打造价值40万Offer的朋友圈
数据人的宝藏朋友圈
欢迎邀请你的同事同学参观

公众号：import_bigdata
知　乎：https://www.zhihu.com/people/wang-zhi-wu-66
CSDN ：https://blog.csdn.net/u013411339
Github：https://github.com/wangzhiwubigdata/God-of-Bigdata

本文已经加入「大数据成神之路PDF版」中提供下载，你可以关注公众号，后台回复：「**PDF**」即可获取。

更多PDF下载可以参考：**《重磅,大数据成神之路PDF可以分类下载啦!》**

Spark重点难点系列：

- **《【Spark重点难点01】你从未深入理解的RDD和关键角色》**
- **《【Spark重点难点02】你以为的Shuffle和真正的Shuffle》**
- **《【Spark重点难点03】你的数据存在哪了?》**
- **《【Spark重点难点04】你的代码跑起来谁说了算？(内存管理)》**
- **《我们在学习Spark的时候，到底在学习什么？》**
- **《我在B站读大学，大数据专业》**
- **Spark源码阅读的正确打开方式**

前言

在之前的课中我们讲了Spark的RDD以及整个Spark系统中的一些关键角色：

- **《【Spark重点难点】你从未深入理解的RDD和关键角色》**。

以及Spark中非常重要的一个概念Shuffle:

- **《【Spark重点难点】你以为的Shuffle和真正的Shuffle》**

以及关于Spark中的数据存储：

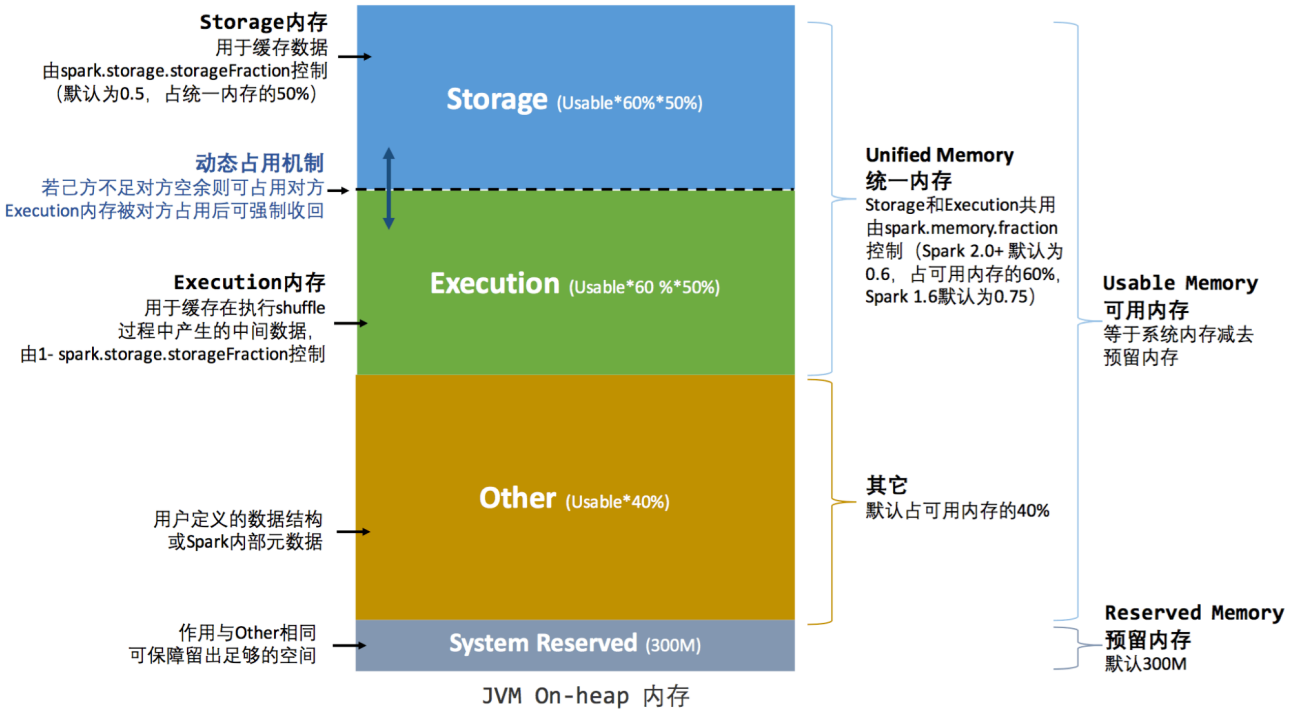
- **《【Spark重点难点】你的数据存在哪了?》**

这节课我们要讲的是Spark中的 **【内存模型】**，也就是决定我们Spark代码运行所需要的资源信息。

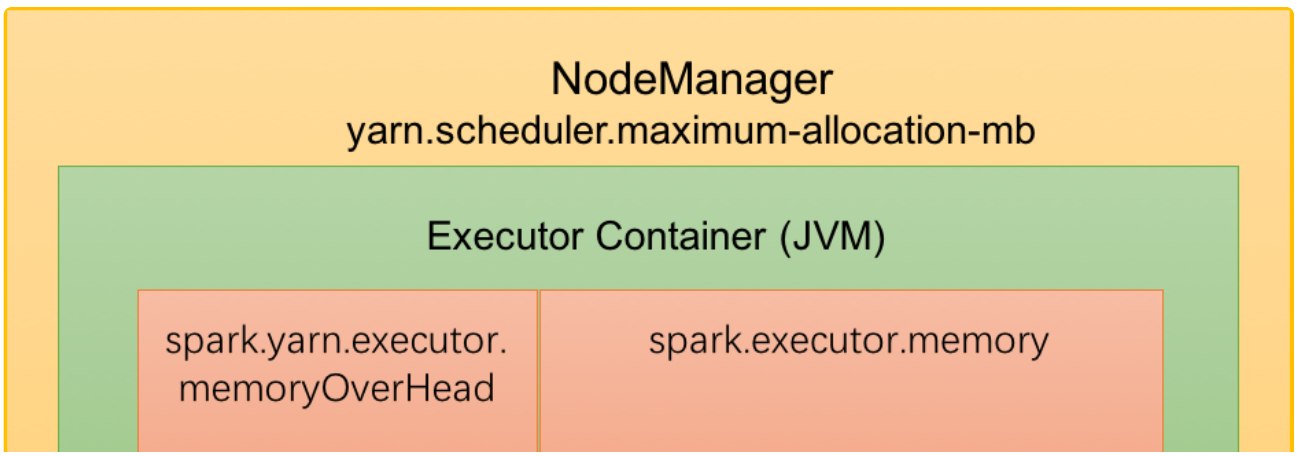
Spark这种基于内存计算的大数据引擎相比Hadoop的特性与优势就是【内存计算】。如何充分合理的利用内存资源，是Spark的核心竞争力之一。

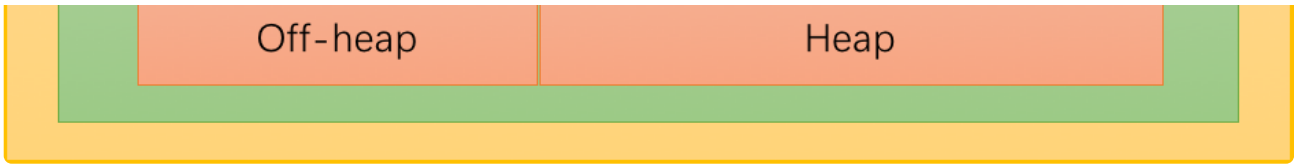
内存划分

我们先来一张经典的Spark内存分布图:



Spark2.0采用了 `统一内存管理模式` ，统一内存管理模块包括了堆内内存(`On-heap Memory`)和堆外内存(`Off-heap Memory`)两大区域。



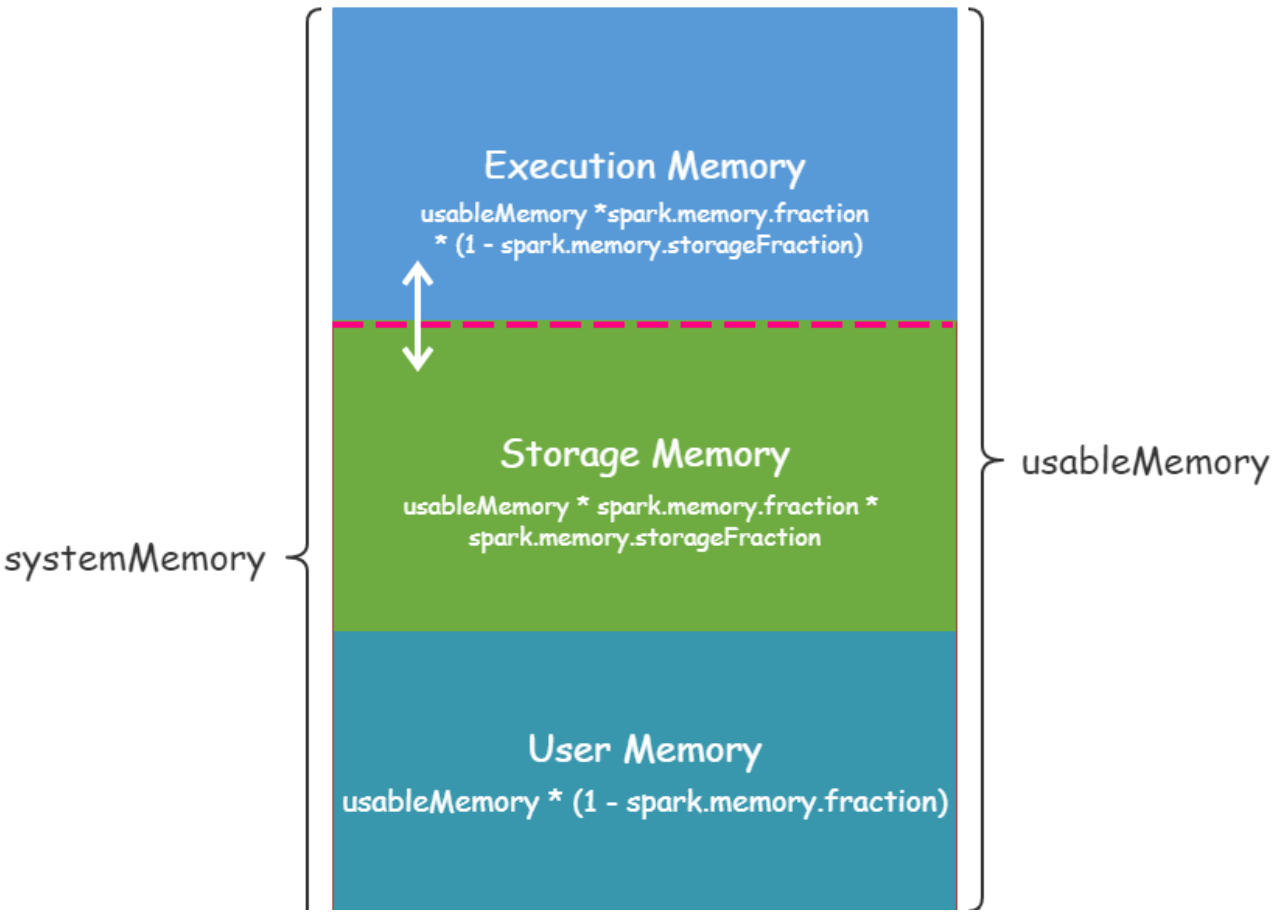


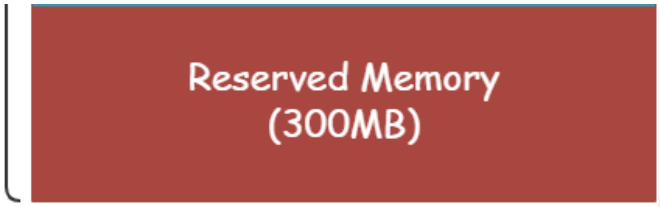
堆外内存 由Spark控制，直接在工作节点的系统内存中开辟空间，即对于大内存，Spark自行和内存打交道。堆外内存只区分 Execution内存 和 Storage内存 ，这部分用户代码无法直接操作。堆外内存部分主要用于JVM自身，如字符串、NIO Buffer等开销。另外还有部分堆外内存由 spark.memory.offHeap.enabled 及 spark.memory.offHeap.size 控制的堆外内存，这部分也归offheap,但主要是供统一内存管理使用的。

堆内内存 依赖JVM:

- Execution 内存: 主要用于存放Shuffle、Join、Sort、Aggregation 等计算过程中的临时数据;
- Storage 内存:主要用于存储Spark的cache数据，例如RDD的缓存、unroll数据;
- 用户内存（User Memory）：主要用于存储 RDD 转换操作所需要的数据，例如 RDD 依赖等信息;
- 预留内存（Reserved Memory）：系统预留内存，会用来存储Spark内部对象。

如下图所示:





有哪些配置?

堆内内存配置

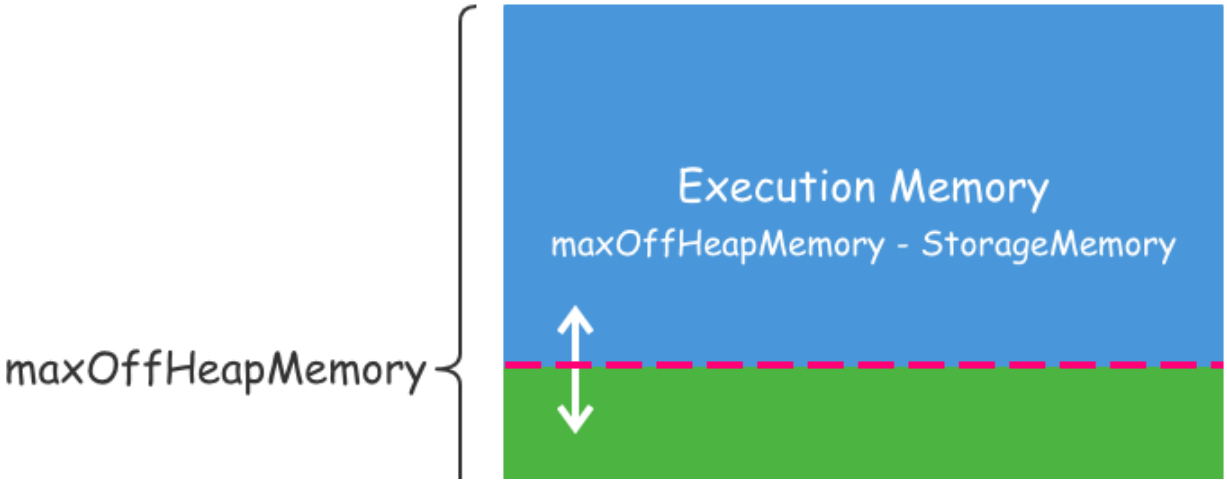
- Reserved Memory 保留内存，系统默认值为300，一般无需改动，不用关心此部分内存。 但如果Executor分配的内存小于 1.5 * 300 = 450M时，Executor将无法执行。
- Storage Memory 存储内存，用于存放广播数据及RDD缓存数据。

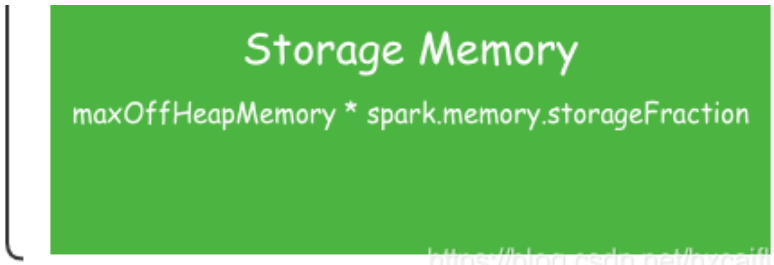
Spark 2.0以及更新的版本中，初始状态下， `Storage Memory` 及 `Execution Memory` 均约占系统总内存的30%，也就是（1 * 0.6 * 0.5 = 0.3）。

> 在 统一内存管理模式 下，这两部分内存可以相互借用。

也就是说,如果 `Execution` 内存不足,而 `Storage` 内存有空闲,那么 `Execution` 可以从 `Storage` 中申请空间;反之亦然。

堆外内存配置





Spark1.6开始引入了 `Off-heap memory` 。这种模式不在JVM内申请内存,而是调用Java的unsafe相关API进行诸如C语言里面的 `malloc()` 直接向操作系统申请内存,由于这种方式不进过JVM内存管理,所以可以避免频繁的GC,这种内存申请的缺点是必须自己编写内存申请和释放的逻辑。

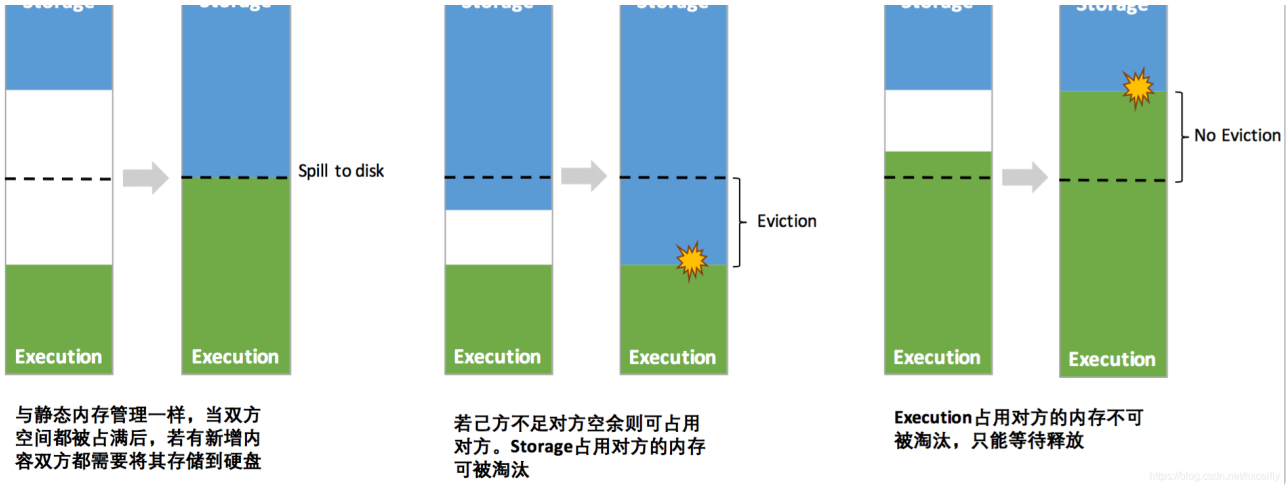
堆外内存在Spark中可以从逻辑上分成两种: 一种是 `DirectMemory` , 一种是 `JVM Overhead(下面统称为off heap)` 。 默认情况下，堆外内存是关闭的，我们可以通过 `spark.memory.offHeap.enabled` 参数启用，并且通过 `spark.memory.offHeap.size` 设置堆外内存大小，单位为字节。如果堆外内存被启用，那么 `Executor` 内将同时存在堆内和堆外内存，两者的使用互补影响，这个时候 `Executor` 中的 `Execution内存` 是**堆内的Execution内存和堆外的Execution内存之和**，同理，Storage内存也一样。相比堆内内存，堆外内存只区分 `Execution内存` 和 `Storage 内存` 。

动态调整是个啥?

Spark 1.6 之后引入的统一内存管理机制，与静态内存管理的区别在于存储内存和执行内存共享同一块空间，可以动态占用对方的空闲区域。

其中最重要的优化在于动态占用机制，其规则如下:

- 设定基本的 `Storage内存` 和 `Execution内区` 域(spark.storage.storageFraction 参数),该设定确定了双方各自拥有的空间的范围
- 双方的空间都不足时,则存储到硬盘;若己方空间不足而对方空余时,可借用对方的空间;(存储空间不足是指不足以放下一个完整的内存数据块Block)
- `Execution内存` 的空间被对方占用后，可让对方将占用的部分转存到硬盘,然后"归还"借用的空间
- `Storage内存` 的空间被对方占用后，无法让对方"归还"，多余的Storage内存被转存到硬盘，因为需要考虑Shuffle过程中的很多因素，实现起来较为复杂。



内存管理

这部分我们引用之前发过的文章:

- Spark性能优化总结
- 三万字长文 | Spark性能优化实战手册

Storage存储内存管理

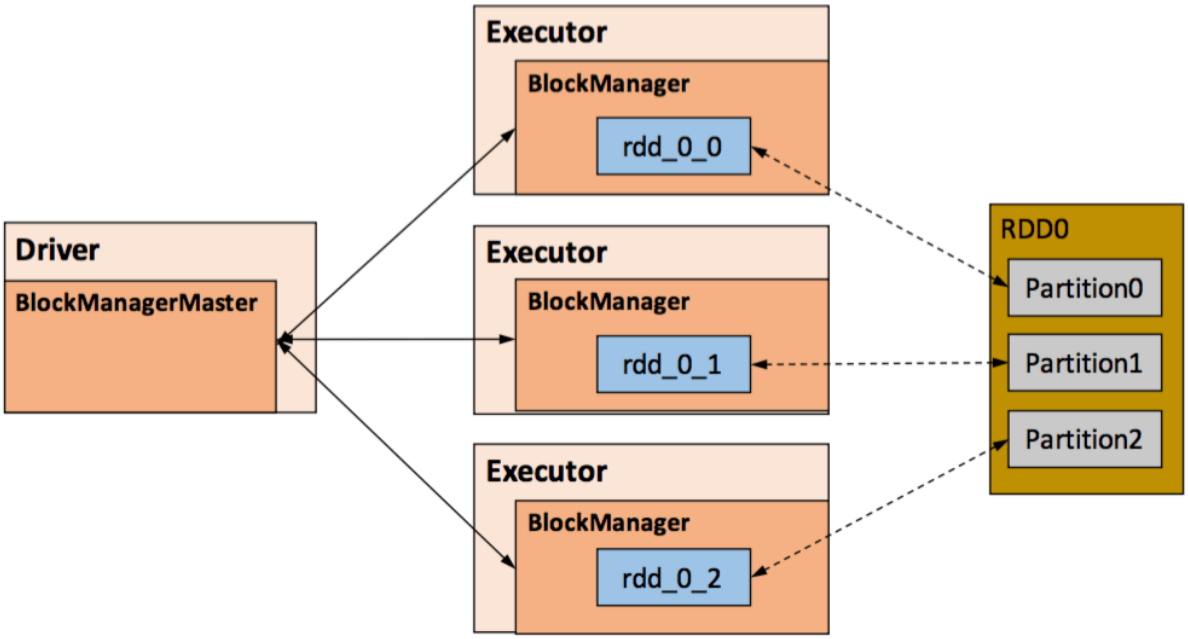
RDD持久化机制

弹性分布式数据集（RDD）作为 Spark 最根本的数据抽象，是只读的分区记录（Partition）的集合，只能基于在稳定物理存储中的数据集上创建，或者在其他已有的 RDD 上执行转换（Transformation）操作产生一个新的 RDD。转换后的 RDD 与原始的 RDD 之间产生的依赖关系，构成了血统（Lineage）。凭借血统，Spark 保证了每一个 RDD 都可以被重新恢复。但 RDD 的所有转换都是惰性的，即只有当一个返回结果给 Driver 的行动（Action）发生时，Spark 才会创建任务读取 RDD，然后真正触发转换的执行。

Task 在启动之初读取一个分区时，会先判断这个分区是否已经被持久化，如果没有则需要检查 Checkpoint 或按照血统重新计算。所以如果一个 RDD 上要执行多次行动，可以在第一次行动中使用 persist 或 cache 方法，在内存或磁盘中持久化或缓存这个 RDD，从而在后面的行动时提升计算速度。事实上，cache 方法是使用默认的 MEMORY_ONLY 的存储级别将 RDD 持久化到内存，故缓存是一种特殊的持久化。 堆内和堆外存储内存

的设计，便可以对缓存 RDD 时使用的内存做统一的规划和管 理 （存储内存的其他应用场景，如缓存 broadcast 数据，暂时不在本文的讨论范围之内）。

RDD 的持久化由Spark的Storage模块负责，实现了RDD与物理存储的解耦合。Storage模块负责管理 Spark 在计算过程中产生的数据，将那些在内存或磁盘、在本地或远程存取数据的功能封装了起来。在具体实现时 Driver 端和 Executor 端的 Storage 模块构成了主从式的架构，即 Driver 端的 BlockManager 为 Master，Executor 端的 BlockManager 为 Slave。Storage 模块在逻辑上以 Block 为基本存储单位，RDD 的每个 Partition 经过处理后唯一对应一个 Block（BlockId 的格式为 rdd_RDD-ID_PARTITION-ID ）。Master 负责整个 Spark 应用程序的 Block 的元数据信息的管理和维护，而 Slave 需要将 Block 的更新等状态上报到 Master，同时接收 Master 的命令，例如新增或删除一个 RDD。



在对RDD持久化时，Spark规定了MEMORY_ONLY、MEMORY_AND_DISK 等 7种不同的存储级别:

存储级别	存储介质		存储形式		副本数量	备注
	内存	磁盘	对象值	序列化		
MEMORY_ONLY	✅		✅		1	
MEMORY_ONLY_2	✅		✅		2	
MEMORY_ONLY_SER	✅			✅	1	
MEMORY_ONLY_SER_2	✅			✅	2	
DISK_ONLY		✅		✅	1	
DISK_ONLY_2		✅		✅	2	
DISK_ONLY_3		✅		✅	3	

MEMORY_AND_DISK	✅	✅	✅	✅	1	内存的部分以对象值存储，磁盘部分序列化
MEMORY_AND_DISK_2	✅	✅	✅	✅	2	
MEMORY_AND_DISK_SER	✅	✅		✅	1	内存和磁盘都以序列化的字节数组形式进行存储
MEMORY_AND_DISK_SER2	✅	✅		✅	2	

而存储级别是以下5个变量的组合：

```
class StorageLevel private(
private var _useDisk: Boolean, //磁盘
private var _useMemory: Boolean, //这里其实是指堆内内存
private var _useOffHeap: Boolean, //堆外内存
private var _deserialized: Boolean, //是否为非序列化
private var _replication: Int = 1 //副本个数
)
```

通过对数据结构的分析，可以看出存储级别从三个维度定义了 RDD 的 Partition（同时也就是 Block）的存储方式：

- 存储位置：磁盘 / 堆内内存 / 堆外内存。如 MEMORY_AND_DISK 是同时在磁盘和堆内内存上存储，实现了冗余备份。OFF_HEAP 则是只在堆外内存存储，目前选择堆外内存时不能同时存储到其他位置。
- 存储形式：Block 缓存到存储内存后，是否为非序列化的形式。如 MEMORY_ONLY 是非序列化方式存储，OFF_HEAP 是序列化方式存储。
- 副本数量：大于 1 时需要远程冗余备份到其他节点。如 DISK_ONLY_2 需要远程备份 1 个副本。

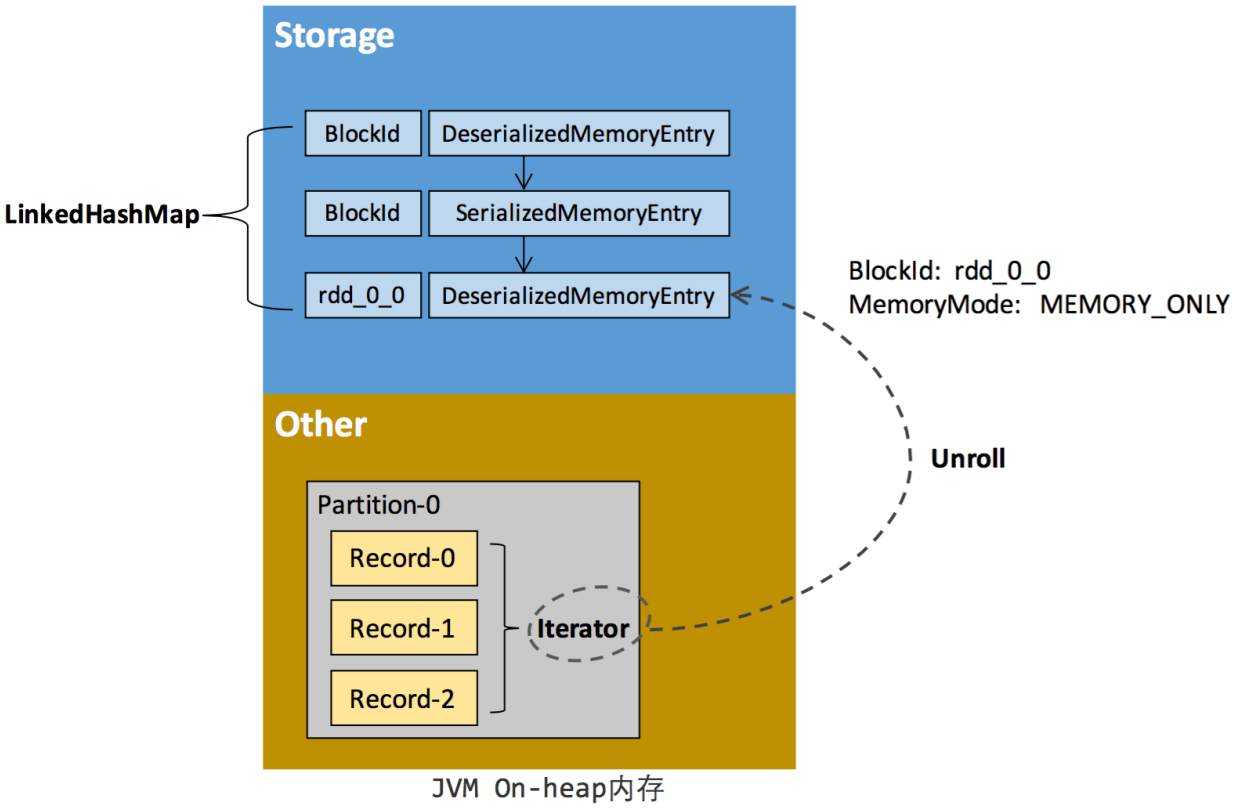
RDD缓存的过程

RDD 在缓存到存储内存之前，Partition 中的数据一般以迭代器（Iterator）的数据结构来访问，这是 Scala 语言中一种遍历数据集合的方法。通过 Iterator可以获取分区中每一条序列化或者非序列化的数据项(Record)，这些 Record 的对象实例在逻辑上占用了 JVM 堆内内存的 other 部分的空间，同一 Partition 的不同 Record 的空间并不连续。

RDD 在缓存到存储内存之后，Partition 被转换成 Block，Record 在堆内或堆外存储内存中占用一块连续的空间。将Partition由不连续的存储空间转换为连续存储空间的过程，Spark称之为"展开"（Unroll）。Block 有序列化和非序列化两种存储格式，具体以哪种方式取决于该 RDD 的存储级别。非序列化的 Block 以一种 DeserializedMemoryEntry 的

数据结构定义，用一个数组存储所有的对象实例，序列化的 Block 则以SerializedMemoryEntry的数据结构定义，用字节缓冲区（ByteBuffer）来存储二进制数据。每个 Executor 的 Storage 模块用一个链式 Map 结构（LinkedHashMap）来管理堆内和堆外存储内存中所有的 Block 对象的实例，对这个 LinkedHashMap 新增和删除间接记录了内存的申请和释放。

因为不能保证存储空间可以一次容纳 Iterator 中的所有数据，当前的计算任务在 Unroll 时要向 MemoryManager 申请足够的 Unroll 空间来临时占位，空间不足则 Unroll 失败，空间足够时可以继续进行。对于序列化的 Partition，其所需的 Unroll 空间可以直接累加计算，一次申请。而非序列化的 Partition 则要在遍历 Record 的过程中依次申请，即每读取一条 Record，采样估算其所需的 Unroll 空间并进行申请，空间不足时可以中断，释放已占用的 Unroll 空间。如果最终 Unroll 成功，当前 Partition 所占用的 Unroll 空间被转换为正常的缓存 RDD 的存储空间，如下图所示。



在静态内存管理时，Spark 在存储内存中专门划分了一块 Unroll 空间，其大小是固定的，统一内存管理时则没有对 Unroll 空间进行特别区分，当存储空间不足时会根据动态占用机制进行处理。

淘汰和落盘机制

由于同一个 Executor 的所有的计算任务共享有限的存储内存空间，当有新的 Block 需要缓存但是剩余空间不足且无法动态占用时，就要对 LinkedHashMap 中的旧 Block 进行淘汰（Eviction），而被淘汰的 Block 如果其存储级别中同时包含存储到磁盘的要求，则要对其进行落盘（Drop），否则直接删除该 Block。

存储内存的淘汰规则为：

- 被淘汰的旧 Block 要与新 Block 的 MemoryMode 相同，即同属于堆外或堆内内存
- 新旧 Block 不能属于同一个 RDD，避免循环淘汰
- 旧 Block 所属 RDD 不能处于被读状态，避免引发一致性问题
- 遍历 LinkedHashMap 中 Block，按照最近最少使用（LRU）的顺序淘汰，直到满足新 Block 所需的空间。其中 LRU 是 LinkedHashMap 的特性。

落盘的流程则比较简单，如果其存储级别符合_useDisk 为 true 的条件，再根据其 `_deserialized` 判断是否是非序列化的形式，若是则对其进行序列化，最后将数据存储到磁盘，在 Storage模块中更新其信息。

Execution执行内存管理

多任务间内存分配

Executor内运行的任务同样共享执行内存，Spark 用一个 HashMap 结构保存了任务到内存耗费的映射。每个任务可占用的执行内存大小的范围为 1/2N ~ 1/N，其中 N 为当前Executor内正在运行的任务的个数。每个任务在启动之时，要向 `MemoryManager` 请求申请最少为 1/2N 的执行内存，如果不能被满足要求则该任务被阻塞，直到有其他任务释放了足够的执行内存，该任务才可以被唤醒。

Shuffle 的内存占用

执行内存主要用来存储任务在执行 Shuffle 时占用的内存，Shuffle 是按照一定规则对 RDD 数据重新分区的过程，我们来看 Shuffle 的 Write 和 Read 两阶段对执行内存的使用：

- Shuffle Write

1. 若在 map 端选择普通的排序方式，会采用 ExternalSorter 进行外排，在内存中存储数据时主要占用堆内执行空间。

2. 若在 map 端选择 Tungsten 的排序方式，则采用 ShuffleExternalSorter 直接对以序列化形式存储的数据排序，在内存中存储数据时可以占用堆外或堆内执行空间，取决于用户是否开启了堆外内存以及堆外执行内存是否足够。

- Shuffle Read

1. 在对 reduce 端的数据进行聚合时，要将数据交给 Aggregator 处理，在内存中存储数据时占用堆内执行空间。
2. 如果需要进行最终结果排序，则要将再次将数据交给 ExternalSorter 处理，占用堆内执行空间。

在 `ExternalSorter` 和 `Aggregator` 中，Spark 会使用一种叫 `AppendOnlyMap` 的哈希表在堆内执行内存中存储数据，但在 Shuffle 过程中所有数据并不能都保存到该哈希表中，当这个哈希表占用的内存会进行周期性地采样估算，当其大到一定程度，无法再从 `MemoryManager` 申请到新的执行内存时，Spark 就会将其全部内容存储到磁盘文件中，这个过程被称为溢存(Spill)，溢存到磁盘的文件最后会被归并(Merge)。

`Shuffle Write` 阶段中用到的 `Tungsten` 是Databricks公司提出的对 Spark 优化内存和 CPU 使用的计划，解决了一些 JVM 在性能上的限制和弊端。

Spark会根据Shuffle的情况来自动选择是否采用Tungsten排序。Tungsten采用的页式内存管理机制建立在 `MemoryManager` 之上，即 `Tungsten` 对执行内存的使用进行了一步的抽象，这样在 Shuffle 过程中无需关心数据具体存储在堆内还是堆外。每个内存页用一个 `MemoryBlock` 来定义，并用 Object obj 和 long offset 这两个变量统一标识一个内存页在系统内存中的地址。堆内的 `MemoryBlock` 是以 long 型数组的形式分配的内存，其 obj 的值为是这个数组的对象引用，offset 是 long 型数组的在 JVM 中的初始偏移地址，两者配合使用可以定位这个数组在堆内的绝对地址；堆外的 `MemoryBlock` 是直接申请到的内存块，其 obj 为 null，offset是这个内存块在系统内存中的 64 位绝对地址。Spark用MemoryBlock巧妙地将堆内和堆外内存页统一抽象封装，并用页表(pageTable)管理每个 Task 申请到的内存页。

Tungsten 页式管理下的所有内存用 64 位的逻辑地址表示，由页号和页内偏移量组成：

- 页号：占 13 位，唯一标识一个内存页，Spark 在申请内存页之前要先申请空闲页号。
- 页内偏移量：占 51 位，是在使用内存页存储数据时，数据在页内的偏移地址。

有了统一的寻址方式，Spark 可以用 64 位逻辑地址的指针定位到堆内或堆外的内存，整个 `Shuffle Write` 排序的过程只需要对指针进行排序，并且无需反序列化，整个过程非常高效，对于内存访问效率和 CPU 使用效率带来了明显的提升。

Spark 的 `Storage存储内存` 和 `Execution执行内存` 有着截然不同的管理方式:对于存储内存来说，Spark用一个LinkedHashMap来集中管理所有的 Block，Block由需要缓存的RDD的Partition转化而成;而对于执行内存，Spark用 `AppendOnlyMap` 来存储 `Shuffle` 过程中的数据，在Tungsten排序中甚至抽象成为页式内存管理，开辟了全新的JVM内存管理机制。