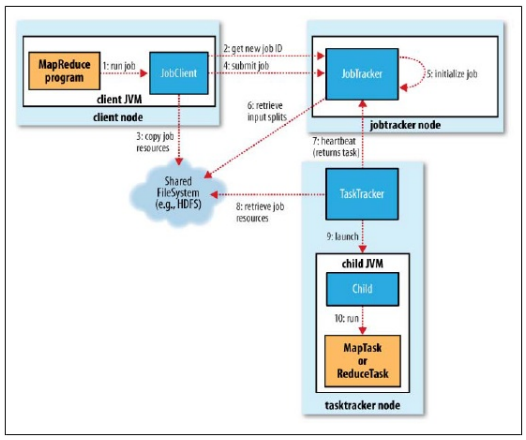

Figure 1 MapReduce Job Flow

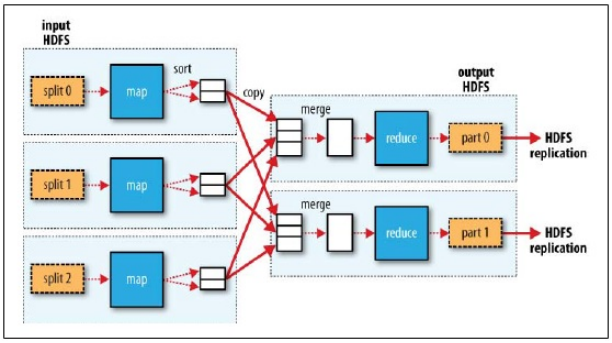

Figure 2 MapReduce Task Flow

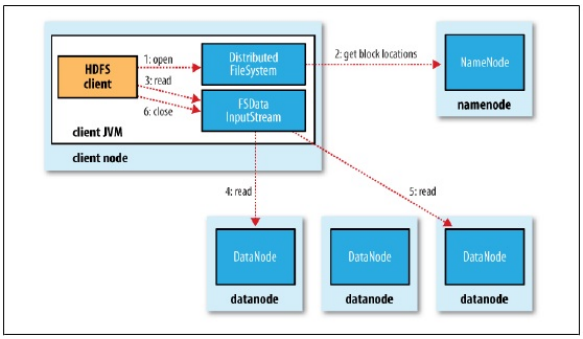

Figure 3 HDFS read flow

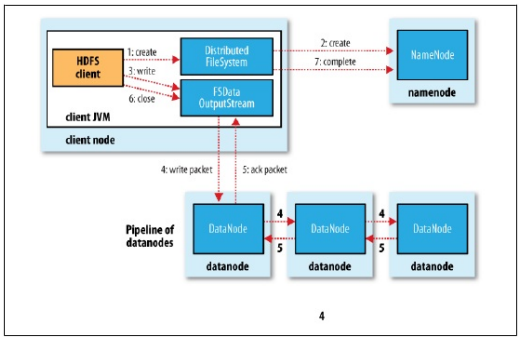

Figure 4 HDFS write flow

System Introduction

Our facility is mainly made up of two core systems, KPFS distributed file system and MapReduce system.

1. KPFS system

The architecture of our distributed file system is designed to be a DataMaster and several DataNodes, which are similar to Hadoop NameNode and DataNode. Data Master manages each file's metadata, and Data Nodes maintain real files. The overall file system structure is implemented based on KPFile class, which represents each file by its directory relative to the root directory and its file name. KPFS system provides file operation interface for manipulating files in distributed system

(1) splitFile : split the input file into chunk files before the map phase

(2) duplicateFiles : make replicas of the split files on DataNodes for failure recovery.

(3) removeFileInSlave : remove files on DataNodes

(4) getFileLocation/addFileLocation/removeFileLocation : get/register/unregister location of file on DataMaster for distributed file operation.

For each individual file, class KPFile provides file operation interface:

(1) getFileString : get the content of file by string

(2) getFileBytes : get the content of file by bytes
(3) saveFileLocally : save the file in the local DataNode and notify the DataMaster
(4) getRelPath/getLocalAbsPath : get relative path/local absolute path
The file read/write on KPFS is based on a procedure that firstly the location of the file is obtained from the DataMaster and then the content of the file is obtained from the DataNodes. So every time the file is written onto the KPFS system, the location should be registered on the DataMaster. It is similar to the metadata of HDFS in Hadoop.
The failure tolerance of the facility is mainly based on replications of the KPFS system. When some DataNode failed, the files on this DataNode will be copied onto another DataNode for failure recovery.

2. MapReduce system

The architecture of MapReduce system is made up of a Master and several Slaves. Communication between the Master and Slaves is based on Socket and Message Handling. The MapReduce procedure can be divided into Split, Map, Sort, Partition and Reduce. After the input file is split, the map and reduce tasks are dispatched to the participant nodes, which can be executed concurrently. We define a linkedlist data structure to store assigned tasks in participant nodes, which allows multithread processing of all the tasks if necessary. Accordingly, when failure happens, appropriate processing of the waiting tasks other than running ones should be considered. Here, we simplify the design to re-dispatch all the waiting tasks if failure happens.
The JobManager and JobDispater schedule and dispatch maps and reduces tasks to maximize the performance gain through parallelism. Similar to Hadoop's design, we prepare a JobQueue in JobManager. If there is a job request submitted, it will be dispatched to a participant node.
We also provide the failure recovery. Before each map task, we set a specific number of replication. If a map task on a participant node failed, we look for a new participant node, and restart the map task. Similar to Hadoop, the participant failure is detected by heartbeat transmission, in other words, if in a period of time, no heartbeat is sent to the master to report a healthy status, this participant node is considered unhealthy and all of its tasks will be restarted. Similarly, when reduce task failure is detected, all of its tasks will also be restarted.
We provide a general-purpose I/O facility to support a line-by-line input format in map and reduce tasks. To facilitate the implementation of Mapper and Reducer for application programmer, we define a Pair (Key-Value) Class and PairContainer (a list of Pairs) Class for input/output. The Mapper input is a file name, output is a PairContainer..
Compared to Hadoop, we have made some simplification. We assume that our input is a single input file, the input/output format of map and reduce is simple String type. In heartbeat transmission, we simplify the content of heartbeat and heartbeat response, e.g. we don't compute the progress of each task and each job.

Requirement Analysis

According to the Writeup, the following requirements we accomplished are listed:

1. Use configuration files, which are readily human-readable and human-editable, to configure your instance, including identifying the participant node(s), master node(s), port numbers

Yes.

We provide a Configuration File (config.txt) for customizing the following parameters

(1) Master/Participant nodes HostName, Port Number
(2) NameNode/DataNode HostName, Port Number
(3) FileChunkSize
(4) NumOfReducer
(5) Maximum Number of Mapper, Reducer Tasks
(6) SplitFile, MapperResultFile, ReducerResultFile Directory Name

2. The system should be designed to minimize dispatch latency and overhead, such as by maintaining processes at all participants, rather than by launching them a new for each job

Yes.

We initiate the execution of the program from the MapReduce main function, which can be started on any participating node, only if configuring the Master HostName.

3. Initiate the execution of the program from any participating node

Yes.

We can start our MapReduce facility from any participating node (Slave).

4. Execute portions of the program other than maps and reduces locally on the initiating node, or on other nodes, as your design should dictate

Yes.

Our Maps and Reduces are implemented on remote nodes other than locally.

5. Execute several jobs concurrently and correctly, without any concurrency related problems, except as the result of programmer-visible and mitigatable sharing

Yes.

Several jobs can be executed concurrently and correctly.

6. Schedule and dispatch maps and reduces, to maximize the performance gain through parallelism within each phase, subject to the constraints of the initiating program

Yes.

Maps and reduces schedule and dispatch are executed through parallelism with each

phase.

7. Recover from failure of map and reduce workers
Yes.
We provide failure recovery of map and reduce.

8. Provide a general-purpose I/O facility to support the necessary operations
Yes.
We have general purpose I/O facility.

9. Provide management tools enabling the start-up and shut-down of the facility, as well as the management of jobs, e.g. start, monitor, and stop.
Yes.
We provide a console to manage facility start-up and shut-down and start, monitor and stop of jobs.

10. Provide documentation for system administrators describing system requirements, configuration steps, how to confirm successful configuration, how to start, stop, and monitor the system, and how to run and manage jobs
Yes.
We provide documentation for system administrators. Please refer to System_Administrator_Manual

11. Provide documentation for programmers, describing each of your map-reduce library and your I/O library, and how they are used together. This should include programmer references, as well as a tutorial with at least two described examples.
Yes.
We provide documentation for application programmer. Please refer to Application_Programmer_Manual.

12. Include at least two working examples, including code and data or links to data
Yes.
We have two examples. One is the classic WordCounter and a new Twitter example. The latter example is based on a twitter dataset processed by JSON library to extract userId and photoNum in map phase, and compute sum of photoNum in reduce phase.

Future Improvement

Due to the limitation of time, we left some ideas unimplemented.

1. Start a task from any host including master and slaves
2. Merge all the result files into the starting machine
3. Provide more flexibility for each task, e.g. one config file for one task

UML

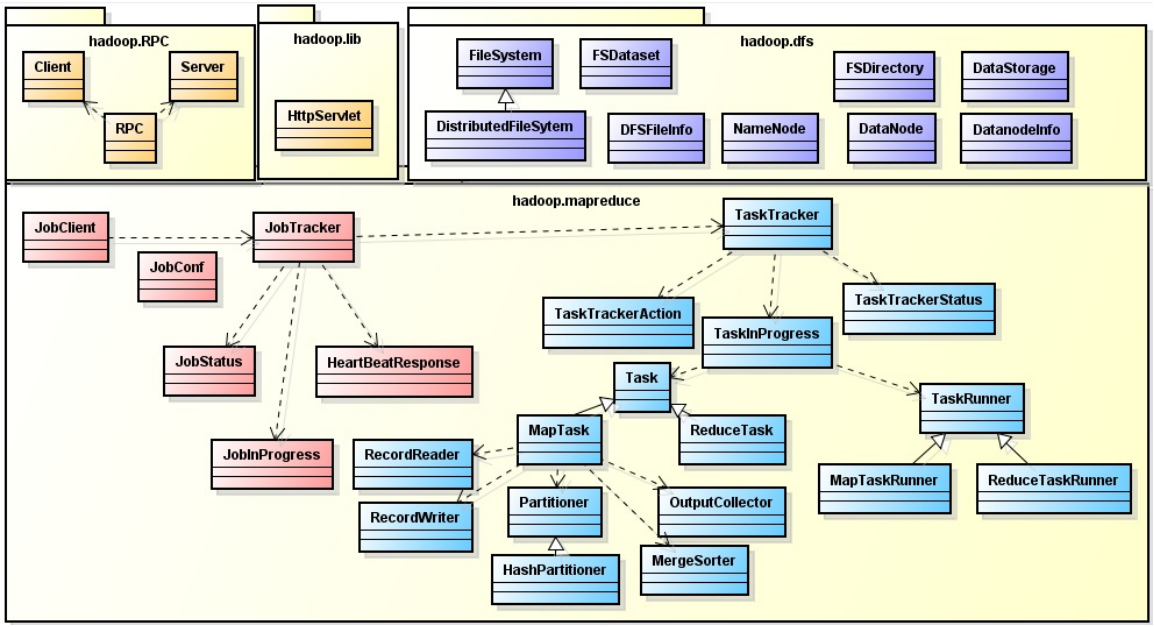

Figure 5 Hadoop MapReduce Class Diagram

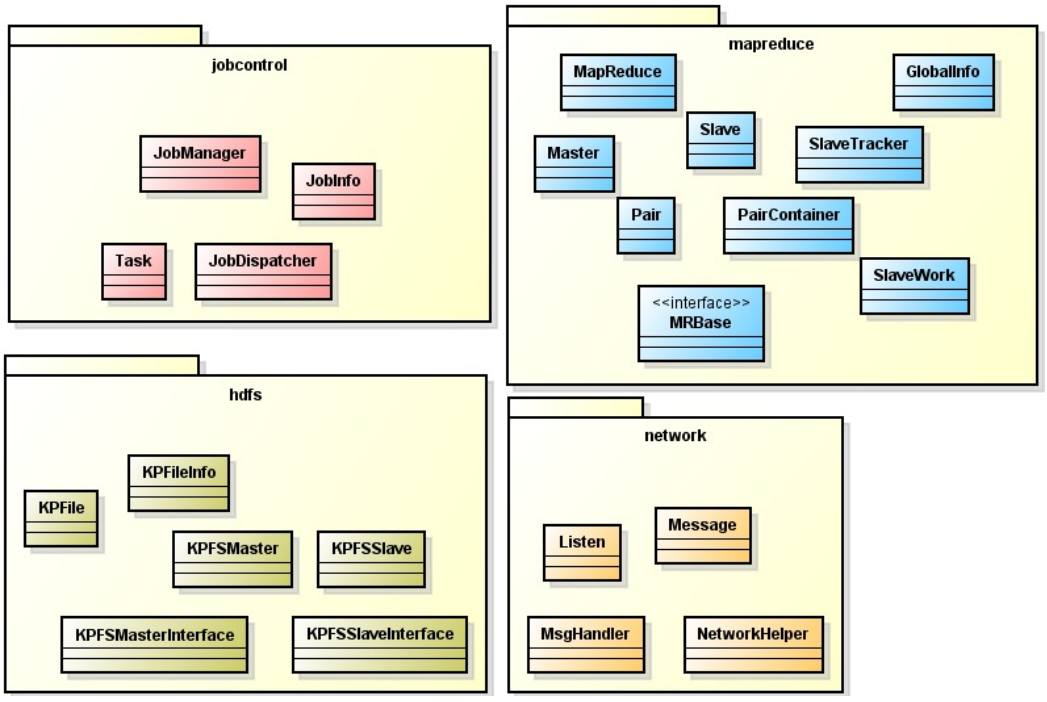

Figure 6 Our MapReduce Class Diagram

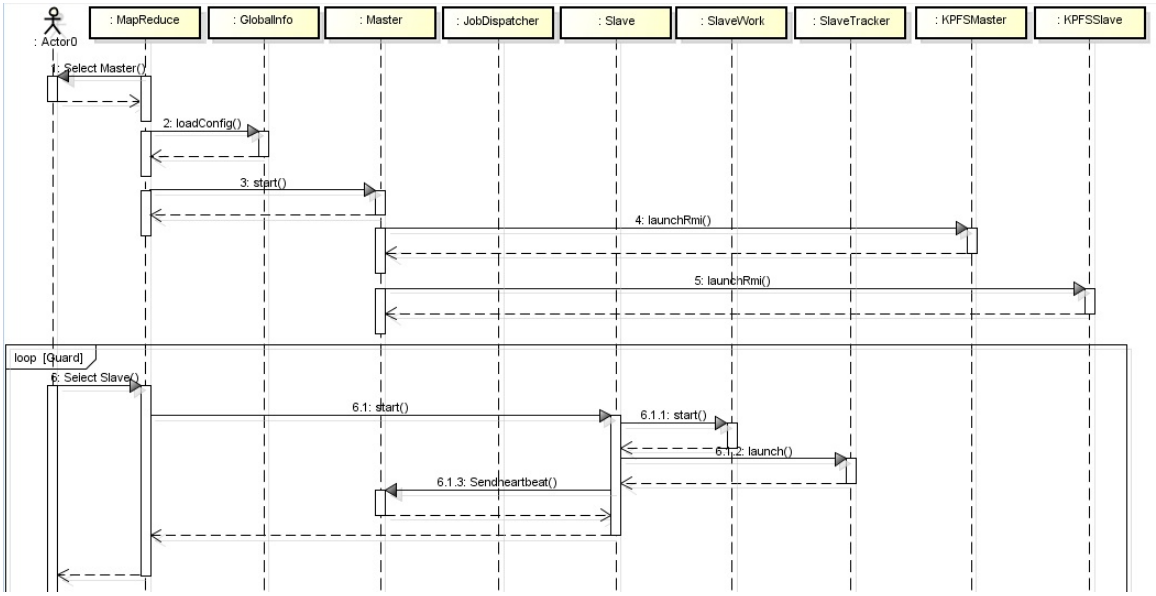

Figure 7 Initialization Sequence Diagram

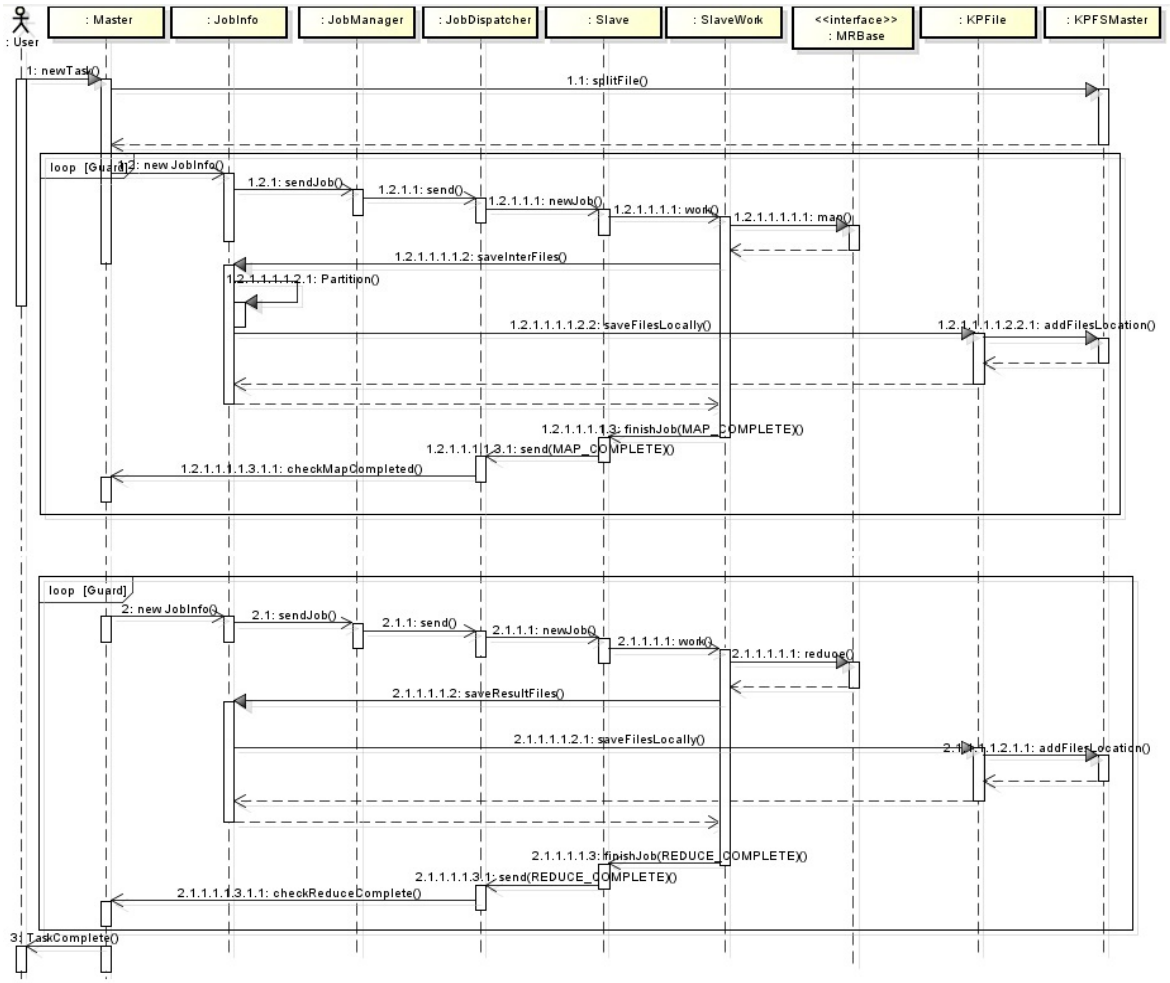

Figure 8 MapReduce Sequence Diagram

Special features

1. Support MapReduce Master/Participant node, DFS Namenode/Datanode configuration
2. Support files transferring and read/write between MapReduce and DFS based on RMI
3. Support message communication, including task dispatch between Master and Participant node
4. Support replication and failure recovery during map and reduce tasks
5. Support easy-to-use Mapper/Reducer interface for Application Programmer
6. Support easy-to-extend and loosely coupled communication module

Implementation

Class Implementation

According to the UML class diagram, our system consists of:

1) KPFile : class encapsulates all the file operation in KPFS.
2) KPFSException : customized exception in KPFS.
3) KPFSFileInfo : class used in data master to store the location and size of a KPFile.
4) KPFSMaster : service class in data master, mainly used to provide the location information of every KPFile.
5) KPFSMasterInterface : interface for KPFSMaster
6) KPFSSlave : service class in data node, used to provide the content of a file node and store a new file in this data.
7) KPFSSlaveInterface : interface for KPFSSlave

8) JobDispatcher : class to periodically check if the sending queue in JobManager is empty. If not, send the queueing job out to the specific slave
9) JobInfo : class to store all the information of a job and provide some convenient methods to get the information of this job.
10) JobManager : manager class encapsulate the operation to sending queue in master
11) Task : class to store all the information of a task and provide some convenient methods to get the information of this task. Note: we use a different description from Hadoop. Task is larger work and the job is smaller work.

12) GlobalInfo : class to store all the information in "config.txt". And provide some convenient methods to get the information. Served for both map-reduce system and KPFS.
13) MapReduce : the main entrance of the entire system.
14) Master : the main thread of map-reduce master. Read in the user input and handle the commands. Also provide the handlers for messages from slaves. Maintain all the running tasks.
15) MRBase : the interface for user-defined map-reduce class. All developers should implement this interface to write their map and reduce methods.
16) Pair : a structure to store a key-value pair
17) PairContainer : a structure to store a pair
18) Slave : the main thread of a map-reduce slave.
19) SlaveTracker : class to store the status of a slave. Sent as the content of a heart beat from map-reduce slave to master
20) SlaveWork : the working thread in slave. Only one instance is coordinator that will

check the queue and put the job in that queue into work. The other instances will be the worker.

21) Listen : class used by map-reduce master which is responsible for accepting connection from slaves. Running in a standalone thread.
22) Message : the message structure used in communication between map-reduce master and slaves. Every message has a type indicating its purposes.
23) MsgHandler : caused by map-reduce master and slaves. Distribute the incoming message to different handlers.
24) NetworkFailInterface : implement this interface to handle the situation where a connection to sid is down.
25) NetworkHelper : helper class with some function defined 1. Sending and receiving message with socket (map-reduce) 2. Get the service of RMI (KPFS)

I/O Library

For programmers who use our system

We simplify the Hadoop MapReduce library and provide two classes Pair and PairContainer for Application Programmer.

Pair is a class which has a String type key and an ArrayList<String> type value. It has following APIs

(1) `Pair(String line)`: constructor which will create a Pair if a line of String is input (the defined delimiter is colon)
(2) `Pair(String key, value)`: constructor which will create a Pair if a String type key and a String type value is input
(3) `Pair(String key, Iterator<String> val)` : constructor which will create a Pair if a String type key and a String iterator is input
(4) `String toString()`: a Pair can convert to a String output, with key : value1,value2,…,valueN
(5) `String getFirst()/setFirst()` : getter/setter for String type key
(6) `Iterator<String>getSecond/setSecond()`: getter/setter for Iterator<String> type value
(7) `int compareTo ()`: Pair instances will need to be sorted by key in PairContainer so it has to implement Comparable

`PairContainer` is a class which maintains a series of Pair instances. It has a

`ArrayList<Pair>` structure to put/get/sort if needed. It has the following APIs:

(1) `PairContainer(Iterator<Pair> itor)` : constructor which will create a PairContainer if an iterator to a Pair data structure is input.
(2) `void emit(Pair pair)` : put a Pair instance into management.
(3) `void emit(String key, String val)` : construct and put a Pair instance into management if a String type key and String type value is input
(4) `void mergeSameKey()` : sort and merge the values if the key is the same
(5) `Iterator<Pair> getInitialIterator()` : get the iterator to the list of values
(6) `String toString()` : a PairContainer can convert to a String, with each Pair per line
(7) `void restoreFromString()` : create a PairContainer from a String input, a Pair per line, a key with a list of value per Pair

For programmers who want to improve/understand our system

We designed the KPFS based on HDFS. Every file in this file system is identified by the relative path, which is relative to the root directory specified in "config.txt". We implemented a class "KPFile" that encapsulates all the file operations (read and write) on KPFS. In our system, we take advantage of "KPFile" to simplify our I/O operation. For more information, please refer to the comments in the source code of KPFS, which is in package "hdfs".