Hadoop archive is a facility which packs up small files into one compact HDFS block to avoid memory wastage of name node.name node stores the metadata information of the the HDFS data.SO,say 1GB file is broken in 1000 pieces then name node will have to store metadata about all those 1000 small files. In that manner, name node memory will be wasted it storing and managing a lot of data.

HAR is created from a collection of files and the archiving tool will run a Map Reduce job. these Maps reduce jobs to process the input files in parallel to create an archive file.

HAR command
Hadoop is created to deal with large files data .So small files are problematic and to be handled efficiently.

As large input file is splitted into number of small input files and stored across all the data nodes, all these huge number of records are to be stored in name node which makes name node inefficient. To handle this problem, Hadoop Archieve has been created which packs the HDFS files into archives and we can directly use these files an as input to the MR jobs. It always comes with *.har extension.

HAR Syntax:
hadoop archive -archiveName NAME -p <parent path> <src>* <dest>
Example:
hadoop archive -archiveName foo.har -p /user/hadoop dir1 dir2 /user/zoo
If you have a hadoop archive stored in HDFS in /user/zoo/foo.har then for using this archive for MapReduce input, all you need to specify the input directory as har:///user/zoo/foo.har.

If we list the archive file:

```
$hadoop fs -ls /data/myArch.har

 /data/myArch..har/_index
 /data/myArch..har/_masterindex
 /data/myArch..har/part-0
```

part files are the original files concatenated together with big files and index files are to look up for the small files in the big part file.

Limitations of HAR Files:

1) Creation of HAR files will create a copy of the original files. So, we need as much disk space as size of original files which we are archiving.We can delete the original files after creation of archive to release some disk space.
2) Once an archive is created, to add or remove files from/to archive we need to re-create the archive.
3) HAR file will require lots of map tasks which are inefficient.

data compression

For any large-capacity distributed storage, file compression is required, the benefits of file compression are:

I: Reduce the storage space required for files;

II: Accelerate the transfer rate on the network or on the disk;
Hadoop About file compressed code is almost all in package.org.apache.hadoop.io.compress.

(1) Hadoop selection of compression tools

压缩格式	工具	算法	文件扩展名	多文件	可分割性
DEFLATE*	无	DEFLATE	.deflate	不	不
Gzip	gzip	DEFLATE	.gz	不	不
ZIP	zip	DEFLATE	.zip	是	是，在文件范围内
bzip2	bzip2	bzip2	.bz2	不	是
LZO	lzop	LZO	.lzo	不	不

Compression is generally a trade-off on time and space. In general, longer compression time will have more space. There is a certain difference between different compression algorithms.
(2) Compression split and input split
It is important to compress segmentation and input division. BZIP2 supports file segmentation, users can read each piece of content and processed separately, so Bizip2 compressed files can be divided into storage
(3) Using compressed in the MapReduce program
Using compressed in MapReduce is very exciting, you can configure confctions when it is configured in the JOB configuration.

The code for the compressed data after setting the MAP processing is as follows:

JobConf conf = new Jobconf();

conf.setBoolean(“mapred.compress.map.output”,true);

For general conditions, compression is always good, whether it is compressed or the intermediate data after the MAP processing is compressed.

CodeC

implements a compressed decompression algorithm. Compression decompression class in Hadoop Recommend COMPRESSIONCODEC interface

createoutputStream to create a compressionoutputstream, write its compressed format into the underlying stream

Demonstrates a 1.bzip2 algorithm compressed file decompression, then compress the decompressed file into 2.GZ

Local library

Hadoop Develops using Java, but some needs and operations are not suitable for Java, so the local library Native is introduced. Some operations can be performed efficiently. If you use Gzip compression and decompression, use a local library to shorten approximately 10% by using Java time, unzip 50%. Under Hadoop_Home / Lib / Native

You can set up NATIVE in Hadoop configuration file core-site.xml

Hadoop.native.lib

true

The default is to enable the local library. If you frequently use the native library to decompress the rule task, you can use CodeCpool, get the compressor object through the CodeCpool's getCompressor method, you need to pass

CODEC. This saves creating a CODEC object overhead, allowing it to be used repeatedly.

Serialized operation in I / O

Serialization is a method of converting an object into a byte stream, or a method of describing an object with a byte stream. The relative serialization is reverse selecente, and the deserialization is a method of converting byte stream into an object. Serialization has two purposes:
Interprocess communication;
Data Persistence Storage;
Hadoop uses RPC to implement inter-process communication. In general, the serialization mechanism of RPC has the following characteristics:
Compact: Compact formats can make full use of bandwidth, speed up the transmission speed;
Quick: Can reduce serialization and reverse sequencing overhead;
Scalability: Come can be gradually changed, which is directly related to the server side;
Interoperability: Supports the client and server interaction data written in different languages;
In Hadoop, serialization is in a core position. Because the storage file or the transfer data is transmitted, the serialization process needs to be executed. Hadoop did not adopt Java Object Serialization, but he wrote a serialization mechanism Writeables, which is compact, fast, more convenient.
(1) Writable class
Writable is the core of Hadoop, Hadoop defines the data type and operation based on Hadoop. In general, whether it is uploaded downloading data or runs a MapReduce program, it is not necessary to use Writable classes.

Writeable class only defines two methods: serialized output data streams and reverse sequence input data streams;

or this classification: Write its status to DataOutput binary flow (serialization) and read status from DataInput binary flow (sequence);

1) Hadoop's comparator

WritableComparable is the interface class in HadoP.

In executing mapredure, we know that the default sorting of Key is the credit of WritableComparable.

2) Data type in the Writable class

Java basic class; for example: Boolean, Byte, Int, Float, Long, Double (6)

Other classes

NullWritable: This is a placeholder, his serialization length is 0

byteswritable and byteswritable: byteswritable is a package of binary data arrays; ByTewritable is a binary data package

text: Hadoop overrides the String type, using standard UTF-8 encoding. It can be understood in Java String class, but there is a certain difference, such as index, variability, etc.

ObjectWritable: A multi-type package.

ArrayWritable and TwoArrayWritable: The data type built for arrays and two-dimensional arrays.

mapwritable and sortedmapwritable: Siza is the implementation of java.util.map () and java.util.SortedMap ().

compressedWritable: Save the data structure of the compressed data.

genericWritable: Universal data package type.

VersionedWritable: An abstract version check class.

(2) Implement your own Hadoop data type

Hadoop can support your own data type.

Serialized frame AVRO

Although most of the MapReduce programs use the Writeable type of Key and Value, it is not forced to use. Any type can be used, as long as there is a mechanism to turn back and forth in the type and binary representation of each type.

The sequenceless framework is to solve this problem, which is implemented in a serialization, and AVRO is a serialization framework.

Hadoop I/O serialization

First, about serialization:

Serialization, can store "live" (only survive in memory, power off when there is no power, generally can only be used by local processes, but can not be sent to another computer on the network) objects, you can The live "object" is sent to the remote computer.

To serialize a "live" object is to convert a "live" object into a string of bytes, and "deserialization" is to parse a "live" object from a string of bytes. So, if you want to store the "live" object in a file, store the string of bytes. If you want to send the "live" object to the remote host, send the string of bytes. When you need the object, do it. By deserializing, the object can be "resurrected".

Serializing objects to a file, the term is also called "persistence." The object is serialized and sent to a remote computer, also known as "data communication."

The purpose of serialization:

- Interprocess communication

- Data persistence storage

two,**Serialization of HADOOP:**

Hadoop uses RPC for internal communication between nodes. The RPC protocol translates the message into a binary byte stream and sends it to the remote node. The remote node then converts the binary stream into the original information through deserialization. The serialization of RPC needs to achieve the following:

1. Compression can play the role of compression, and the occupied broadband resources should be small.

2. Fast, internal processes build high-speed links for distributed systems, so they must be fast between serialization and deserialization, and cannot make transmission speed a bottleneck.

3. Scalable, the new server adds a parameter to the new client, and the old client can still use it.

4. Compatibility is good, can support clients in multiple languages

In Hadoop, serialization is at the core. Because whether you are storing files or transferring data in a calculation, you need to perform a serialization process. The speed of serialization and deserialization, and the size of the serialized data all affect the speed of data transmission, which affects the efficiency of the calculation.

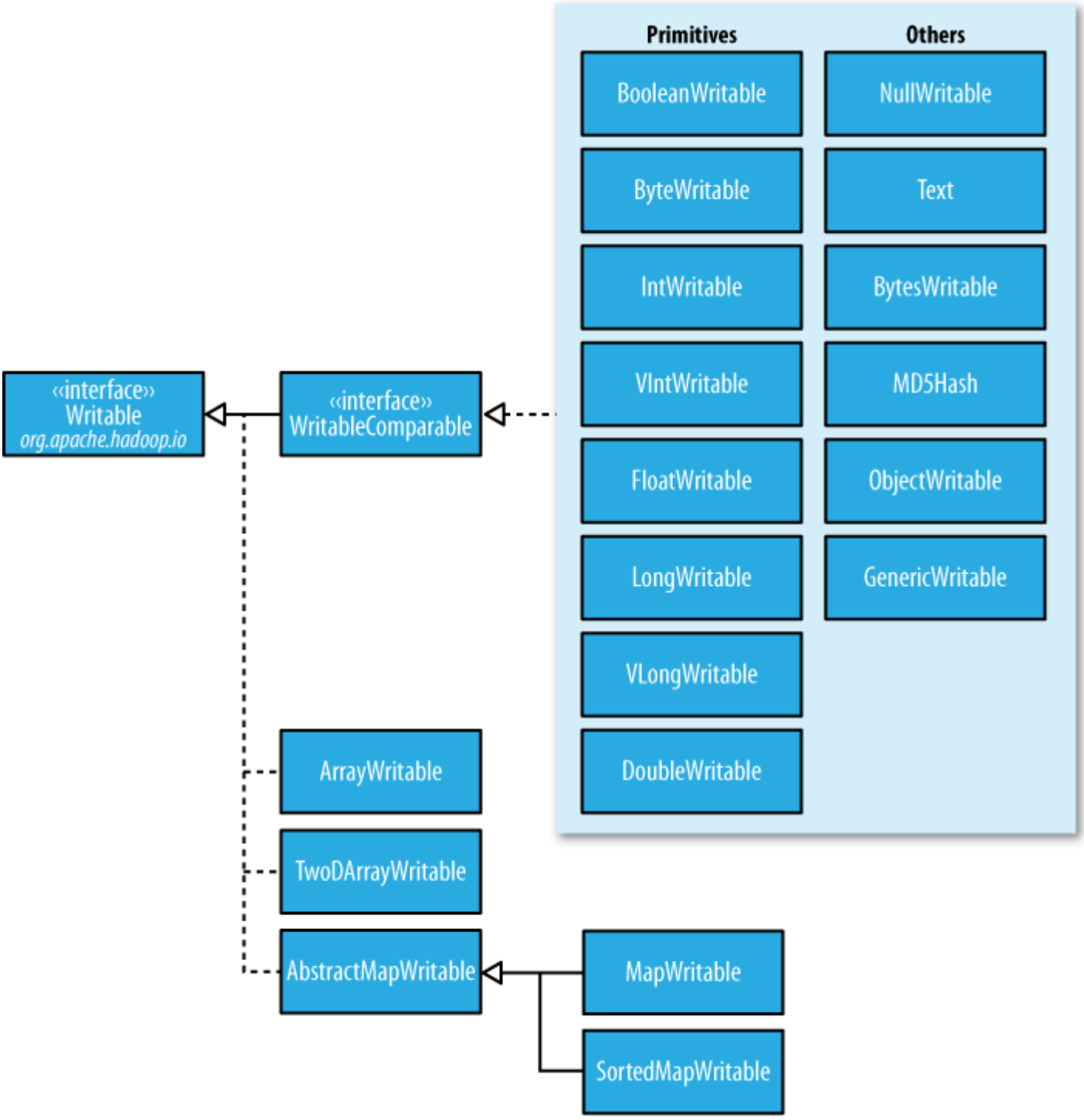
Why not use java serialization?

- The serialization mechanism of java saves the information of each class when the object of each class first appears, such as the class name. The second class object will have a reference to the class, resulting in wasted space.
- There are thousands of objects (for example, more than that) to be deserialized, and the java serialization mechanism cannot reuse objects. When java is deserialized, each time a new object is constructed, in Hadoop In the serialization mechanism, deserialized objects are reusable.
- Java serialization is not flexible enough, and it is easy for hadoop to write serialization itself, and java is not.

The following describes the serialization mechanism of Hadoop:

Writable class:

Writable is the core of Hadoop, which is used by Hadoop to define the basic data types and operations in Hadoop. The Writable class is also the org.apache.hadoop.io.Writable interface. All serializable objects of Hadoop must implement this interface. The key and value in Hadoop must be objects that implement the Writable interface to support the MapReduce task. Serialization and deserialization. There are two methods in the Writable interface, one is the write method, the object is written to the byte stream, and the other is the readFields method, which parses the object from the byte stream. Here is the hierarchy of the Writable class:

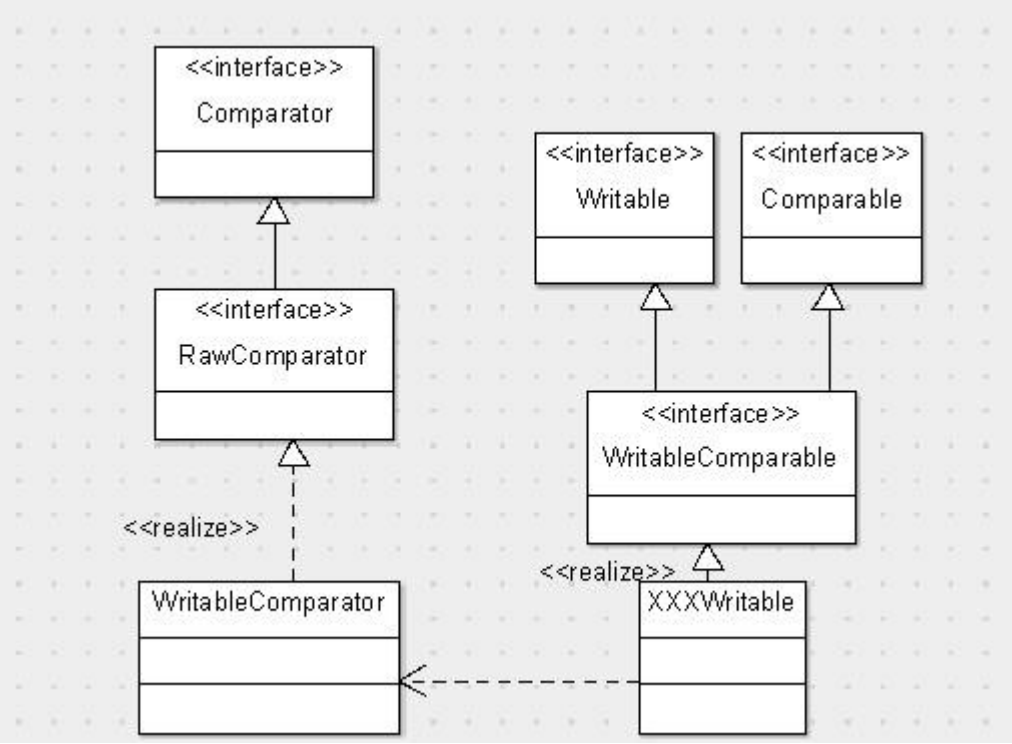


From the above figure, we can see that classes such as BooleanWritable and ByteWritable are not directly inherited from the Writable class, but inherit from

the WritableComparable class. WritableComparable is a very important interface class in Hadoop. The WritableComparable interface is a serializable and comparable interface. All key value types in MapReduce must implement this interface. Since it is serializable, you must implement the two serialization and deserialization functions readFiels() and write(). Since it is also comparable, then The compareTo() function must be implemented. This function is the implementation of the comparison and sorting rules, so that the key value in MR can be serializable and comparable, so why is the key value of MR in Hadoop comparable? of? How does Hadoop implement the comparator? With these two questions, let's introduce the Hadoop comparator.

<a>, Hadoop comparator

As we all know, when MapReduce is executed, the Reducer (the machine that performs the reduce task) will collect the key/value pairs of the same KEY value, and there will be a sorting process before reduce. The sorting is the core technology of MapReduce, although the application itself does not. Data needs to be sorted, but MapReduce's sorting capabilities can be used to organize the data. The WritableComparable comparator is implemented by WritableComparator. The comparison of MR key values is done by comparing the WritableComparator types. The class diagram is as follows:



From the class diagram, we can also see that the WritableComparator implements the RawComparator interface class. The RawComparator class defines the reading of un-serialized objects, which can be directly used to serialize the comparison between bytes. ? Everyone knows that interprocess communication on multiple nodes of Hadoop is implemented by remote procedure call RPC, and the RPC protocol serializes the message into a binary stream and then sends it to the remote node. The remote node receives the binary stream and then deserializes it. For the original message, if you compare directly to the serialized binary stream, it will improve efficiency and avoid the overhead of creating objects.

The only way to get the RawComparator interface is int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2), where b1 is the byte array in which the first object is located, and s1 is the object in The starting position in b1, l1 is the length of the object in b1, b2 is the byte array in which the first object is

located, s2 is the starting position of the object in b2, and l2 is the length of the object in b2. The WritableComparator class is a subclass of RawComparator that compares objects that implement the WritableComparable interface. The basic implementation is to compare in natural order. If you want to compare in other order, such as reverse order, you can inherit the class and override compare(WritableComparable, WritableComparable) to implement the custom comparator. This is because in the compare method for bytes, the compare(WritableComparable, WritableComparable) method is finally called, so the method determines how the comparison is performed. The comparison operation can be optimized by overriding compare(byte[], int, int, byte[], int, int), which provides a number of static methods to aid in the optimization of the method.