Apache Spark unified memory management model Detailed

This article will Spark memory management model to analyze all of the following analysis is based on Apache Spark 2.2.1 conducted. In order for the following article does not look boring, I do not intend to put the code level things. Articles only unified memory management module (U nifiedMemoryManager) for analysis, such as static memory management before interest, see other articles online.

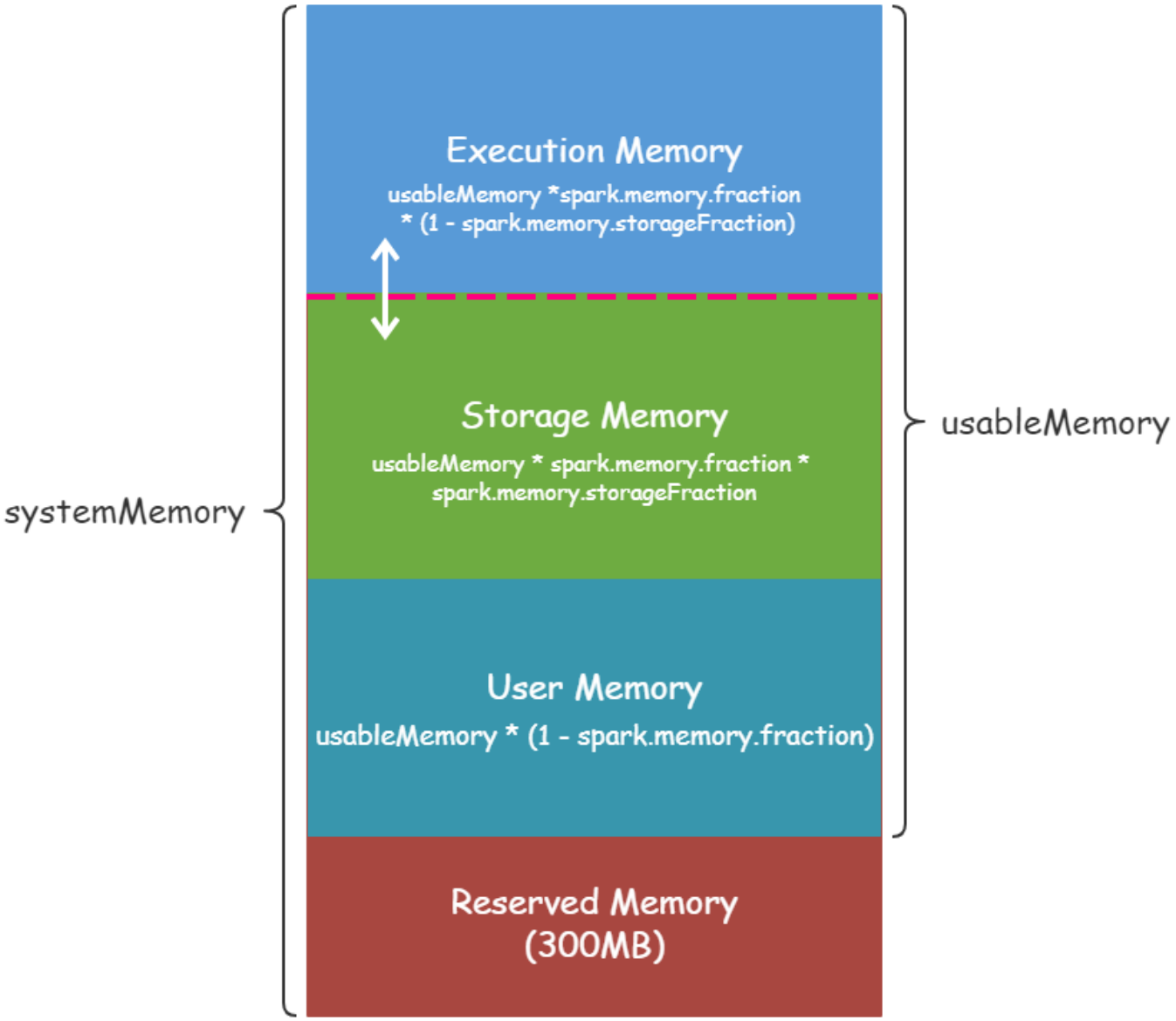
We all know that Spark efficient use of memory and can be distributed computing, its memory management module plays a very important role in the overall system. To make better use
Spark, in-depth understanding of its memory management model has very important significance, which helps us to better tune Spark; in the event of a variety of memory problems, be able to find out the mind, to find piece of memory problem areas. Hereinafter all referring to the presentation memory model Executor end memory model, Driver
Memory model does not describe the end of this article. Unified memory management module included in the heap memory (On-heap
Memory) memory and heap outside the two regions (Off-heap Memory), these two regions following detailed description

Within heap memory (On-heap Memory)

By default, Spark used only within the heap memory. Executor end of the heap
memory area can be divided into the following four blocks:

- **Execution memory** : Mainly used to store temporary data calculation process Shuffle, Join, Sort, Aggregation, etc.
- **Memory Storage** : Mainly used to cache data storage spark, e.g. RDD buffer, unroll By transactions;
- **User memory (User Memory)** : Mainly used for storing data required for the operation converter RDD, e.g. RDD dependence information.
- **Memory reserved (Reserved Memory)** : Reserved for system memory, Spark will be used to store internal objects.

Throughout the heap if Executor end represented by FIG., It can be summarized as follows:



If you want a timely manner

Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

We conducted the following description of the figure:

- systemMemory = Runtime.getRuntime.maxMemory, in fact, the parameter spark.executor.memory or --executor-memory configuration. reservedMemory Spark 2.2.1 is written in the death of a value equal
- to 300MB, this value can not be changed (if in a test environment, we can modify the parameters by spark.testing.reservedMemory);

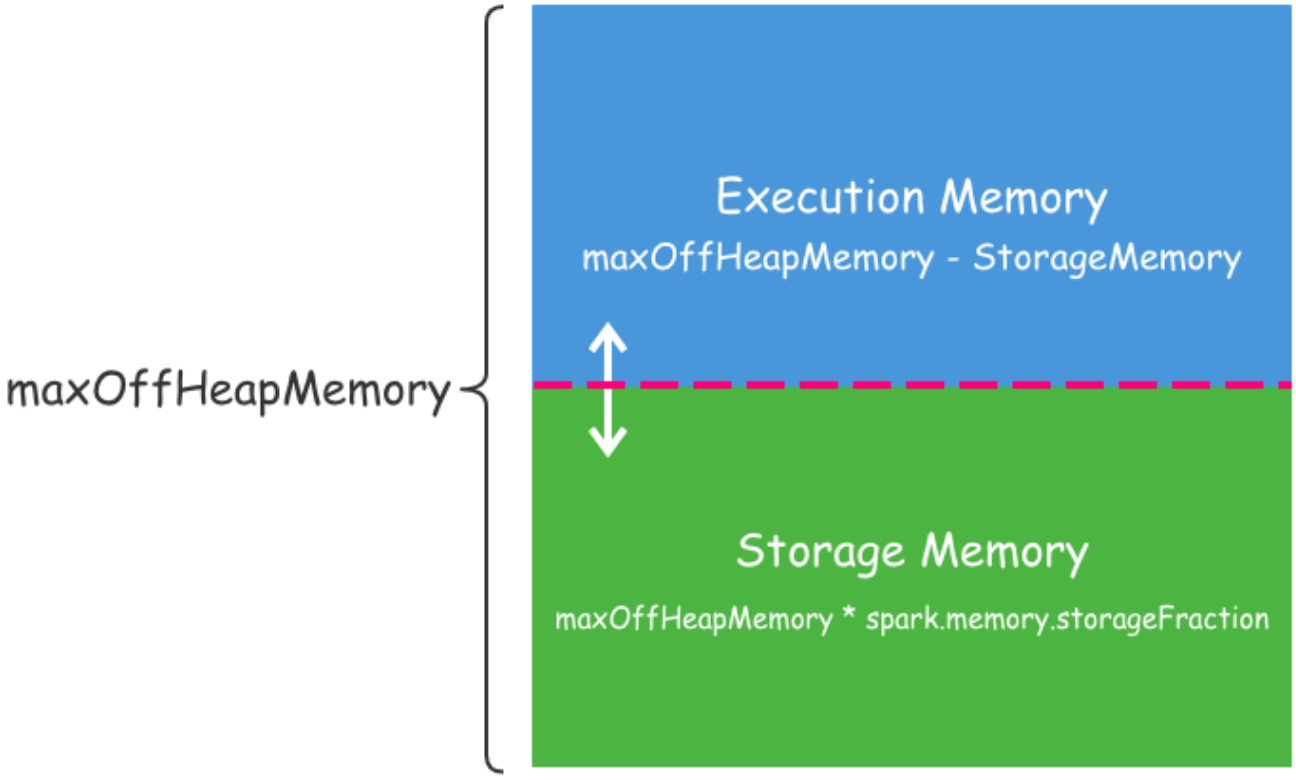
- usableMemory = systemMemory - reservedMemory, the available memory is Spark;

External memory heap (Off-heap Memory)

Spark 1.6 began to introduce the Off-heap memory (see SPARK-11389). This model does not apply within the JVM memory, but call the Java API for the unsafe related language such as C inside malloc () directly to the operating system for memory, since this method is not been to the JVM memory management, so avoid frequent GC the disadvantage of this application is that memory must write their own logic and memory applications release.

By default, the external memory heap is off, we can enable through spark.memory.offHeap.enabled parameters, and by spark.memory.offHeap.size

Disposed outside the heap memory size, in bytes. If the external memory heap is enabled, it will simultaneously exist in the Executor within heap and stack memory outside, using complementary effects of the two, this time in the Execution Executor memory is outside the Execution of memory in the heap and stack memory of Execution and, with Li, Storage memory, too. Compared to in-memory heap, the heap memory, external memory and Storage Execution only to distinguish between the memory that the memory allocation as shown below:



If you want a timely manner

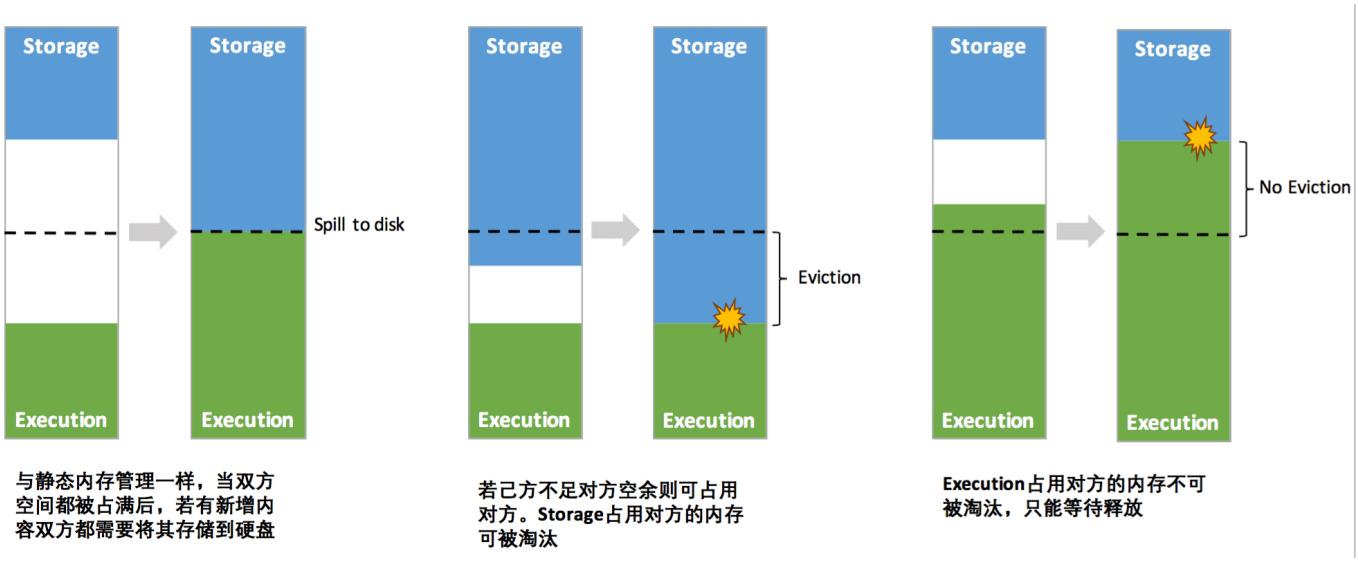
Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

The image above *maxOffHeapMemory* equal *spark.memory.offHeap.size* Parameter configuration.

Execution dynamically adjust memory and Storage memory

Careful students certainly see that there is a dotted line between the above two pictures of Execution memory and Storage memory, which is why?

Used Spark students should know before Spark 1.5, Execution memory and Storage memory allocation is static, in other words, if insufficient Execution memory, Storage memory, even if there is not a great free program to use; and vice versa. This leads us to difficult memory tuning work, we must be very clear understanding of the distribution of Execution and Storage memory two regions. At present, Execution memory and Storage memory can be shared with each other. That is, if insufficient Execution memory, Storage memory is idle, you can apply for Execution from Storage space in; and vice versa. Therefore, the figure above the dashed line represents Execution and Memory Storage memory can be dynamically adjusted with the operation, so that memory resources can be effectively utilized. Dynamically adjusting between Execution and Memory Storage memory can be summarized as follows:



If you want a timely manner

Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

Specific implementation logic is as follows:

- When the program is submitted we will set the basic Execution memory and Storage memory area (by spark.memory.storageFraction parameter setting);
- The program is running, if both sides of the space is insufficient, then stored in the hard disk; the storage memory block to disk LRU policy is conducted in accordance with the rules. If one's own lack of space and other spare time, they can borrow each other's space; (lack of storage space is not sufficient to lay down a complete Block) Execution
- After the memory space is occupied by the other side, the other side can dump will occupy part of the hard disk, and then "return" after
- borrowing the memory space Storage space is occupied the other side, the current implementation is not to let the other side "return" because of the need to consider Shuffle the course of a number of factors are complex to implement; and Shuffle
  File process produces will be used to in the back, and Cache data in memory is not necessarily use later.

Note that the above-borrowing each other's memory needs of the lender and the lender of memory types are the same, are in-memory heap or are outside the heap memory, heap memory is not enough space in the external memory heap to borrow does not exist.

Memory distribution between Task

In order to make better use of the use of memory, shared memory between the Execution Task running within Executor. Specifically, Spark

Internal memory maintains a record for each Task HashMap occupied. When the need to apply Task Execution numBytes memory area of memory, which first determines whether there HashMap Task maintains the memory usage, if not, the memory usage of this Task is set to 0, and in TaskId as key, memory usage value added to HashMap inside. After application of this Task numBytes memory, if there is just larger than the memory area Execution numBytes free memory, which is currently in the HashMap Task memory plus numBytes used, then return; Execution If the current application to each memory area can not be minimized Task memory applications, the current Task

It is blocked until some other task releases enough memory to perform the task before they can be awakened. Each Task Execution memory size may be used in the range of 1 / 2N ~ 1 / N, where N is the number of the current Task Executor running. Task can run a minimum amount of memory is required to apply to the (1 / 2N * Execution memory); the time when N = 1, Task Execution can use all memory.

Execution memory such as memory size range if 10GB is, the number of the current Task Executor 5 is running, the Task can filed 10 / (2 * 5) - 10/5, i.e. in the range of 1GB ~ 2GB.

An example

For a better understanding to the above in-memory heap and the heap memory usage, here is a simple example.

Only within the heap memory

Now Spark jobs we submitted on memory configuration is as follows:

--executor-memory 18g

Since no spark.memory.fraction and spark.memory.storageFraction parameters, we can see Spark UI displayed on Storage Memory as follows:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	123.206.77.132:33618	Active	0	0.0 B / 429.1 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump
1	www.iteblog.com:50956	Active	0	0.0 B / 10.1 GB	0.0 B	1	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump

Showing 1 to 2 of 2 entries Previous 1 Next

If you want a timely manner

Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

FIG clearly see the Storage Memory is memory available

10.1GB, this number is a loud noise come from? According to the previous rules, we can draw the following calculation:

systemMemory = spark.executor.memory
reservedMemory = 300MB
usableMemory = systemMemory - reservedMemory
StorageMemory = usableMemory * spark.memory.fraction * spark.memory.storageFraction

If we go in on behalf of the data, we obtained the following results:

systemMemory = 18Gb = 19327352832 bytes
reservedMemory = 300MB = 300 * 1024 * 1024 = 314572800
usableMemory = systemMemory - reservedMemory = 19327352832 - 314572800 = 190127800 32

StorageMemory = usableMemory * spark.memory.fraction * spark.memory.storageFraction
= 19012780032 * 0.6 * 0.5 = 5703834009.6 = 5.312109375GB

Not, ah, and above 10.1GB not on the ah. why? This is because the display of Spark UI on top of Storage Memory available memory is actually equal to the Execution memory and Storage memory sum, that is, *usableMemory * spark.memory.fraction* :

StorageMemory = usableMemory * spark.memory.fraction
= 19012780032 * 0.6 = 11407668019.2 = 10.62421GB

Right or wrong, because although we set up - *executor-memory 18g* However Executor end by Spark *Runtime.getRuntime.maxMemory* Get the memory actually not so large, only 17,179,869,184 bytes, *systemMemory = 17179869184* , The data is then calculated as follows:

systemMemory = 17179869184 bytes
reservedMemory = 300MB = 300 * 1024 * 1024 = 314572800
usableMemory = systemMemory - reservedMemory = 17179869184 - 314572800 = 168652963 84

StorageMemory = usableMemory * spark.memory.fraction
= 16865296384 * 0.6 = 9.42421875 GB

Through the above 16,865,296,384 bytes divided by 0.6 * 1024 * 1024 * 1024 into display on 9.42421875 GB, and the UI still do not, because in addition to Spark UI through 1000 * 1000 * 1000 byte conversion to make

GB, as follows:

systemMemory = 17179869184 bytes
reservedMemory = 300MB = 300 * 1024 * 1024 = 314572800
usableMemory = systemMemory - reservedMemory = 17179869184 - 314572800 = 168652963 84

StorageMemory = usableMemory * spark.memory.fraction
* 0.6 = 16,865,296,384 bytes = 16865296384 * 0.6 / (1000 * 1000 * 1000) = 10.1GB

Now finally on the up.

The byte into a specific computation logic GB follows (core module below
/core/src/main/resources/org/apache/spark/ui/static/utils.js) :

```
function formatBytes (bytes, type) {
    if (type == 'display'!) return bytes; if (bytes
    == 0) return '0.0 B'; var k = 1000; var dm =
    1;

    var sizes = [ 'B', 'KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']; var i =
    Math.floor (Math. log (bytes) / Math.log (k));
    return parseFloat (. (bytes / Math.pow (k, i)) toFixed () dm) + '' + sizes [i];}
```

We set up - *executor-memory 18g* However Executor end by Spark
Runtime.getRuntime.maxMemory Get the memory actually not so large, only 17,179,869,184 bytes, this data is how
calculated?
Runtime.getRuntime.maxMemory Is the largest program memory can be used, its value will be less than the actual value of the actuator memory configuration.
This is because part of the memory heap allocation pool is divided into Eden, Survivor and Tenured space of three parts, which is inside contains a total of two
Survivor areas, and these two Survivor areas at any time, we can only use one of them, so we can be described using the following formula:

ExecutorMemory = Eden + 2 * Survivor + Tenured
Runtime.getRuntime.maxMemory = Eden + Survivor + Tenured

The above 17,179,869,184 bytes may be because your GC
The same configuration is not obtained data are not the same, but the formulas are the same as above.

With the inner and outer reactor heap memory

Now if we enable external heap memory, the situation Zeyang it? Our memory-related configuration is as follows:

spark.executor.memory 18g
spark.memory.offHeap.enabled true
spark.memory.offHeap.size 10737418240

As can be seen from the above, the outer heap memory is 10GB, now Spark UI shown above Storage Memory memory available

20.9GB, as follows:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	(GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	123.206.77.132:48167	Active	0	0.0 B / 11.2 GB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump
1	www.iteblog.com:51908	Active	0	0.0 B / 20.9 GB	0.0 B	1	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
2	www.iteblog.com:56932	Active	0	0.0 B / 20.9 GB	0.0 B	1	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
3	www.iteblog.com:43380	Active	0	0.0 B / 20.9 GB	0.0 B	1	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump

If you want a timely manner

Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

In fact Storage Memory Spark UI shown above

Available memory equal to the in-memory heap and the heap memory of the outer and calculated as follows:

Within the heap

systemMemory = 17179869184 bytes
reservedMemory = 300MB = 300 * 1024 * 1024 = 314572800
usableMemory = systemMemory - reservedMemory = 17179869184 - 314572800 = 168652963 84

totalOnHeapStorageMemory = usableMemory * spark.memory.fraction
= 0.6 = 10,119,177,830 16,865,296,384 *

Heap outside

totalOffHeapStorageMemory = spark.memory.offHeap.size = 10737418240

StorageMemory = totalOnHeapStorageMemory + totalOffHeapStorageMemory
= (10,119,177,830 + 10,737,418,240) bytes =
(20856596070 / (1000 * 1000 * 1000)) GB = 20.9 GB