**Spark性能调优**

在大数据性能的调优，它的本质是硬件的调优！即基于 **CPU(计算)、Memory(存储)、IO-Disk/ Network(数据交互)** 基础上构建算法和性能调优！我们在计算的时候，数据肯定是存储在内存中的。磁盘IO怎么去处理和网络IO怎么去优化。

在大数据性能本质的思路上，我们应该需要在那些方面进行调优呢？比如：

* 并行度
* 压缩
* 序例化
* 数据倾斜
* JVM调优 (例如 JVM 数据结构化优化)
* 内存调优
* Task性能调优 (例如包含 Mapper 和 Reducer 两种类型的 Task)
* Shuffle 网络调优 (例如小文件合并)
* RDD 算子调优 (例如 RDD 复用、自定义 RDD)
* 数据本地性
* 容错调优
* 参数调优

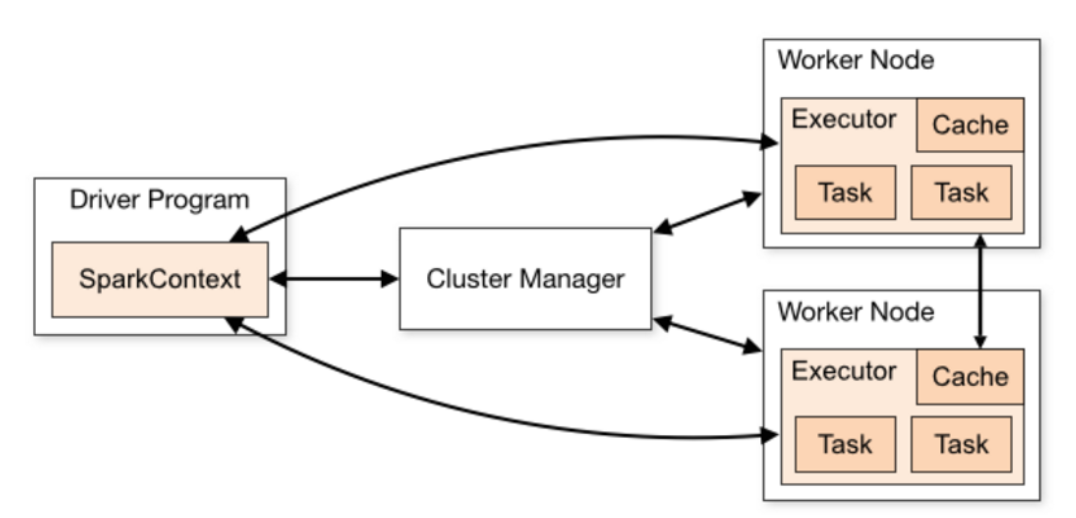
调优 Spark 的重点一定是在数据本地性和数据倾斜入手。

* 资源分配和使用：你能够申请多少资源以及如何最优化的使用计算资源
* 关发调优：如何基于 Spark 框架内核原理和运行机制最优化的实现代码功能
* Shuffle调优：分布式系统必然面临的杀手级别的问题
* 数据倾斜：分布式系统业务本身有数据倾斜

## **Spark 资源使用原理流程**

这里有三大组件，Driver 部份，具体处理数据的部份，资源管理部份。这一张图中间有一个过程，这表示在程序运行之前向资源管理器申请资源。在实际生产环境中，Cluster Manager 一般都是 Yarn 的 ResourceManager，Driver 会向 ResourceManager 申请计算资源(一般情况下都是在发生计算之前一次性进行申请请求)，分配的计算资源就是 CPU Core 和 Memory，我们具体的 Job 里的 Task 就是基于这些分配的内存和 Cores 构建的线程池来运行 Tasks 的。

[下图是 Spark 官方网站上的经典Spark架框图]



当然在 Task 运行的过程中会大量的消耗内存，而Task又分为 Mapper 和 Reducer 两种不同类型的 Task，也就是 ShuffleMapTask 和 ResultTask 两种类型，这类有一个很关建的调优点就是**如何对内存进行使用**。在一个 Task 运行的时候，默应会占用 Executor 总内存的 20％，Shuffle 拉取数据和进行聚合操作等占用了 20% 的内存，剩下的大概有 60% 是用于 RDD 持久化 (例如 cache 数据到内存)，Task 在运行时候是跑在 Core 上的，比较理想的是有足够的 Core 同时数据分布比较均匀，这个时候往往能够充分利用集群的资源。

**核心调优参数如下：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | num-executors  executor-memory  executor-cores  driver-memory  spark.default.parallelizm  spark.storage.memoryFraction  spark.shuffle.memoryFraction |

* num-executors：该参数一定会被设置，Yarn 会按照 Driver 的申请去最终为当前的 Application 生产指定个数的 Executors，实际生产环境下应该分配80个左右 Executors 会比较合适呢。
* executor-memory：这个定义了每个 Executor 的内存，它与 JVM OOM 紧密相关，很多时候甚至决定了 Spark 运行的性能。实际生产环境下建义是 8G 左右，很多时候 Spark 运行在 Yarn 上，内存占用量不要超过 Yarn 的内存资源的 50%。
* executor-cores：决定了在 Executors 中能够并行执行的 Tasks 的个数。实际生产环境下应该分配4个左右，一般情况下不要超过 Yarn 队列中 Cores 总数量的 50％。
* driver-memory：默应是 1G
* spark.default.parallelizm：并行度问题，如果不设置这个参数，Spark 会跟据 HDFS 中 Block 的个数去设置这一个数量，原理是默应每个 Block 会对应一个 Task，默应情况下，如果数据量不是太多就不可以充份利用 executor 设置的资源，就会浪费了资源。建义设置为 100个，最好 700个左右。Spark官方的建义是每一个 Core 负责 2-3 个 Task。
* spark.storage.memoryFraction：默应占用 60％，如果计算比较依赖于历史数据则可以调高该参数，当如果计算比较依赖 Shuffle 的话则需要降低该比例。
* spark.shuffle.memoryFraction：默应占用 20％，如果计算比较依赖 Shuffle 的话则需要调高该比例。

## **Spark 更高性能的算子**

Shuffle 分开两部份，一个是 Mapper 端的Shuffle，另外一个就是 Reducer端的 Shuffle，性能调优有一个很重要的总结就是尽量不使用 Shuffle 类的算子，我们能避免就尽量避免，因为一般进行 Shuffle 的时候，它会把集群中多个节点上的同一个 Key 汇聚在同一个节点上，例如 reduceByKey。然后会优先把结果数据放在内存中，但如果内存不够的话会放到磁盘上。Shuffle 在进行数据抓取之前，为了整个集群的稳定性，它的 Mapper 端会把数据写到本地文件系统。这可能会导致大量磁盘文件的操作。如何避免Shuffle可以考虑以下：

1、采用 Map 端的 Join (RDD1 + RDD2 )先把一个 RDD1的数据收集过来，然后再通过 **sc.broadcast( )** 把数据广播到 Executor 上；

2、如果无法避免Shuffle，退而求其次就是需要更多的机器参与 Shuffle 的过程，这个时候就需要充份地利用 Mapper 端和 Reducer 端机制的计算资源，尽量使用 Mapper 端的 Aggregrate 功能，e.g. aggregrateByKey 操作。相对于 groupByKey而言，更倾向于使用**reduceByKey( )** 和 **aggregrateByKey( )** 来取代 groupByKey，因为 groupByKey 不会进行 Mapper 端的操作，aggregrateByKey 可以给予更多的控制。

3、如果一批一批地处理数据来说，可以使用 **mapPartitions( )**，但这个算子有可能会出现 OOM 机会，它会进行 JVM 的 GC 操作！

4、如果进行批量插入数据到数据库的话，建义采用**foreachPartition( )** 。

5、因为我们不希望有太多的数据碎片，所以能批量处理就尽量批量处理，你可以调用**coalesce( )** ，把一个更多的并行度的分片变得更少，假设有一万个数据分片，想把它变得一百个，就可以使用 coalesce( )方法，一般在 filter( ) 算子之后就会用 coalesce( )，这样可以节省资源。

6、官方建义使用**repartitionAndSortWithPartitions( )**；

7、数据进行复用时一般都会进行持久化 **persisit( )**；

8、建义使用 **mapPartitionWithIndex( )**；

9、也建义使用 tree 开头的算子，比如说 **treeReduce( )** 和 **treeAggregrate( )**；

**第二篇:shuffle**

人们对Spark的印象往往是基于内存进行计算，但实际上来讲，**Spark可以基于内存、也可以基于磁盘或者是第三方的储存空间进行计算**，背后有两层含意，第一、Spark框架的架构设计和设计模式上是**倾向于在内存中计算数据的**，第二、这也表达了人们对数据处理的一种美好的愿望，**就是希望计算数据的时候，数据就在内存中**。

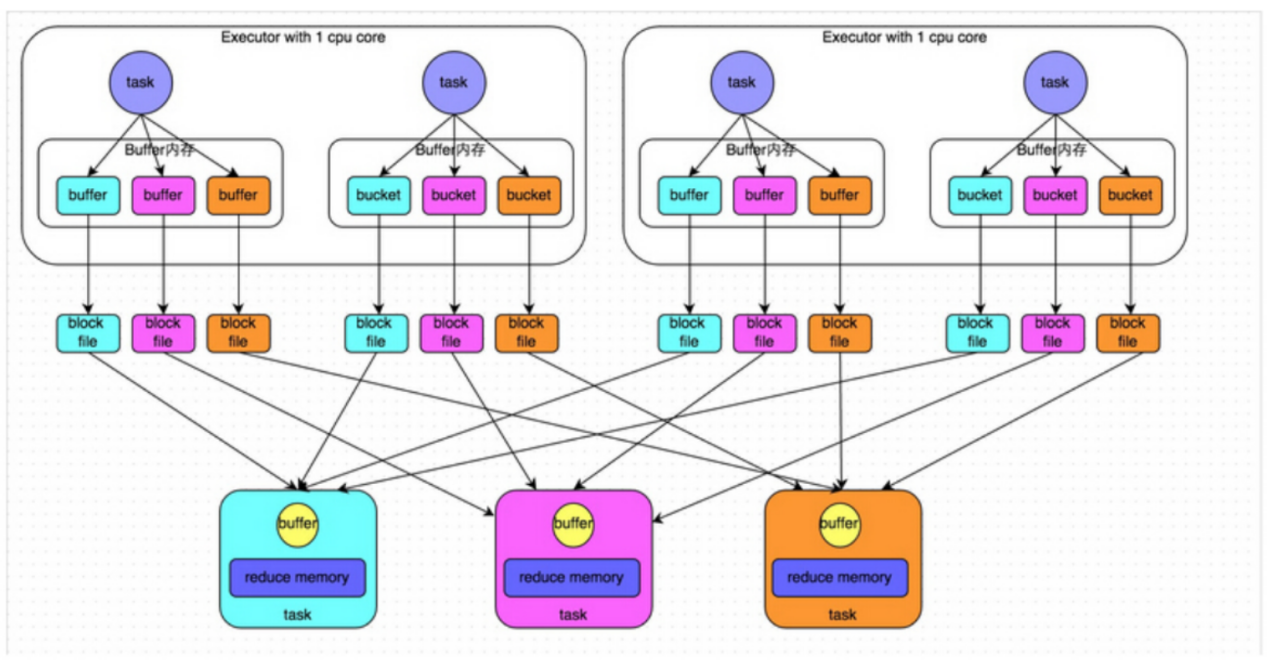
为什么再一次强调 Shuffle 是 Spark 的性能杀手啦，那不就是说，Spark中的 “Shuffle“ 和 “Spark完全是基于内存计算“ 的愿景是相违背的！！！

Spark 运行分成两部份，第一部份是 Driver Program，里面的核心是 SparkContext，它驱动著一个程序的开始，负责指挥，另外一部份是 Worker 节点上的 Task，它是实际运行任务的，当程序运行时，不间断地由 Driver 与所在的进程进行交互，交互什么，有几点，第一、是让你去干什么，第二、是具体告诉 Task 数据在那里，例如说有三个 Stage，第二个 Task 要拿数据，它就会向 Driver 要数据，所以在整个工作的过程中，Executor 中的 Task 会不断地与 Driver 进行沟通，这是一个网络传输的过程。

在这个过程中一方面是 Driver 跟 Executor 进行网络传输，另一方面是Task要从 Driver 抓取其他上游的 Task 的数据结果，所以有这个过程中就不断的产生网络结果。其中，下一个 Stage 向上一个 Stage 要数据这个过程，我们就称之为 Shuffle。

Hash 本身有一套 Hash算法，可以把数据的 Key 进行重新分类，每个 Task 对数据进行分类然后把它们不同类别的数据先写到本地磁盘，然后再经过网络传输 Shuffle，把数据传到下一个 Stage 进行汇聚。

下图有3个 Reducer，从 Task 开始那边各自把自己进行 Hash 计算，分类出3个不同的类别，每个 Task 都分成3种类别的数据，刚刚提过因为分布式的关系，我们想把不同的数据汇聚然后计算出最终的结果，所以下游的 Reducer 会在每个 Task 中把属于自己类别的数据收集过来，汇聚成一个同类别的大集合，抓过来的时候会首先放在内存中，但内存可能放不下，也有可能放在本地 (这也是一个调优点。可以参考上一章讲过的一些调优参数)，**每1个 Task 输出3份本地文件，这里有4个 Mapper Tasks，所以总共输出了4个 Tasks x 3个分类文件 = 12个本地小文件**。

  
 **HashShuffle 也有它的弱点：**

1、Shuffle前在磁盘上会产生海量的小文件，此时会产生大量耗时低效的 IO 操作 (因為产生过多的小文件）

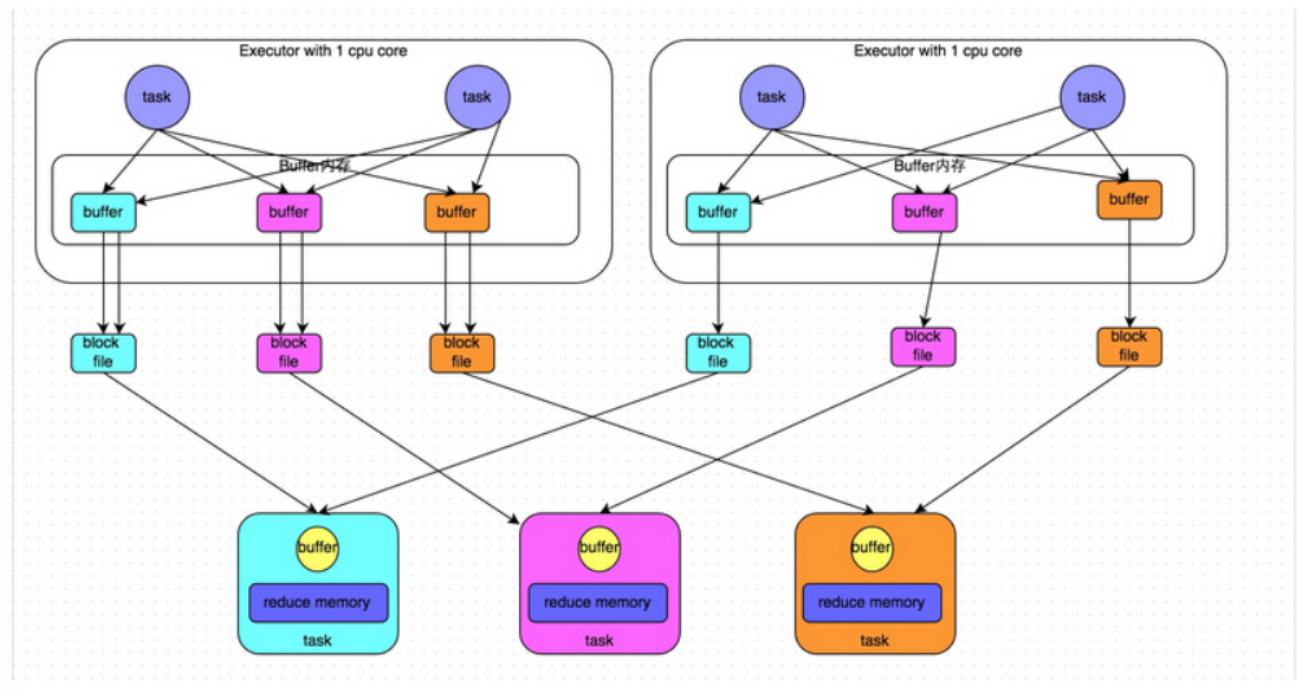
2、内存不够用，由于内存中需要保存海量文件操作句柄和临时信息，如果数据处理的规模比较庞大的话，内存不可承受，会出现 OOM 等问题。

Consoldiated Hash-Shuffle的优化有一个很大的好处就是假设现在有200个Mapper Tasks在同一个进程中，也只会产生3个本地小文件； 如果用原始的 Hash-Based Shuffle 的话，200个Mapper Tasks 会各自产生3个本地小文件，在一个进程已经产生了600个本地小文件。3个对比600已经是一个很大的差异了。

这个优化后的 HashShuffle 叫 ConsolidatedShuffle，在实际生产环境下可以调以下参数：

|  |  |
| --- | --- |
| 1 | spark.shuffle.consolidateFiles=true |

在刚才 HashShuffle 的基础上思考该如何进行优化，这是优化后的实现：

[下图是 Spark Consolidated Hash-Based Shuffle 概念图]  


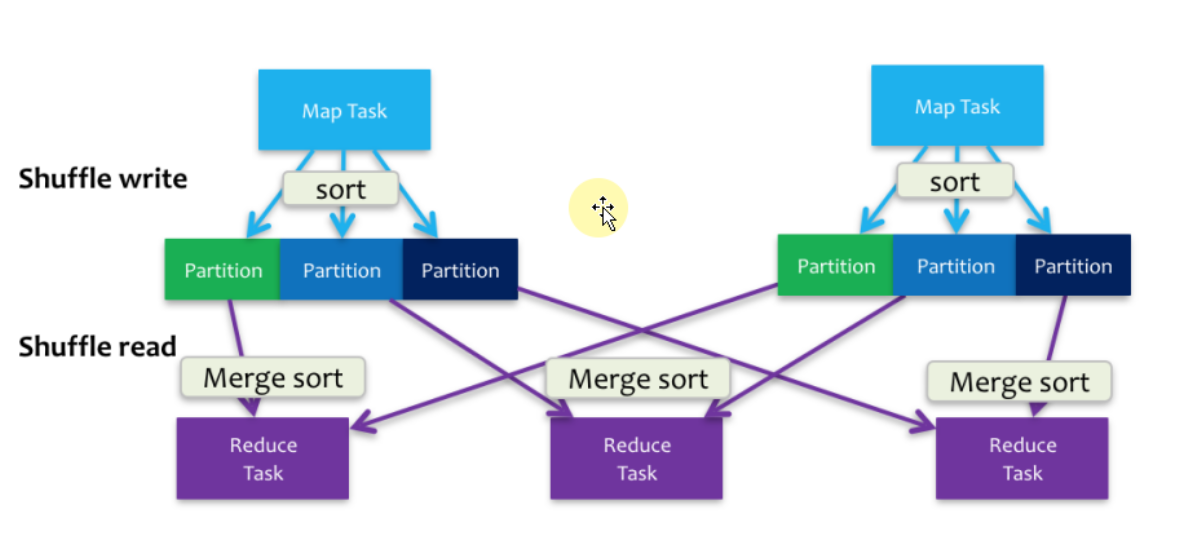
Shuffle 不可以避免是因为在分布式系统中的基本点就是把一个很大的的任务/作业分成一百份或者是一千份，这一百份和一千份文件在不同的机器上独自完成各自不同的部份，我们是针对整个作业要结果，所以在后面会进行汇聚，这个汇聚的过程的前一阶段到后一阶段以至**网络传输**的过程就叫 Shuffle。在 Spark 中为了完成 Shuffle 的过程会把真正的一个作业划分为不同的 Stage，这个Stage 的划分是跟据依赖关系去决定的，Shuffle 是整个 Spark 中最消耗性能的一个地方。试试想想如果没有 Shuffle 的话，Spark可以完成一个纯内存式的操作。

**数据倾斜的问题会引申很多其他问题，比如，网络带宽、各重硬件故障、内存过度消耗、文件掉失。因为 Shuffle 的过程中会产生大量的磁盘 IO、网络 IO、以及压缩、解压缩、序列化和反序列化等等**。

## **Spark Sorted-Based Shuffle介绍**

Sorted-Based Shuffle 的出现，最显著的优势就是把 Spark 从只能处理中小规模的数据平台，变成可以处理无限大规模的数据平台。可能你会问规模真是这么重要吗？当然有，集群规模意为著它处理数据的规模，也意为著它的运算能力。

Sorted-Based Shuffle 不会为每个Reducer 中的Task 生产一个单独的文件，相反Sorted-Based Shuffle 会把Mapper 中每个ShuffleMapTask 所有的输出数据Data 只写到一个文件中，因为每个ShuffleMapTask 中的数据会被分类，所以Sort-based Shuffle 使用了index 文件存储具体ShuffleMapTask 输出数据在同一个Data 文件中是如何分类的信息。所以说基于 Sort-based Shuffle 会在 Mapper 中的每一个 ShuffleMapTask 中产生两个文件 (**并发度的个数 x 2**)！！！



**它会产生一个 Data 文件和一个 Index 文件，其中 Data 文件是存储当前 Task 的 Shuffle 输出的， 而 Index 文件则存储了 Data 文件中的数据通过 Partitioner 的分类信息，此时下一个阶段的 Stage 中的 Task 就是根据这个 Index 文件获取自己所需要抓取的上一个 Stage 中 ShuffleMapTask 所产生的数据**；

**第五大问题：Spark中Sorted-Based Shuffle 数据结果默认是排序的吗？Sorted-Based Shuffle 采用了什么的排序算法？这个排序算法的好处是什么？**  
Spark Sorted-Based Shuffle 在 Mapper 端是排序的，包括 partition 的排序和每个 partition 内部元素的排序！但在 Reducer 端是没有进行排序，所以 Job 的结果默认不是排序的。Sorted-Based Shuffle 采用了 Tim-Sort 排序算法，好处是可以极为高效的使用 Mapper 端的排序成果全局排序

Sorted-Based Shuffle 的核心是**借助于 ExternalSorter 把每个 ShuffleMapTask 的输出排序到一个文件中 (FileSegmentGroup)**，为了区分下一个阶段 Reducer Task 不同的内容，它还需要有一个索引文件 (Index) 来告诉下游 Stage 的并行任务，那一部份是属于你的。

* **ShuffleMapTask**: ShuffleMapTask 在计算具体的 Partition 之后实际上会通过 ShuffleManager 获得的 ShuffleWriter 把当前 Task 计算的数据具体 ShuffleManger 的实现来写入到具体的文件。操作完成后会把 MapStatus 发送给 DAGScheduler； (把 MapStatus 汇报给 MapOutputTracker)
* **ResultTask**: 根据前面 Stage 的执行结果进行 Shuffle 产生整个 Job 最后的结果；(MapOutputTracker 會把 ShuffleMapTask 執行結果交給 ResultTask)

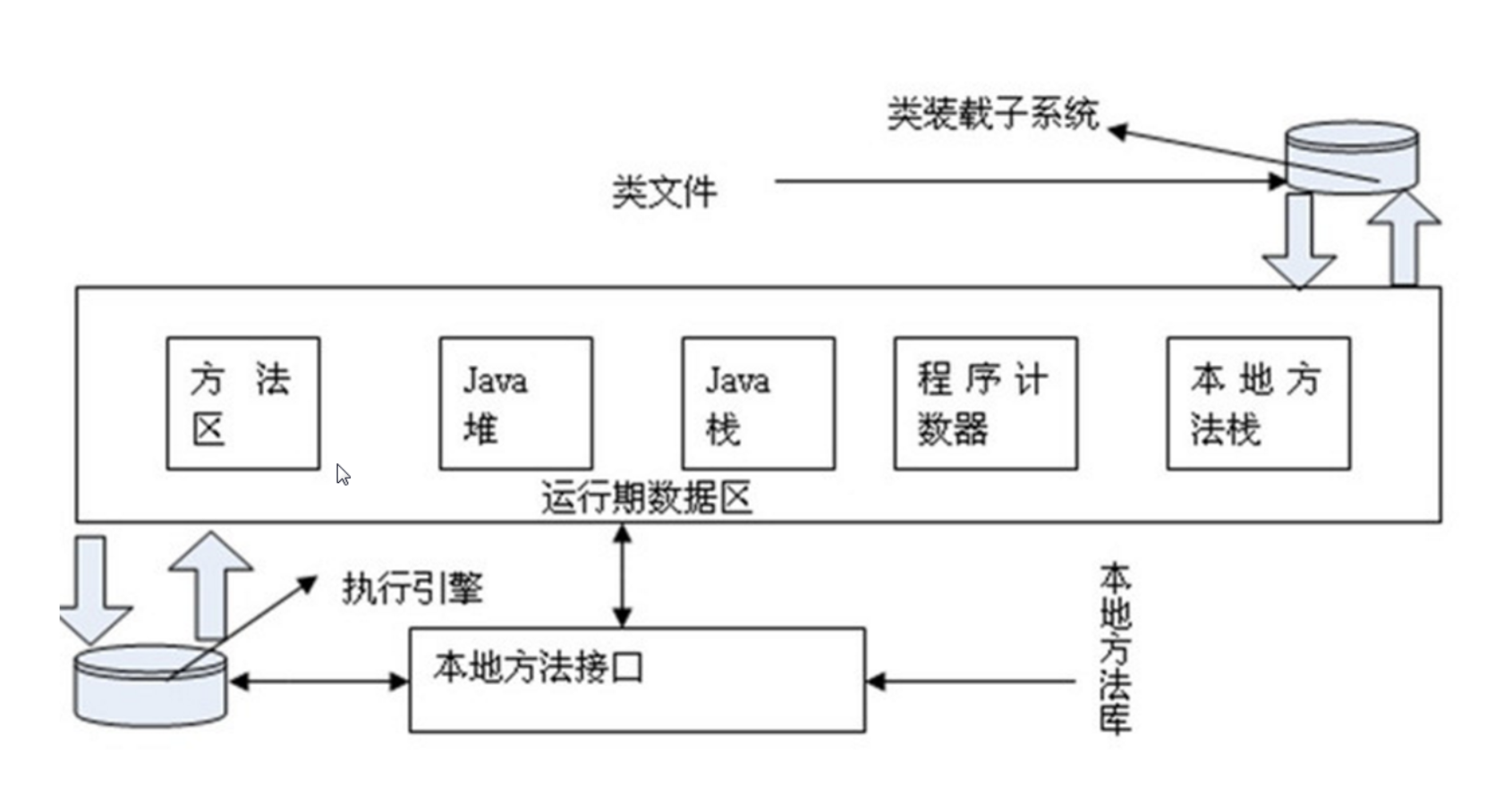
在 Application 启动的时候会在 spark-env.sh 中注册 BlockMangerMaster 以及 MapOutputTracker，其中：

* BlockManagerMaster：对整集群的 Block 数据进行管理；
* MapOutputTracker：跟踪所有的 Mapper 的输出；

1、CacheManager 管理的是**缓存中的数据**，缓存可以是基于内存的缓存，也可以是基于磁盘的缓存；

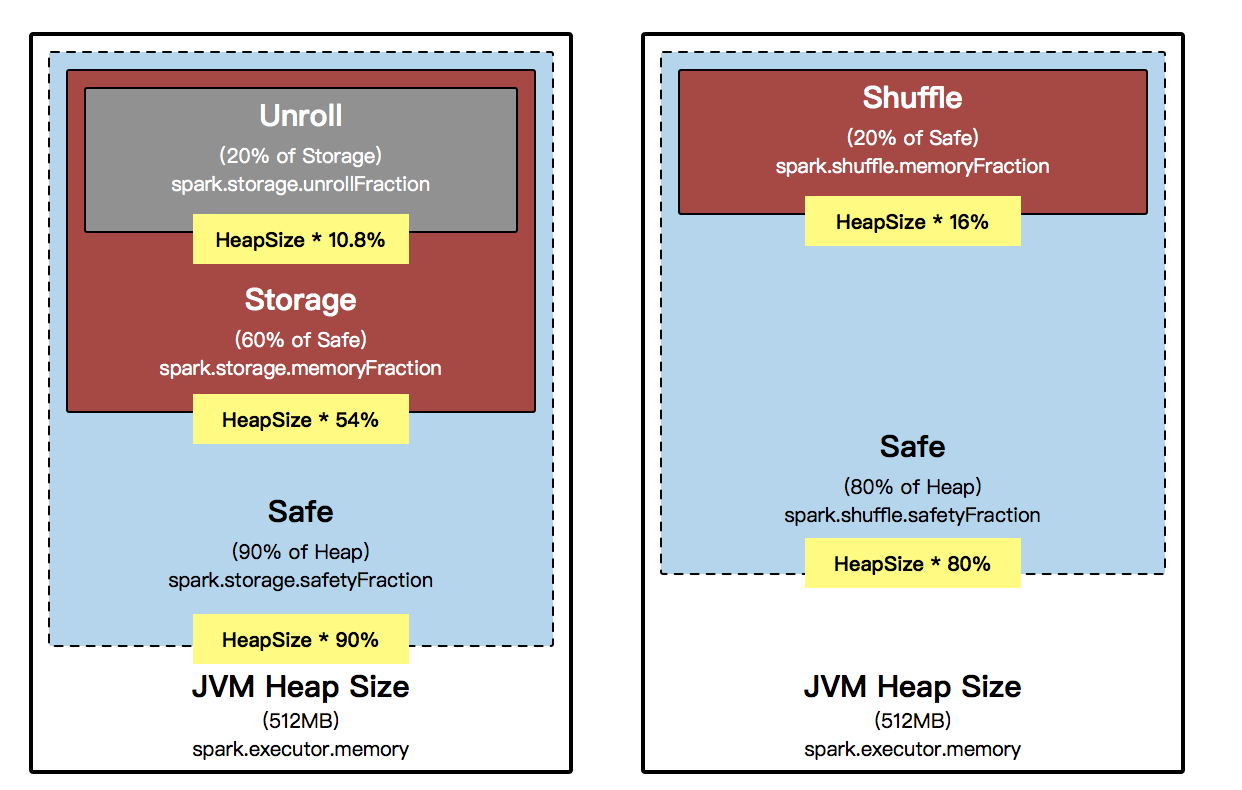
2、CacheManager 需要通过 BlockManager 来操作数据

JVM 有很多不同的区，最开始的时候，它会通过类装载器把类加载进来，在运行期数据区中有 **"本地方法栈"，"程序计数器"，"Java 栈"、"Java 堆"和"方法区"**以及**本地方法接口**和它的**本地库**。从 Spark 的角度来谈代码的运行和数据的处理，主要是谈 Java 堆 (Heap) 空间的运用。

[下图是JVM 内存架构图]  


* **本地方法栈**：这个是在迭归的时候肯定是至关重要的；
* **程序计数器**：这是一个全区计数器，对于线程切换是至关重要的；
* **Java 栈 (Stack)**：**Stack 区属于线程私有**，高效的程序一般都是并发的，每个线程都会包含一个 Stack 区域，Stack 区域中含有**基本的数据类型以及对象的引用**，其它线程均不能直接访问该区域；Java 栈分为三大部份：基本数据类型区域、操作指令区域、上下文等；
* **Java 堆 (Heap)**：存储的全部都是 Object 对象实例，对象实例中一般都包含了其数据成员以及与该对象对应类的信息，它会指向类的引用一个，不同线程肯定要操作这个对象；一个 JVM 实例在运行的时候只有一个 Heap 区域，而且**该区域被所有的线程共享**；补充说明：垃圾回收是回收堆 (heap) 中内容，**堆上才有我们的对象**。
* **方法区**：又名静态成员区域，包含整个程序的 class、static 成员等，类本身的字节码是静态的；它会**被所有的线程共享和是全区级别的**；

下图显示的是Spark 1.6.x 以前版本对 Java 堆 (heap) 的使用情况，左则是 Storage 对内存的使用，右则是 Shuffle 对内存的使用，这叫 **StaticMemoryManagement**，数据处理以及类的实体对象都存放在 JVM 堆 (heap) 中。



JVM Heap 默认情况下是 512MB，这是取决于 **spark.executor.memory** 的参数

**spark.storage.safetyFaction：安全空间（默认对内存90%，包括storage/unroll/shuffle空间）**

**spark.storage.memoryFraction: 缓存空间=spark.executor.memory x spark.storage.safetyFraction x spark.storage.memoryFraction** Heap Size x 90% x 60%

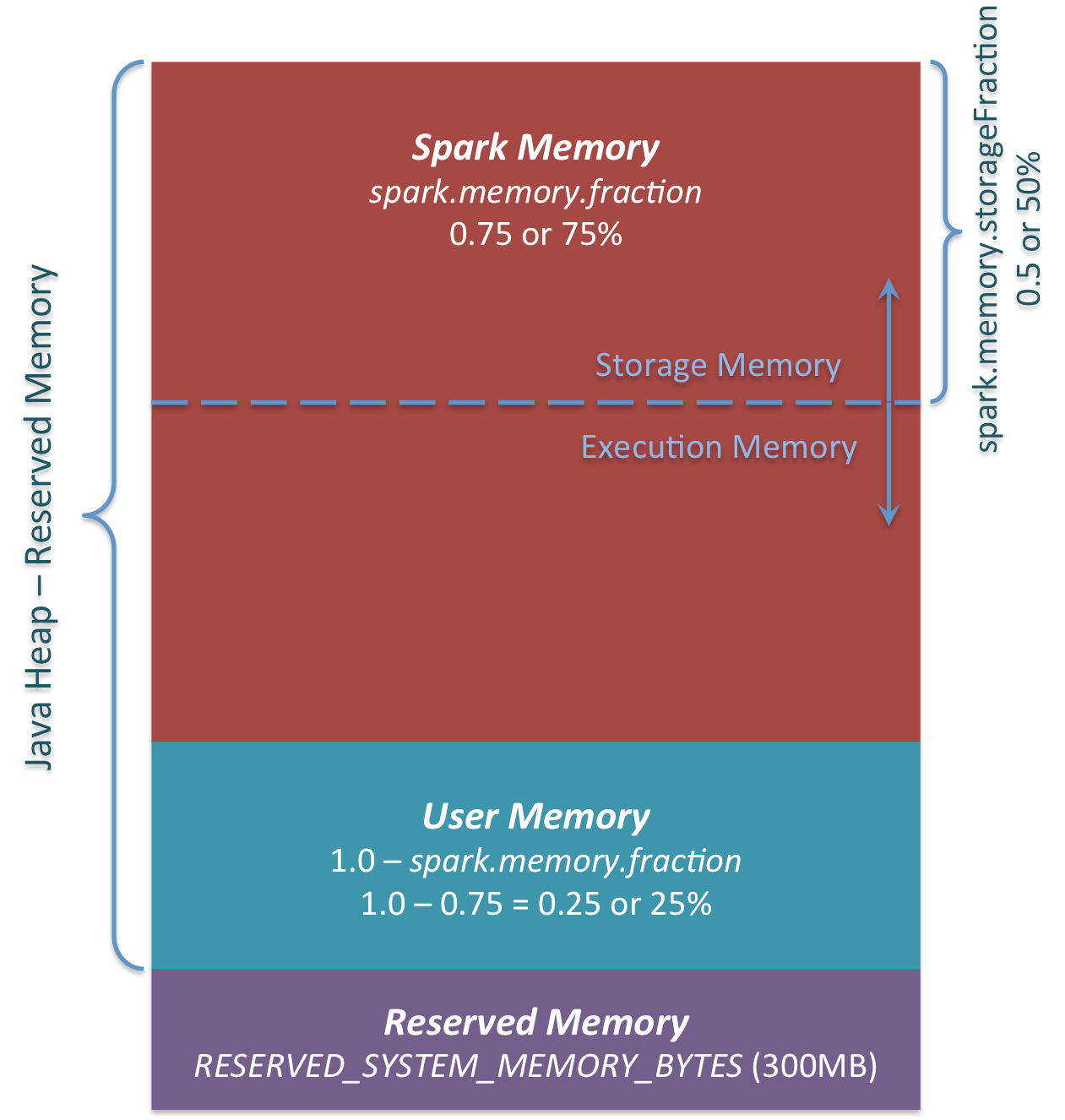
**spark.storage.unrollFraction：unroll空间，进行反序列化=spark.executor.memory x spark.storage.safetyFraction x spark.storage.memoryFraction x spark.storage.unrollFraction**  
也就是 Heap Size x 90% x 60% x 20%；**Heap Size x 10.8%**

**Shuffle 空间**：计算公式是 **spark.executor.memory x spark.shuffle.memoryFraction x spark.shuffle.safteyFraction。**在 Shuffle 空间中也会有一个默认 80％ 的安全空间比例，所以应该是 Heap Size x 20% x 80%；**Heap Size x 16%**，

### **Spark Unified Memory 原理和运行机制**

下图是一种叫联合内存 (Spark Unified Memeory)，数据缓存与数据执行之间的内存可以相互移动，这是一种更弹性的方式，下图显示的是 Spark 2.x 版本对 Java 堆 (heap) 的使用情况，数据处理以及类的实体对象存放在 JVM 堆 (heap) 中。

[下图是 Spark 2.x 版本对 JVM 堆 Storage 和 Execution 的使用分布]



**Spark 2.x 版本对 JVM 堆的使用**

Spark 2.1.0 新型 JVM Heap 分成三个部份：Reserved Memory、User Memory 和 Spark Memory。

**1、Reserved Memory**：默认都是300MB，这个数字一般都是固定不变的，在系统运行的时候 Java Heap 的大小至少为 **Heap Reserved Memory x 1.5**. e.g. 300MB x 1.5 = 450MB 的 JVM配置。一般本地开发例如说在 Windows 系统上，建义系统至少 2G 的大小。

**2、User Memory**：写 Spark 程序中产生的临时数据或者是自己维护的一些数据结构也需要给予它一部份的存储空间，你可以这么认为，这是程序运行时用户可以主导的空间，叫用户操作空间。它占用的空间是 **(Java Heap - Reserved Memory) x 25%**(默认是25％，可以有参数供调优)，这样设计可以让用户操作时所需要的空间与系统框架运行时所需要的空间分离开。

一个 Task 内部所有算子使用的数据空间的大小，处理的数据大小

1. **Spark Memeory**：系统框架运行时需要使用的空间，这是从两部份构成的，分别是 Storage Memeory 和 Execution Memory。现在 Storage 和 Execution (Shuffle) 采用了 Unified 的方式共同使用了 (Heap Size - 300MB) x 75%，默认情况下 Storage 和 Execution 各占该空间的 50%。

* **Storage Memeory**：相当于旧版本的 Storage 空间，在旧版本中 Storage 占了 54% 的 Heap 空间，这个空间会负责存储 **Persist、Unroll 以及 Broadcast 的数据**。假设 Executor 有 4G 的大小，那么 Storage 空间是：(4G - 300MB) x 75% x 50% = 1423.5MB 的空间，也就是说如果你的内存够大的话，你可以扩播足够大的变量，扩播对于性能提升是一件很重要的事情，因为它所有的线程都是共享的。从算子运行的角度来讲，Spark 会倾向于数据直接从 Storgae Memeory 中抓取过来，这也就所谓的内存计算。
* **Execution Memeory**：相当于旧版本的 Shuffle 空间，这个空间会负责存储 **ShuffleMapTask 的数据。**比如说从上一个 Stage 抓取数据和一些聚合的操作、等等**。**在旧版本中 Shuffle 占了 16％ 的 Heap 空间。Execution 如果空间不足的情况下，除了选择向 Storage Memory 借空间以外，也可以把一部份数据 Spill 到磁盘上，但很多时候基于性能调优方面的考虑都不想把数据 Spill 到磁盘上。思考题：你觉得是 Storgae 空间或者是 Execution 空间比较重要呢？