Netty replace Grpc Design

Summary

The purpose of this design is to reduce the serialization/deserialization time of Uniffle Client and ShuffleServer, as well as the GC Time of ShuffleServer

Motivation

Benefits of using this restricted feature

1. reduce the serialization/deserialization time of Uniffle Client and ShuffleServer
2. Reduce the GC Time of ShuffleServer, especially the pause time caused by FullGC

Goals

I will implement the following functions:

1. Use Netty to replace Grpc to implement ShuffleServer
2. Use off-heap memory to manage shuffle data cached in ShuffleServer

Design Details

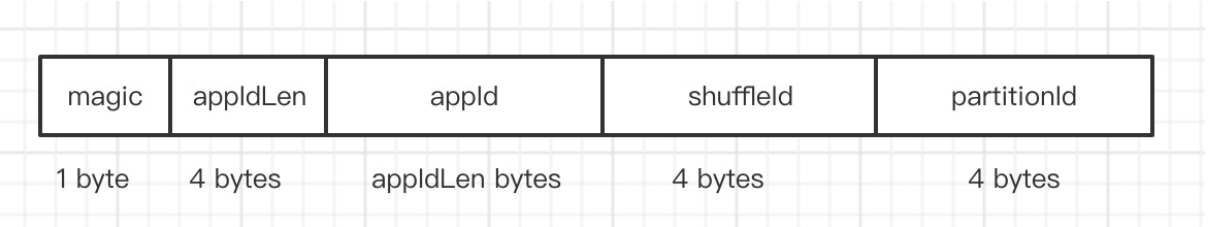
The main communication components are Client, Coordinator and ShuffleServer.

ShuffleServer

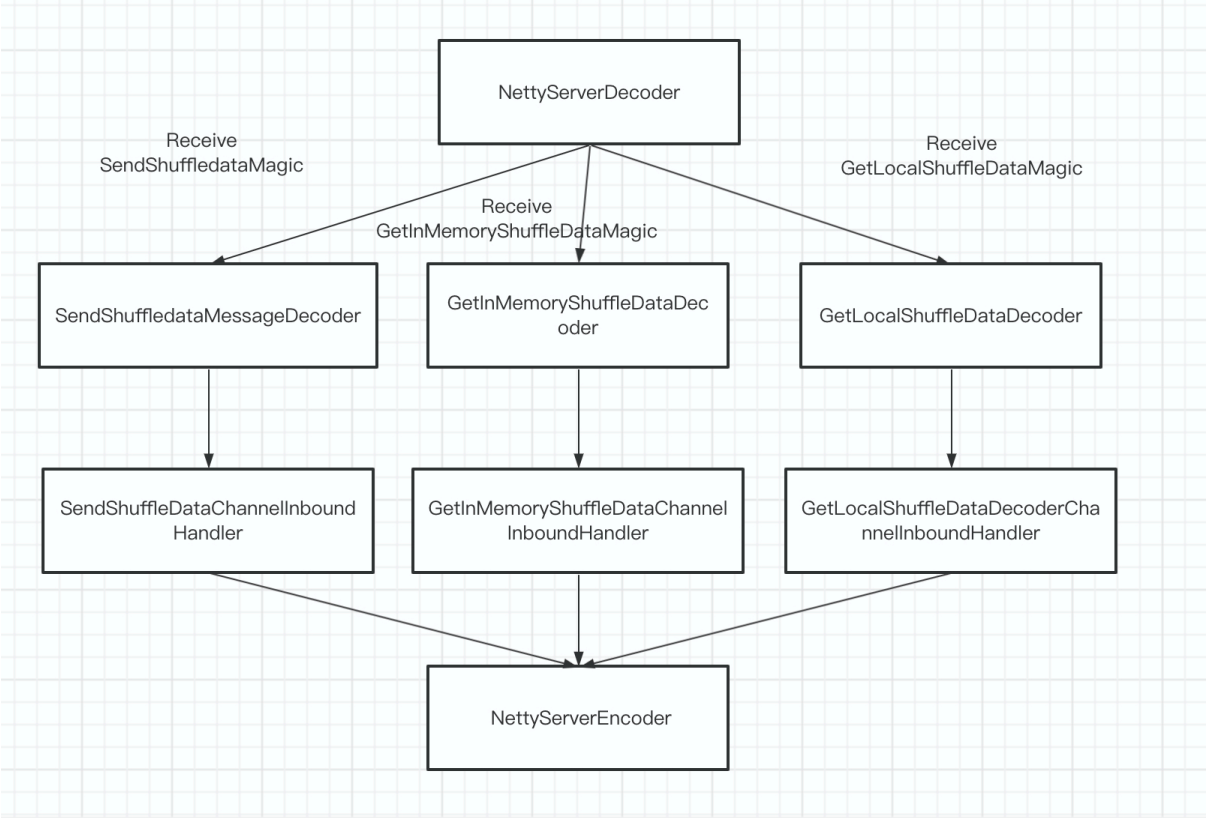
Serialization/deserialization protocol rewriting

After replacing Grpc with Netty, we need to rewrite the serialization/deserialization protocol of each interface. After rewriting, theoretically the transmitted content will become less than Grpc.

Taking a simple interface **GetShuffleResultRequest** as an example, the serialized content will be as follows, there is only one redundant byte here to identify the request type.



I will implement Decoder/Encoder and InboundHandler that handles the corresponding interface processing. Decoder is used to deserialize the binary data sent by the client into a message object, Encoder is used to serialize the response object, and InboundHandler contains the interface processing logic of ShuffleServer



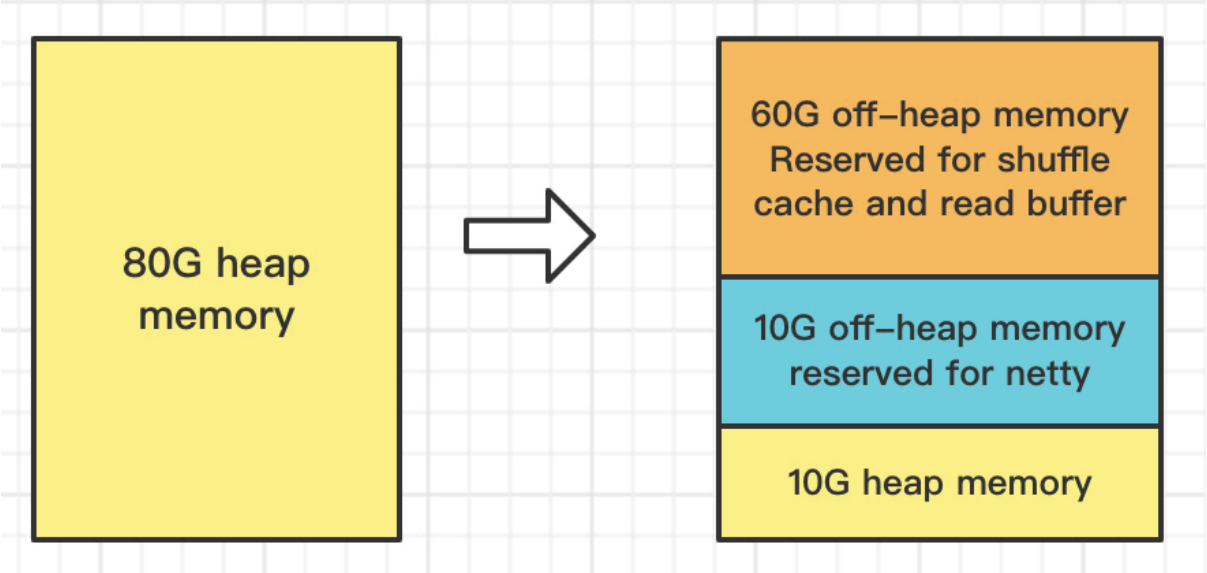
Use off-heap memory to manage shuffle data

Currently, the shuffle data cached in Shuffle Server is managed through on-heap memory, and we will replace it with off-heap memory, which will greatly reduce GC Time and avoid FullGC when ShuffleServer flushes data. This means that the data in **org.apache.uniffle.common.ShufflePartitionedBlock** will be allocated through off-heap memory.

Risk point

To use off-heap memory, we need to manage the allocation size of off-heap memory. When the off-heap memory requested by the jvm process exceeds -XX:MaxDirectMemorySize, **an OOM exception will be thrown**. Considering this problem, ShuffleServer mainly uses the off-heap memory in two places, the first place is the buffer needed by the Netty server. The size of off-memory used here will be related to the number of concurrent requests processed and the content of the request. Here we expect to use a fixed configuration. Reserve a part of the memory. In addition, we need to pass some pressure tests to prove that the reserved memory is sufficient. The second place is the cache of shuffle data, which is fixed here, as long as it matches rss.server.buffer.capacity + rss.server.read.buffer.capacity is equal.

Assuming Xmx=80G, the memory allocation after design will be like this



Client

Serialization/deserialization protocol rewriting

Reuse the Encoder/Decoder logic in ShuffleServer.

Connection management

I would implement TransportClientFactory similar to TransportClientFactory in spark to create and manage client connections.

Coordinator

The Coordinator is not important, because the load of the Coordinator is low. In the early stage, we mainly realized the design of the above-mentioned Client and ShuffleServer, and finally refactored the interface of the Coordinator

to do list

1. Implementation of basic framework is used for netty to replace grpc, mainly including Encoder and Decoder
2. Rewrite the serialization/deserialization protocol of each interface in ShuffleServer
3. Implementation of client connection management
4. The startup script supports setting Xmx and MaxDirectMemory according to whether netty is enabled
5. Support ShuffleServer to use off-heap memory to manage shuffle data
6. coordinator supports netty

Performance Test

Purpose

Test the processing performance of a single ShuffleServer by testing a production task with a 200G shuffle data, to compare the performance difference between different RSS.

Environment

hardware environment

CPU: Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz * 2 , 48 processors
Memory: 128G
Disk: nvme ssd 8T * 1(max read 2.5g/s, max write 1.3g/s)
Bandwidth: 25G

Test task information

Stage	number of tasks	shuffle read	shuffle write	output
Stage-1	394	\	216G	\
Stage-2	1000	216G	\	64G

spark conf: --conf spark.driver.memory=15g --conf spark.sql.adaptive.enabled=false --conf spark.dynamicAllocation.enabled=false --conf spark.shuffle.service.enabled=false --num-executors 200 --conf spark.executor.cores=2 --conf spark.executor.memory=4g

key configuration

Apache Uniffle

version: 0.6.0
conf:
rss.server.buffer.capacity 30gb
rss.server.read.buffer.capacity 15gb
rss.server.flush.thread.alive 2
rss.server.flush.threadPool.size 2
rss.storage.type MEMORY_LOCALFILE
rss.server.single.buffer.flush.enabled true
rss.server.single.buffer.flush.threshold 64mb
XMX_SIZE="60g"

Apache Celeborn

version: 0.2.0-SNAPSHOT
conf:
rss.rpc.io.serverThreads 16
rss.push.io.threads 80
rss.fetch.io.threads 80
celeborn.push.replicate.enabled false
celeborn.worker.storage.dirs /data/rssdata:disktype=SSD:flushthread=8

ByteDance CSS

version: 1.0.0
conf:
export WORKER_JAVA_OPTS="-Xmx8192m -XX:MaxDirectMemorySize=52g"
css.push.io.threads = 128
css.fetch.io.threads = 64

Uber RSS

version: 0.0.9
conf:
-Xmx40g -XX:MaxDirectMemorySize=20g

Test Results

I counted the time consumption of the app, the time consumption of the stage, the physical usage rate of the machine where the worker process is located, and the gc statistics of the worker process.

rss_name	cost_time	stage 1	stage 2	peek_cpu_usage	peek_io_util	peek_in_bytes	peek_out_bytes	full_gc_times	young_gc_times	gc_time
internal uniffle	7.4min	4.3min	3.1min	88.5%	61.1%	1.4g/s	1.6g/s	1	144	44.3s
celeborn	4.5min	2.0min	2.5min	51.5%	98.2%	2.2g/s	2.0g/s	1	19	1.3s
css	4.7min	2.2min	2.5min	23.8%	100%	1.9g/s	2.1g/s	3	18	0.7s
uber	6.0min	3.5min	2.5min	29.2%	100%	1.5g/s	2.2/gs	0	382	1.0s

Conclusion

Apache Celeborn and ByteDance CSS perform best, they have similar designs, They design off-heap memory as a cache for shuffle data, and the fetch/push server built on netty has better throughput performance, come Uber CSS, and finally Apache Uniffle.