# Deep Understanding of Spark Memory Management Model

- by Alex
- ·
- 07 Aug 2018
- ·
- Topics: Spark , Memory Management
- ·
- 5495 Views

## 1. Introduction

As a memory-based distributed computing engine, Spark's memory management module plays a very important role in a whole system. Understanding the basics of Spark memory management helps you to develop Spark applications and perform performance tuning.

Generally, a Spark Application includes two JVM processes, **Driver** and **Executor**. The Driver is the main control process, which is responsible for creating the Context, submitting the Job, converting the Job to Task, and coordinating the Task execution between Executors. The Executor is mainly responsible for performing specific calculation tasks and returning the results to the Driver. Because the memory management of Driver is relatively simple, and the difference between the general JVM program is not big, I'll focuse on the memory management of Executor in this article. Therefore, the memory management mentioned in this article refers to the memory management of Executor.

## 2. On-Heap memory and Off-Heap memory

Executor acts as a JVM process, and its memory management is based on the JVM. So JVM memory management includes two methods:

- On-Heap memory management: Objects are allocated on the JVM heap and bound by GC.
- Off-Heap memory management: Objects are allocated in memory outside the JVM by serialization, managed by the application, and are not bound by GC. This memory management method can avoid frequent GC, but the disadvantage is that you have to write the logic of memory allocation and memory release.

In general, the objects' read and write speed is:

> on-heap > off-heap > disk

## 3. Memory allocation

In Spark, there are supported two memory management modes: Static Memory Manager and Unified Memory Manager.

Spark provides a unified interface MemoryManager for the management of Storage memory and Execution memory. The tasks in the same Executor call the interface to apply for or release memory. When coming to implement the

MemoryManager, it uses the StaticMemory Management by default before Spark 1.6, while the default method has changed to the UnifiedMemoryManager after Spark 1.6. In Spark 1.6+, static memory management can be enabled via the spark.memory.useLegacyMode parameter.

## 3.1. Static Memory Manager

Under the Static Memory Manager mechanism, the size of Storage memory, Execution memory, and other memory is fixed during the Spark application's operation, but users can configure it before the application starts. Though this allocation method has been eliminated gradually, Spark remains for compatibility reasons.

Here mainly talks about the drawbacks of Static Memory Manager: the Static Memory Manager mechanism is relatively simple to implement, but if the user is not familiar with the storage mechanism of Spark, or doesn't make the corresponding configuration according to the specific data size and computing tasks, it is easy to cause one of the Storage memory and Execution memory has a lot of space left, while the other one is filled up first—thus it has to be eliminated or removed the old content for the new content.

## 3.2. Unified Memory Manager

The Unified Memory Manager mechanism was introduced after Spark 1.6. The difference between Unified Memory Manager and Static Memory Manager is that under the Unified Memory Manager mechanism, the Storage memory and Execution memory share a memory area, and both can occupy each other's free area.

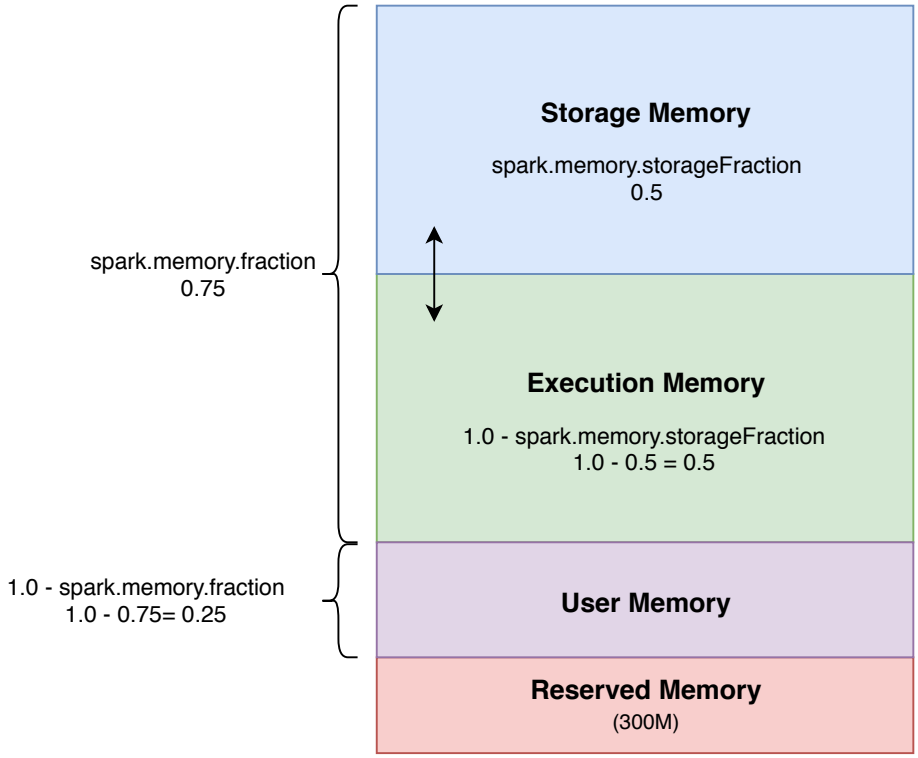
### 3.2.1. On-heap model

By default, Spark uses On-heap memory only. The size of the On-heap memory is configured by the `–executor-memory` or `spark.executor.memory` parameter when the Spark Application starts. The concurrent tasks running inside Executor share JVM's On-heap memory.

The On-heap memory area in the Executor can be roughly divided into the following four blocks:

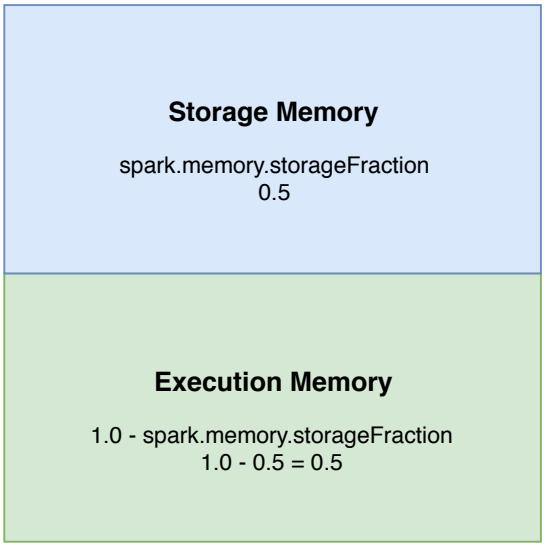
- Storage Memory: It's mainly used to store Spark cache data, such as RDD cache, Broadcast variable, Unroll data, and so on.
- Execution Memory: It's mainly used to store temporary data in the calculation process of Shuffle, Join, Sort, Aggregation, etc.
- User Memory: It's mainly used to store the data needed for RDD conversion operations, such as the information for RDD dependency.
- Reserved Memory: The memory is reserved for system and is used to store Spark's internal objects.

The memory distribution is shown below:

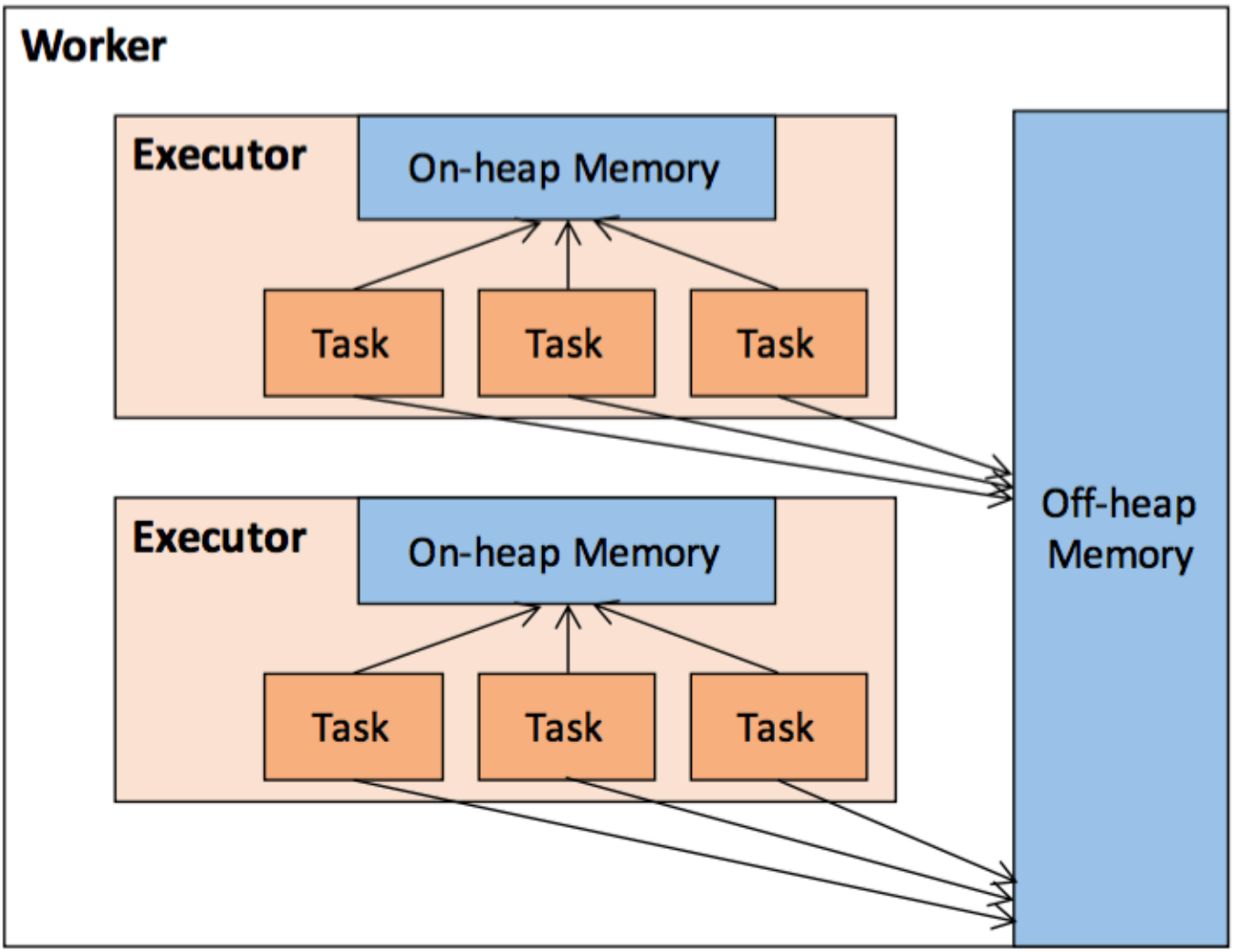



### 3.2.2. Off-heap model

Spark 1.6 began to introduce Off-heap memory (SPARK-11389). By default, Off-heap memory is disabled, but we can enable it by the `spark.memory.offHeap.enabled` parameter, and set the memory size by `spark.memory.offHeap.size` parameter. Compared to the On-heap memory, the model of the Off-heap memory is relatively simple, including only Storage memory and Execution memory, and its distribution is shown in the following picture:




If the Off-heap memory is enabled, there will be both On-heap and Off-heap memory in the Executor. At this time, the Execution memory in the Executor is the sum of the Execution memory inside the heap and the Execution memory outside the heap. The same is true for Storage memory. The following picture shows the on-heap and off-heap memory inside and outside of the Spark heap.




#### 3.2.3. Dynamic occupancy mechanism

- When the program is submitted, the Storage memory area and the Execution memory area will be set according to the `spark.memory.storageFraction` parameter.
- When the program is running, if the space of both parties is not enough (the storage space is not enough to put down a complete block), it will be stored to the disk according to LRU; if one of its space is insufficient but the other is free, then it will borrow the other's space .
- Storage occupies the other party's memory, and transfers the occupied part to the hard disk, and then "return" the borrowed space.
- Execution occupies the other party's memory, and it can't make to "return" the borrowed space in the current implementation. Because the files generated by the Shuffle process will be used later, and the data in the Cache is not necessarily used later, returning the memory may cause serious performance degradation.

# 4. Reference

- Difference between "on-heap" and "off-heap"
- Spark Memory Management



# 0 Comment