06 | 存储系统：空间换时间，还是时间换空间？

2021-03-26 吴磊

Spark性能调优实战 进入课程 >

讲述：吴磊
时长 19:15 大小 17.63M

你好，我是吴磊。

今天，我们来学习 Spark 的存储系统，它和我们上一讲学过的调度系统一样，都是 Spark 分布式计算引擎的基础设施之一。

你可能会问：“在日常的开发工作中，除了业务逻辑实现，我真的需要去关心这么底层的东西吗？”确实，存储系统离开发者比较远。不过，如果把目光落在存储系统所服务的对象上，你很可能会改变这种看法。

接下来，咱们就先来看看 Spark 存储系统都为谁服务，再去探讨它有哪些重要组件，以及它是如何工作的，带你一次性摸透存储系统。

Spark 存储系统是为谁服务的?

Spark 存储系统用于存储 3 个方面的数据，**分别是 RDD 缓存、Shuffle 中间文件、广播变量。我们一个一个来说。**

RDD 缓存指的**是将 RDD 以缓存的形式物化到内存或磁盘的过程**。对于一些计算成本和访问频率都比较高的 RDD 来说，缓存有两个好处：一是通过截断 DAG，可以降低失败重试的计算开销；二是通过对缓存内容的访问，可以有效减少从头计算的次数，从整体上提升作业端到端的执行性能。

而要说起 Shuffle 中间文件，我们就不得不提 Shuffle 这个话题。在很多场景中，Shuffle 都扮演着性能瓶颈的角色，解决掉 Shuffle 引入的问题之后，执行性能往往能有立竿见影的提升。因此，凡是与 Shuffle 有关的环节，你都需要格外地重视。

关于 Shuffle 的工作原理，我们后面会详细来讲。这里，咱们先简单理解一下 Shuffle 的计算过程就可以了。它的计算过程可以分为 2 个阶段：

Map 阶段：Shuffle writer 按照 Reducer 的分区规则将中间数据写入本地磁盘；

Reduce 阶段：Shuffle reader 从各个节点下载数据分片，并根据需要进行聚合计算。

Shuffle 中间文件实际上就是 Shuffle Map 阶段的输出结果，这些结果会以文件的形式暂存于本地磁盘。在 Shuffle Reduce 阶段，Reducer 通过网络拉取这些中间文件用于聚合计算，如求和、计数等。在集群范围内，Reducer 想要拉取属于自己的那部分中间数据，就必须要知道这些数据都存储在哪些节点，以及什么位置。而这些关键的元信息，正是由 Spark 存储系统保存并维护的。因此你看，**没有存储系统，Shuffle 是玩不转的。**

最后，我们再来说说广播变量。在日常开发中，广播变量往往用于在集群范围内分发访问频率较高的小数据。**利用存储系统，广播变量可以在 Executors 进程范畴内保存全量数据。**这样一来，对于同一 Executors 内的所有计算任务，应用就能够以 Process local 的本地性级别，来共享广播变量中携带的全量数据了。

总的来说，**这 3 个服务对象是 Spark 应用性能调优的有力“抓手”，而它们又和存储系统有着密切的联系，因此想要有效运用这 3 个方面的调优技巧，我们就必须要对存储系统有足够的理解。**

存储系统的基本组件有哪些?

与调度系统类似，Spark 存储系统是一个囊括了众多组件的复合系统，如 BlockManager、BlockManagerMaster、MemoryStore、DiskStore 和 DiskBlockManager 等等。

不过，家有千口、主事一人，**BlockManager 是其中最为重要的组件，它在 Executors 端负责统一管理和协调数据的本地存取与跨节点传输。**这怎么理解呢？我们可以从 2 方面来看。

> 对外，BlockManager 与 Driver 端的 BlockManagerMaster 通信，不仅定期向 BlockManagerMaster 汇报本地数据元信息，还会不定时按需拉取全局数据存储状态。另外，不同 Executors 的 BlockManager 之间也会以 Server/Client 模式跨节点推送和拉取数据块。
>
> 对内，BlockManager 通过组合存储系统内部组件的功能来实现数据的存与取、收与发。

那么，对于 RDD 缓存、Shuffle 中间文件和广播变量这 3 个服务对象来说，BlockManager 又是如何存储的呢？**Spark 存储系统提供了两种存储抽象：MemoryStore 和 DiskStore。BlockManager 正是利用它们来分别管理数据在内存和磁盘中的存取。**

其中，广播变量的全量数据存储在 Executors 进程中，因此它由 MemoryStore 管理。Shuffle 中间文件往往会落盘到本地节点，所以这些文件的落盘和访问就要经由 DiskStore。相比之下，RDD 缓存会稍微复杂一些，由于 RDD 缓存支持内存缓存和磁盘缓存两种模式，因此我们要视情况而定，缓存在内存中的数据会封装到 MemoryStore，缓存在磁盘上的数据则交由 DiskStore 管理。

有了 MemoryStore 和 DiskStore，我们暂时解决了数据“存在哪儿”的问题。但是，这些数据该以“什么形式”存储到 MemoryStore 和 DiskStore 呢？**对于数据的存储形式，Spark 存储系统支持两种类型：对象值（Object Values）和字节数组（Byte Array）。**它们之间可以相互转换，其中，对象值压缩为字节数组的过程叫做序列化，而字节数组还原成原始对象值的过程就叫做反序列化。

形象点来说，序列化的字节数组就像是从宜家家具超市购买的待组装板材，对象值则是将板材根据说明书组装而成的各种桌椅板凳。显而易见，对象值这种存储形式的优点是拿来即用、所见即所得，缺点是所需的存储空间较大、占地儿。相比之下，序列化字节数组的空间利用率要高得多。不过要是你着急访问里面的数据对象，还得进行反序列化，有点麻烦。

由此可见，对象值和字节数组二者之间存在着一种博弈关系，也就是所谓的“以空间换时间”和“以时间换空间”，两者之间该如何取舍，我们还是要看具体的应用场景。**核心原则就是：如果想省地儿，你可以优先考虑字节数组；如果想以最快的速度访问对象，还是对象值更直接一些。** 不过，这种选择的烦恼只存在于 MemoryStore 之中，而 DiskStore 只能存储序列化后的字节数组，毕竟，凡是落盘的东西，都需要先进行序列化。

透过 RDD 缓存看 MemoryStore

知道了存储系统有哪些核心的组件，下面，我们接着来说说 MemoryStore 和 DiskStore 这两个组件是怎么管理内存和磁盘数据的。

刚刚我们提到，**MemoryStore 同时支持存储对象值和字节数组这两种不同的数据形式，并且统一采用 MemoryEntry 数据抽象对它们进行封装。**

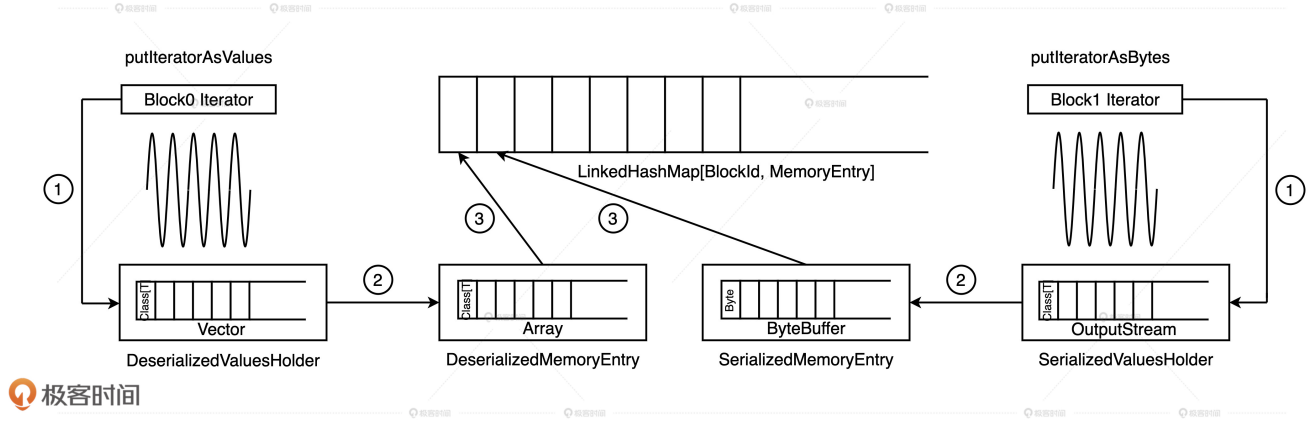
MemoryEntry 有两个实现类：DeserializedMemoryEntry 和 SerializedMemoryEntry，分别用于封装原始对象值和序列化之后的字节数组。DeserializedMemoryEntry 用 Array[T]来存储对象值序列，其中 T 是对象类型，而 SerializedMemoryEntry 使用 ByteBuffer 来存储序列化后的字节序列。

得益于 MemoryEntry 对于对象值和字节数组的统一封装，MemoryStore 能够借助一种高效的数据结构来统一存储与访问数据块：LinkedHashMap[BlockId, MemoryEntry]，即 Key 为 BlockId，Value 是 MemoryEntry 的链式哈希字典。在这个字典中，一个 Block 对应一个 MemoryEntry。显然，这里的 MemoryEntry 既可以是 DeserializedMemoryEntry，也可以是 SerializedMemoryEntry。有了这个字典，我们通过 BlockId 即可方便地查找和定位 MemoryEntry，实现数据块的快速存取。

概念这么多，命名也这么相似，是不是看起来就让人“头大”？别着急，接下来，咱们以 RDD 缓存为例，来看看存储系统是如何利用这些数据结构，把 RDD 封装的数据实体缓存到内存里去。

在 RDD 的语境下，我们往往用数据分片（Partitions/Splits）来表示一份分布式数据，但在存储系统的语境下，我们经常会用数据块（Blocks）来表示数据存储的基本单元。**在逻辑关系上，RDD 的数据分片与存储系统的 Block 一一对应，也就是说一个 RDD 数据分片会被物化成一个内存或磁盘上的 Block。**

因此，如果用一句话来概括缓存 RDD 的过程，就是将 RDD 计算数据的迭代器（Iterator）进行物化的过程，流程如下所示。具体来说，可以分成三步走。



利用MemoryStore在内存中缓存RDD数据内容

既然要把数据内容缓存下来，自然得先把 RDD 的迭代器展开成实实在在的数据值才行。因此，**第一步就是通过调用 putIteratorAsValues 或是 putIteratorAsBytes 方法，把 RDD 迭代器展开为数据值，然后把这些数据值暂存到一个叫做 ValuesHolder 的数据结构里。**这一步，我们通常把它叫做“Unroll”。

第二步，为了节省内存开销，我们可以在存储数据值的 ValuesHolder 上直接调用 toArray 或是 toByteBuffer 操作，把 ValuesHolder 转换为 MemoryEntry 数据结构。注意啦，这一步的转换不涉及内存拷贝，也不产生额外的内存开销，因此 Spark 官方把这一步叫做“从 Unroll memory 到 Storage memory 的 Transfer（转移）”。

第三步，这些包含 RDD 数据值的 MemoryEntry 和与之对应的 BlockId，会被一起存入 Key 为 BlockId、Value 是 MemoryEntry 引用的链式哈希字典中。因此，LinkedHashMap[BlockId, MemoryEntry]缓存的是关于数据存储的元数据，MemoryEntry 才是真正保存 RDD 数据实体的存储单元。换句话说，大面积占用内存的不是哈希字典，而是一个又一个的 MemoryEntry。

总的来说，RDD 数据分片、Block 和 MemoryEntry 三者之间是一一对应的，当所有的 RDD 数据分片都物化为 MemoryEntry，并且所有的（Block ID, MemoryEntry）对都记

录到 LinkedHashMap 字典之后，RDD 就完成了数据缓存到内存的过程。

这里，你可能会问：“如果内存空间不足以容纳整个 RDD 怎么办？”很简单，强行把大 RDD 塞进有限的内存空间肯定不是明智之举，所以 Spark 会按照 LRU 策略逐一清除字典中最近、最久未使用的 Block，以及其对应的 MemoryEntry。相比频繁的展开、物化、换页所带来的性能开销，缓存下来的部分数据对于 RDD 高效访问的贡献可以说微乎其微。

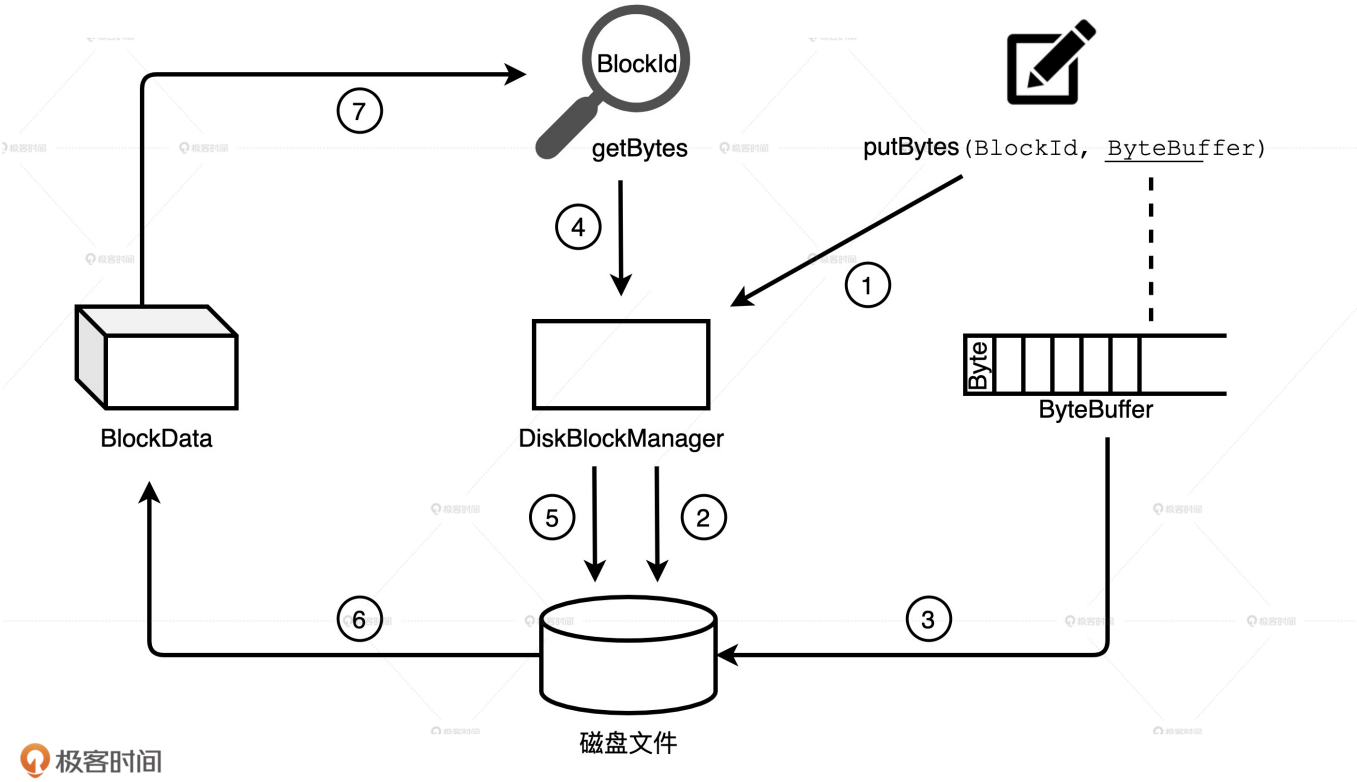
透过 Shuffle 看 DiskStore

相比 MemoryStore，DiskStore 就相对简单很多，因为它并不需要那么多的中间数据结构才能完成数据的存取。**DiskStore 中数据的存取本质上就是字节序列与磁盘文件之间的转换**，它通过 putBytes 方法把字节序列存入磁盘文件，再通过 getBytes 方法将文件内容转换为数据块。

不过，要想完成两者之间的转换，像数据块与文件的对应关系、文件路径等等这些元数据是必不可少的。MemoryStore 采用链式哈希字典来维护类似的元数据，DiskStore 这个狡猾的家伙并没有亲自维护这些元数据，而是请了 DiskBlockManager 这个给力的帮手。

DiskBlockManager 的主要职责就是，记录逻辑数据块 Block 与磁盘文件系统中物理文件的对应关系，每个 Block 都对应一个磁盘文件。同理，每个磁盘文件都有一个与之对应的 Block ID，这就好比货架上的每一件货物都有唯一的 ID 标识。

DiskBlockManager 在初始化的时候，首先根据配置项 spark.local.dir 在磁盘的相应位置创建文件目录。然后，在 spark.local.dir 指定的所有目录下分别创建子目录，子目录的个数由配置项 spark.diskStore.subDirectories 控制，它默认是 64。所有这些目录均用于存储通过 DiskStore 进行物化的数据文件，如 RDD 缓存文件、Shuffle 中间结果文件等。



DiskStore 中数据的存与取

接下来，我们再以 Shuffle 中间文件为例，来说说 DiskStore 与 DiskBlockManager 的交互过程。

Spark 默认采用 SortShuffleManager 来管理 Stages 间的数据分发，在 Shuffle write 过程中，有 3 类结果文件：temp_shuffle_XXX、shuffle_XXX.data 和 shuffle_XXX.index。Data 文件存储分区数据，它是由 temp 文件合并而来的，而 index 文件记录 data 文件内不同分区的偏移地址。Shuffle 中间文件具体指的就是 data 文件和 index 文件，temp 文件作为暂存盘文件最终会被删除。

在 Shuffle write 的不同阶段，Shuffle manager 通过 BlockManager 调用 DiskStore 的 putBytes 方法将数据块写入文件。文件由 DiskBlockManager 创建，文件名就是 putBytes 方法中的 Block ID，这些文件会以“temp_shuffle”或“shuffle”开头，保存在 spark.local.dir 目录下的子目录里。

在 Shuffle read 阶段，Shuffle manager 再次通过 BlockManager 调用 DiskStore 的 getBytes 方法，读取 data 文件和 index 文件，将文件内容转化为数据块，最终这些数据块会通过网络分发到 Reducer 端进行聚合计算。

小结

掌握存储系统是我们进行 Spark 性能调优的关键一步，我们可以分为三步来掌握。

第一步，我们要明确存储系统的服务对象，分别是 RDD 缓存、Shuffle 和广播变量。

RDD 缓存：一些计算成本和访问频率较高的 RDD，可以以缓存的形式物化到内存或磁盘中。这样一来，既可以避免 DAG 频繁回溯的计算开销，也能有效提升端到端的执行性能

Shuffle：Shuffle 中间文件的位置信息，都是由 Spark 存储系统保存并维护的，没有存储系统，Shuffle 是玩不转的

广播变量：利用存储系统，广播变量可以在 Executors 进程范畴内保存全量数据，让任务以 Process local 的本地性级别，来共享广播变量中携带的全量数据。

第二步，我们要搞清楚存储系统的两个重要组件：MemoryStore 和 DiskStore。其中，MemoryStore 用来管理数据在内存中的存取，DiskStore 用来管理数据在磁盘中的存取。

对于存储系统的 3 个服务对象来说，广播变量由 MemoryStore 管理，Shuffle 中间文件的落盘和访问要经由 DiskStore，而 RDD 缓存因为会同时支持内存缓存和磁盘缓存两种模式，所以两种组件都有可能用到。

最后，我们要理解 MemoryStore 和 DiskStore 的工作原理。

MemoryStore 支持对象值和字节数组，统一采用 MemoryEntry 数据抽象对它们进行封装。对象值和字节数组二者之间存在着一种博弈关系，所谓的“以空间换时间”和“以时间换空间”，两者的取舍还要看具体的应用场景。

DiskStore 则利用 DiskBlockManager 维护的数据块与磁盘文件的对应关系，来完成字节序列与磁盘文件之间的转换。

每日一练

1. 结合 RDD 数据存储到 MemoryStore 的过程，你能推演出通过 MemoryStore 通过 getValues/getBytes 方法去访问 RDD 缓存内容的过程吗？
2. 参考 RDD 缓存存储的过程，你能推演出广播变量存入 MemoryStore 的流程吗？

期待在留言区看到你的思考和讨论，我们下一讲见！

提建议

上一篇 05 | 调度系统：“数据不动代码动”到底是什么意思？

下一篇 07 | 内存管理基础：Spark如何高效利用有限的内存空间？

精选留言 (5)

写留言

来世愿做友人 A

2021-03-27

第一题：无论 getBytes 还是 getValues，都是使用 blockId 从文中的 linkedHashMap

获取 memoryEntry，并且都转换成 Iterator 返回
第二题：同理，因为都有那几个 BlockManager 等组件管理，所以，广播变量首先也需要 blockId，查看子类实现有 BroadcastBlockId，格式是broadcast_" + broadcastId + xxx。使用 blockCast 的时候，首先在 driver 端进行存储，广播变量driver端默认是 MEM...
展开

作者回复: Perfect + Awesome！欢迎来群里参与更多讨论哈～ 课程的详情页有读者群的二维码，期待你的加入～

ps：有个小细节，从executors拉取分片、分担driver压力，据我了解，这部分功能，code已经ready，不过应该还没有merge到master。至少之前是这样哈～ 不知道最近有没有什么变化

3 4

Shockang
2021-03-27

1.getBytes/getValues 的实现都比较简单，都是先对LinkedHashMap加锁，通过blockId取出对应的MemoryEntry，然后通过模式匹配，getBytes负责处理序列化的SerializedMemoryEntry，并返回Option[ChunkedByteBuffer]，ChunkedByteBuffer是一个只读字节缓冲区，物理上存储为多个块而不是单个连续数组；getValues负责处理对象值序列DeserializedMemoryEntry，返回一个Iterator...
展开

作者回复: Shock老弟，不得不说，简直太棒了！Perfect x 2，标准答案，天衣无缝，满分💯！

从答案能看出来，这部分源码你已经吃的相当透了，大赞～ 咱们有读者群，不知道你在不在里边，务必加入啊！详情页有入口，实在找不到，加我微信好友，搜rJunior，或者“方块K”，我把你拉进去～

3

超级达达
2021-03-26

这里为什么用到了LinkedHashMap而不是普通的HashMap？什么场景下需要保证访问Block的有序性？

作者回复: 非常好的问题！LinkedHashMap也是一种HashMap的，但在内部用一个双向链表，来维护键值对的顺序，每个键值对同时存储在哈希表、和双向链表中。

我们来看LinkedHashMap特性：

1 插入有序，这个好理解，就是在链表后追加元素
2 访问有序，这个就厉害了。访问有序说的是，对一个kv操作，不管put、get，元素都会被挪到链表的末尾，味道出来了～

在spark rdd cache场景下，第一个特性不重要，重要的是第二个特性。当storage memory不足，spark需要删除rdd cache的时候，遵循的是lru，那么问题来了，它咋实现的lru，答案就是它充分利用LinkedHashMap第二个特性，啥也不用做，就轻松地做到了这一点，机不机智？ 6不6?

1 3

L3nvy
2021-03-26

1. 通过blockId在LinkedHashMap[BlockId, MemoryEntry]中获取MemoryEntry，getBytes返回SerializedMemoryEntry中的ByteBuffer，然后还需要反序列化成迭代器才能被使用；getValues返回DeserializedMemoryEntry中的Array[T]并转换成迭代器使用；

展开

作者回复: Perfect！第二题呢?

Stony.修行僧
2021-03-26

问题1: 应该是在操作 linkedhashmap
问题2: 广播变量值相对比较小，应该是存在在array里面

展开

作者回复: 第一题能再展开说说吗？ 比如具体怎么操作?

第二题再想想哈，参考rdd cache的过程～