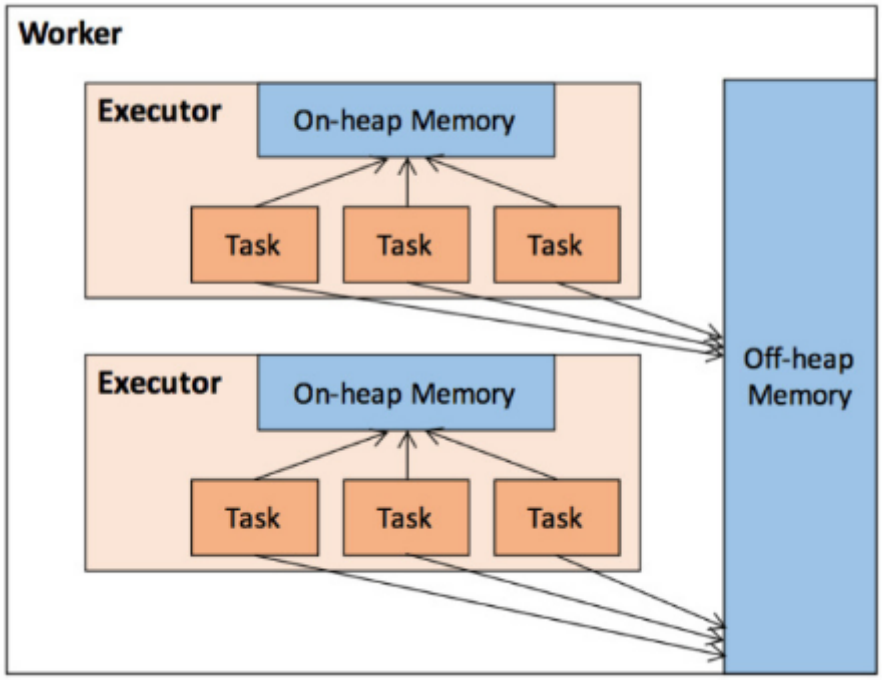
> 专注-反馈-修正

Spark内存管理

堆内和堆外内存规划

> 作为一个 JVM 进程，Executor 的内存管理建立在**JVM 的内存管理之上**，Spark 对 JVM 的堆内（On-heap）空间进行了更为详细的分配，以充分利用内存。同时，Spark 引入了堆外（Off-heap）内存，使之可以直接在工作节点的系统内存中开辟空间，进一步优化了内存的使用。堆内内存受到JVM统一管理，堆外内存是直接向操作系统进行内存的申请和释放。



堆内内存

> 堆内内存的大小，由 `Spark` 应用程序启动时的 `–executor-memory` 或 `spark.executor.memory` 参数配置。`Executor` 内运行的并发任务共享 `JVM` 堆内内存，这些任务在缓存 `RDD` 数据和广播（`Broadcast`）数据时占用的内存被规划为存储（`Storage`）内存，而这些任务在执行 `Shuffle` 时占用的内存被规划为执行（`Execution`）内存，剩余的部分不做特殊规划，那些 `Spark` 内部的对象实例，或者用户定义的 `Spark` 应用程序中的对象实例，均占用剩余的空间。不同的管理模式下，这三部分占用的空间大小各不相同。

> Spark 对堆内内存的管理是一种逻辑上的"规划式"的管理，因为对象实例占用内存的申请和释放都由 JVM 完成，**Spark 只能在申请后和释放前记录这些内存**，我们来看其具体流程：

申请内存流程如下：

1. Spark 在代码中 new 一个对象实例；
2. JVM 从堆内内存分配空间，创建对象并返回对象引用；
3. Spark 保存该对象的引用，记录该对象占用的内存。

释放内存流程如下：

1. Spark记录该对象释放的内存，删除该对象的引用；
2. 等待JVM的垃圾回收机制释放该对象占用的堆内内存。

我们知道，JVM 的对象可以以序列化的方式存储，序列化的过程是将对象转换为二进制字节流，本质上可以理解为将非连续空间的链式存储转化为连续空间或块存储，在访问时则需要进行序列化的逆过程——反序列化，将字节流转化为对象，序列化的方式可以节省存储空间，但增加了存储和读取时候的计算开销。

对于 Spark 中序列化的对象，由于是字节流的形式，其占用的内存大小可直接计算，而对于非序列化的对象，其占用的内存是通过周期性地采样近似估算而得，即并不是每次新增的数据项都会计算一次占用的内存大小，这种方法降低了时间开销但是有可能误差较大，导致某一时刻的实际内存有可能远远超出预期。此外，在被 Spark 标记为释放的对象实例，很有可能在实际上并没有被 JVM 回收，导致实际可用的内存小于 Spark 记录的可用内存。所以 Spark 并不能准确记录实际可用的堆内内存，从而也就无法完全避免内存溢出（OOM, Out of Memory）的异常。

虽然不能精准控制堆内内存的申请和释放，但 Spark 通过对存储内存和执行内存各自独立的规划管理，可以决定是否要在存储内存里缓存新的 RDD，以及是否为新的任务分配执行内存，在一定程度上可以提升内存的利用率，减少异常的出现。

堆外内存

为了进一步优化内存的使用以及提高 Shuffle 时排序的效率，Spark 引入了堆外（Off-heap）内存，使之可以直接在工作节点的系统内存中开辟空间，存储经过序列化的二进制数据。

堆外内存意味着把内存对象分配在Java虚拟机的堆以外的内存，这些内存直接受操作系统管理（而不是虚拟机）。这样做的结果就是能保持一个较小的堆，以减少垃圾收集对应用的影响。

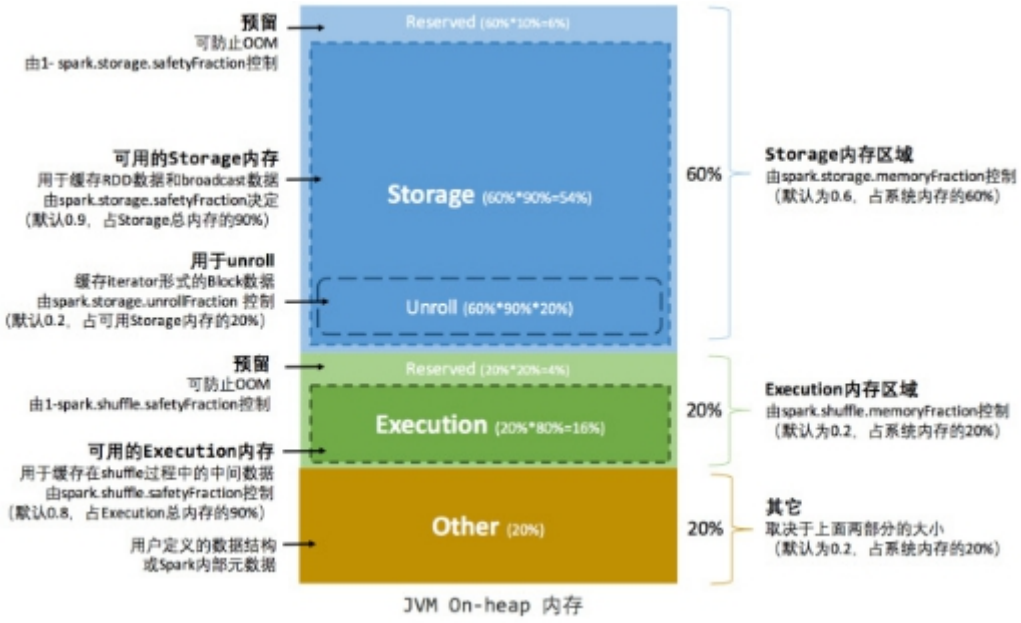
利用 JDK Unsafe API（从 Spark 2.0 开始，在管理堆外的存储内存时不再基于 Tachyon，而是与堆外的执行内存一样，基于 JDK Unsafe API 实现），Spark 可以直接操作系统堆外内存，减少了不必要的内存开销，以及频繁的 GC 扫描和回收，提升了处理性能。堆外内存可以被精确地申请和释放（堆外内存之所以能够被精确的申请和释放，是由于内存的申请和释放不再通过JVM机制，而是直接向操作系统申请，JVM对于内存的清理是无法准确指定时间点的，因此无法实现精确的释放），而且序列化的数据占用的空间可以被精确计算，所以相比堆内内存来说降低了管理的难度，也降低了误差。

在默认情况下堆外内存并不启用，可通过配置 spark.memory.offHeap.enabled 参数启用，并由 spark.memory.offHeap.size 参数设定堆外空间的大小。除了没有 other 空间，堆外内存与堆内内存的划分方式相同，所有运行中的并发任务共享存储内存和执行内存。

内存空间分配

静态内存管理

在 Spark 最初采用的静态内存管理机制下，存储内存、执行内存和其他内存的大小在 Spark 应用程序运行期间均为固定的，但用户可以应用程序启动前进行配置，堆内内存的分配如图所示：



可以看到，可用的堆内内存的大小需要按照代码清单1-1的方式计算：

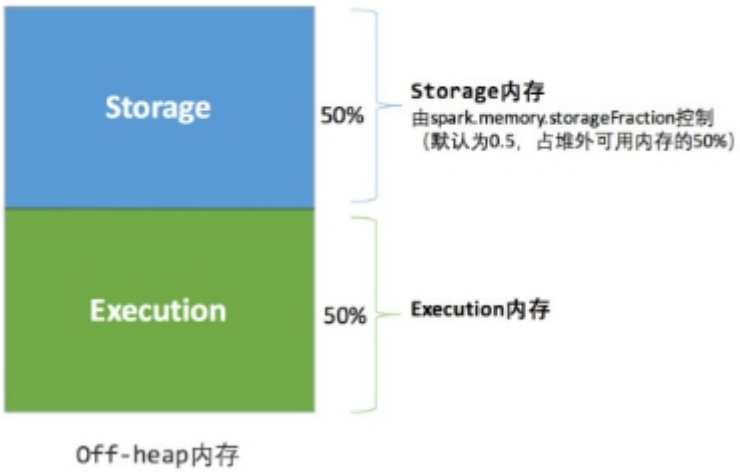
代码清单1-1 堆内内存计算公式

```
可用的存储内存 = systemMaxMemory * spark.storage.memoryFraction *
spark.storage.safety Fraction
可用的执行内存 = systemMaxMemory * spark.shuffle.memoryFraction *
spark.shuffle.safety Fraction
```

其中 systemMaxMemory 取决于当前** JVM 堆内内存的大小**，最后可用的执行内存或者存储内存要在此基础上与各自的 memoryFraction 参数和 safetyFraction 参数相乘得出。上述计算公式中的两个 safetyFraction 参数，其意义在于在逻辑上预留出 1-safetyFraction 这么一块保险区域，降低因实际内存超出当前预设范围而导致 OOM 的风险（上文提到，对于非序列化对象的内存采样估算会产生误差）。值得注意的是，这个预留的保险区域仅仅是一种逻辑上的规划，在具体使用时 Spark 并没有区别对待，和"其它内存"一样交给了 JVM 去管理。

Storage内存和Execution内存都有预留空间，目的是防止OOM，因为Spark堆内内存大小的记录是不准确的，需要留出保险区域。

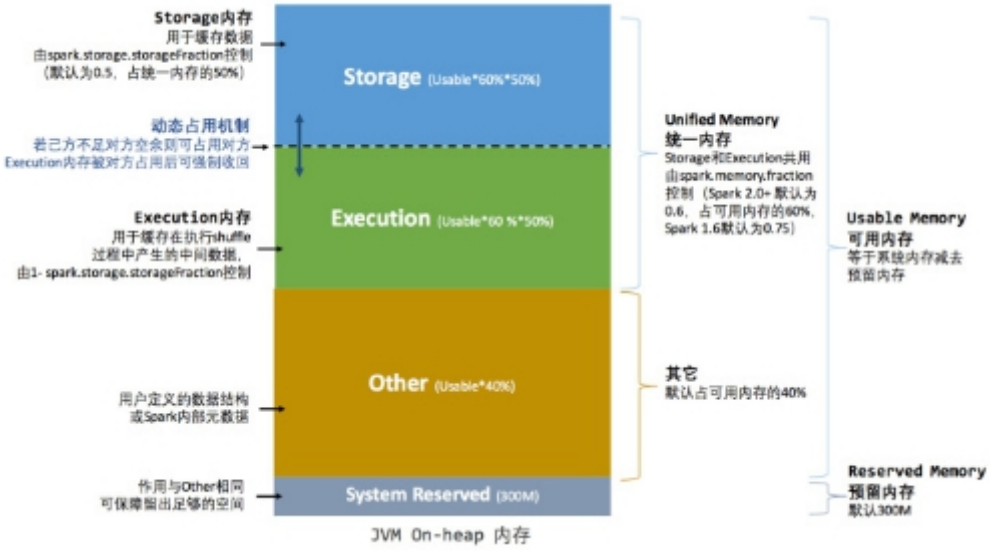
堆外的空间分配较为简单，**只有存储内存和执行内存**，如图所示。可用的执行内存和存储内存占用的空间大小直接由参数spark.memory.storageFraction 决定，由于堆外内存占用的空间可以被精确计算，所以无需再设定保险区域。



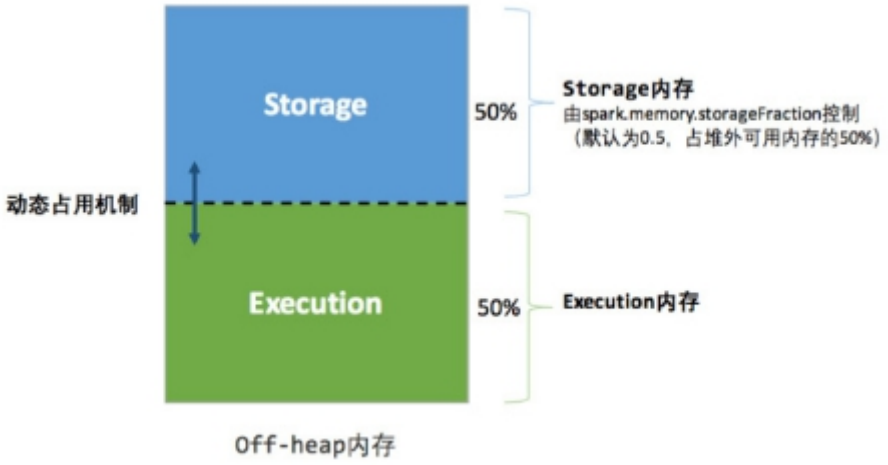
静态内存管理机制实现起来较为简单，但如果用户不熟悉 Spark 的存储机制，或没有根据具体的数据规模和计算任务或做相应的配置，很容易造成**"一半海水，一半火焰"**的局面，即存储内存和执行内存中的一方剩余大量的空间，而另一方却早早被占满，不得不淘汰或移出旧的内容以存储新的内容。由于新的内存管理机制的出现，这种方式目前已经很少有开发者使用，出于兼容旧版本的应用程序的目的，Spark 仍然保留了它的实现。

统一内存管理

> **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，统一内存管理的堆内内存结构如图所示：



统一内存管理的堆外内存结构如图所示：



其中最重要的优化在于动态占用机制，其规则如下：

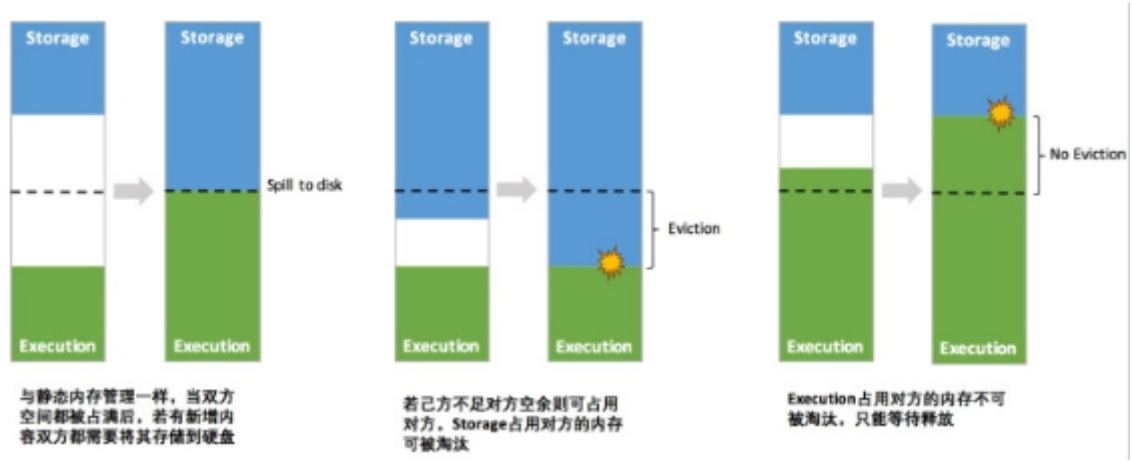
1.设定基本的存储内存和执行内存区域（`spark.storage.storageFraction` 参数），该设定确定了双方各自拥有的空间的范围；

2.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 `Block`）

3.执行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；

4.存储内存的空间被对方占用后，无法让对方"归还"，因为需要考虑 `Shuffle` 过程中的很多因素，实现起来较为复杂。

统一内存管理的动态占用机制如图所示：



> 凭借统一内存管理机制，Spark 在一定程度上**提高了堆内和堆外内存资源的利用率**，降低了开发者维护 Spark 内存的难度，但并不意味着开发者可以高枕无忧。如果存储内存的空间太大或者说缓存的数据过多，反而会导致**频繁的全量垃圾回收**，降低任务执行时的性能，因为**缓存的 RDD 数据通常都是长期驻留内存的**。所以要想充分发挥 Spark 的性能，需要开发者进一步了解存储内存和执行内存各自的管理方式和实现原理。

存储内存管理

RDD的持久化机制

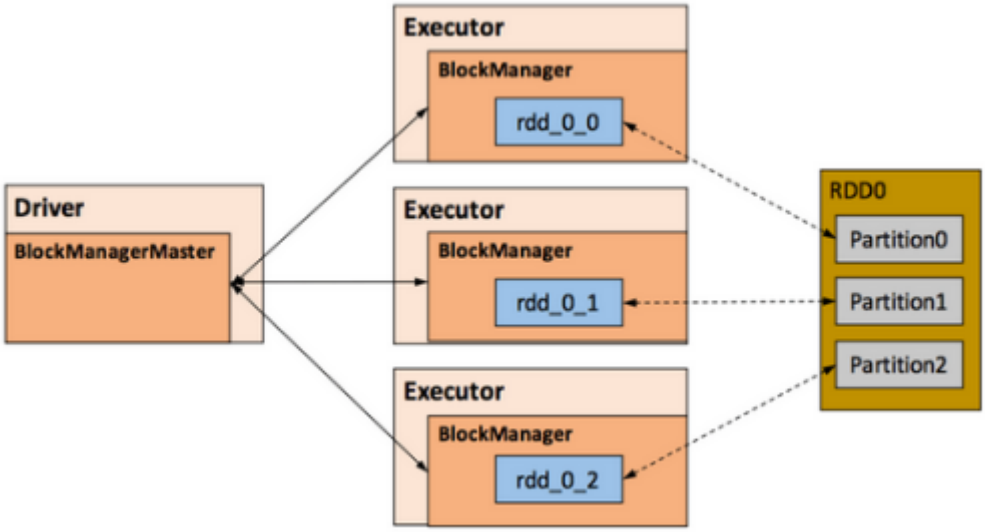
> 弹性分布式数据集（RDD）作为 Spark 最根本的数据抽象，是只读的分区记录（Partition）的集合，只能基于在稳定物理存储中的数据集上创建，或者在其他已有的 RDD 上执行转换（Transformation）操作产生一个新的 RDD。转换后的 RDD 与原始的 RDD 之间产生的依赖关系，构成了血统（Lineage）。**凭借血统，Spark 保证了每一个 RDD 都可以被重新恢复**。但 RDD 的所有转换都是惰性的，即只有当一个返回结果给 Driver 的行动（Action）发生时，Spark 才会创建任务读取 RDD，然后真正触发转换的执行。

`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。

`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` ）。`Driver`端的`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`Executor`端的 `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，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
)
```

Spark中7种存储级别如下

持久化级别	***含义***
MEMORY_ONLY	以非序列化的Java对象的方式持久化在JVM内存中。如果内存无法完全存储RDD所有的partition，那么那些没有持久化的partition就会在下一次需要使用它们的时候，重新被计算
MEMORY_AND_DISK	同上，但是当某些partition无法存储在内存中时，会持久化到磁盘中。下次需要使用这些partition时，需要从磁盘上读取
MEMORY_ONLY_SER	同MEMORY_ONLY，但是会使用Java序列化方式，将Java对象序列化后进行持久化。可以减少内存开销，但是需要进行反序列化，因此会加大CPU开销
MEMORY_AND_DISK_SER	同MEMORY_AND_DISK，但是使用序列化方式持久化Java对象
DISK_ONLY	使用非序列化Java对象的方式持久化，完全存储到磁盘上
MEMORY_ONLY_2**MEMORY_AND_DISK_2**等等	如果是尾部加了2的持久化级别，表示将持久化数据复用一份，保存到其他节点，从而在数据丢失时，不需要再次计算，只需要使用备份数据即可

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：

1) 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。

2) 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。

3) 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的缓存过程

> 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 的存储空间并不连续。**

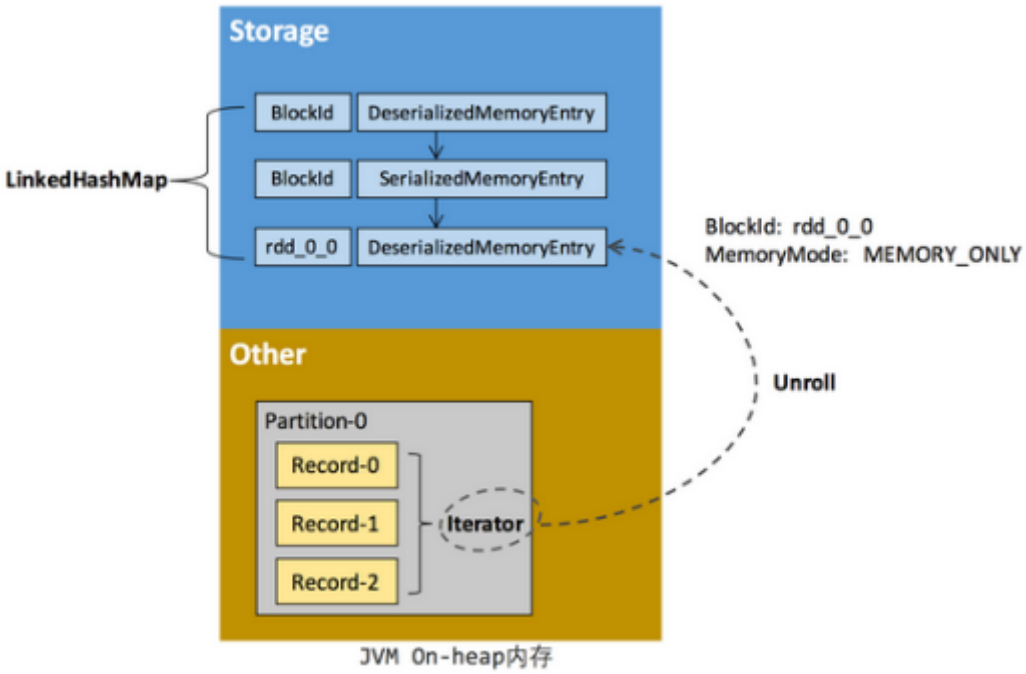
RDD 在缓存到存储内存之后，Partition 被转换成 Block，Record 在堆内或堆外存储内存中占用一块连续的空间。**将Partition由不连续的存储空间转换为连续存储空间的过程，Spark称之为"展开"（Unroll）。**

> `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。
- 对于非序列化的 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

> **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。

执行内存管理

> 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（钨丝计划），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 的存储内存和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。