



下载APP



## 16 | 内存视角（二）：如何有效避免Cache滥用？

2021-04-19 吴磊

Spark性能调优实战

[进入课程 >](#)**讲述：吴磊**

时长 23:01 大小 21.08M



你好，我是吴磊。

在 Spark 的应用开发中，有效利用 Cache 往往能大幅提升执行性能。

但某一天，有位同学却和我说，自己加了 Cache 之后，执行性能反而变差了。仔细看了这位同学的代码之后，我吓了一跳。代码中充斥着大量的 `.cache`，无论是 RDD，还是 DataFrame，但凡有分布式数据集的地方，后面几乎都跟着个 `.cache`。显然，Cache 滥用是执行性能变差的始作俑者。



实际上，在有些场景中，Cache 是灵丹妙药，而在另一些场合，大肆使用 Cache 却成了饮鸩止渴。那 Cache 到底该在什么时候用、怎么用，都有哪些注意事项呢？今天这一讲，我们先一起回顾 Cache 的工作原理，再来回答这些问题。

## Cache 的工作原理

在🔗[存储系统](#)那一讲，我们其实介绍过 RDD 的缓存过程，只不过当时的视角是以 MemoryStore 为中心，目的在于理解存储系统的工作原理，今天咱们把重点重新聚焦到缓存上来。

Spark 的 Cache 机制主要有 3 个方面需要我们掌握，它们分别是：

缓存的存储级别：它限定了数据缓存的存储介质，如内存、磁盘等

缓存的计算过程：从 RDD 展开到分片以 Block 的形式，存储于内存或磁盘的过程

缓存的销毁过程：缓存数据以主动或是被动的方式，被驱逐出内存或是磁盘的过程

下面，我们一一来看。

### 存储级别

Spark 中的 Cache 支持很多种存储级别，比如 MEMORY\_AND\_DISK\_SER\_2、MEMORY\_ONLY 等等。这些长得差不多的字符串我们该怎么记忆和区分呢？其实，**每一种存储级别都包含 3 个基本要素**。

存储介质：内存还是磁盘，或是两者都有。

存储形式：对象值还是序列化的字节数组，带 SER 字样的表示以序列化方式存储，不带 SER 则表示采用对象值。

副本数量：存储级别名字最后的数字代表拷贝数量，没有数字默认为 1 份副本。

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	R		R		1	
MEMORY_ONLY_2	R		R		2	
MEMORY_ONLY_SER	R			R	1	
MEMORY_ONLY_SER_2	R			R	2	
DISK_ONLY		R		R	1	
DISK_ONLY_2		R		R	2	
DISK_ONLY_3		R		R	3	
MEMORY_AND_DISK	R	R	R	R	1	内存的部分以对象值存储， 磁盘部分序列化
MEMORY_AND_DISK_2	R	R	R	R	2	
MEMORY_AND_DISK_SER	R	R		R	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	R	R		R	2	



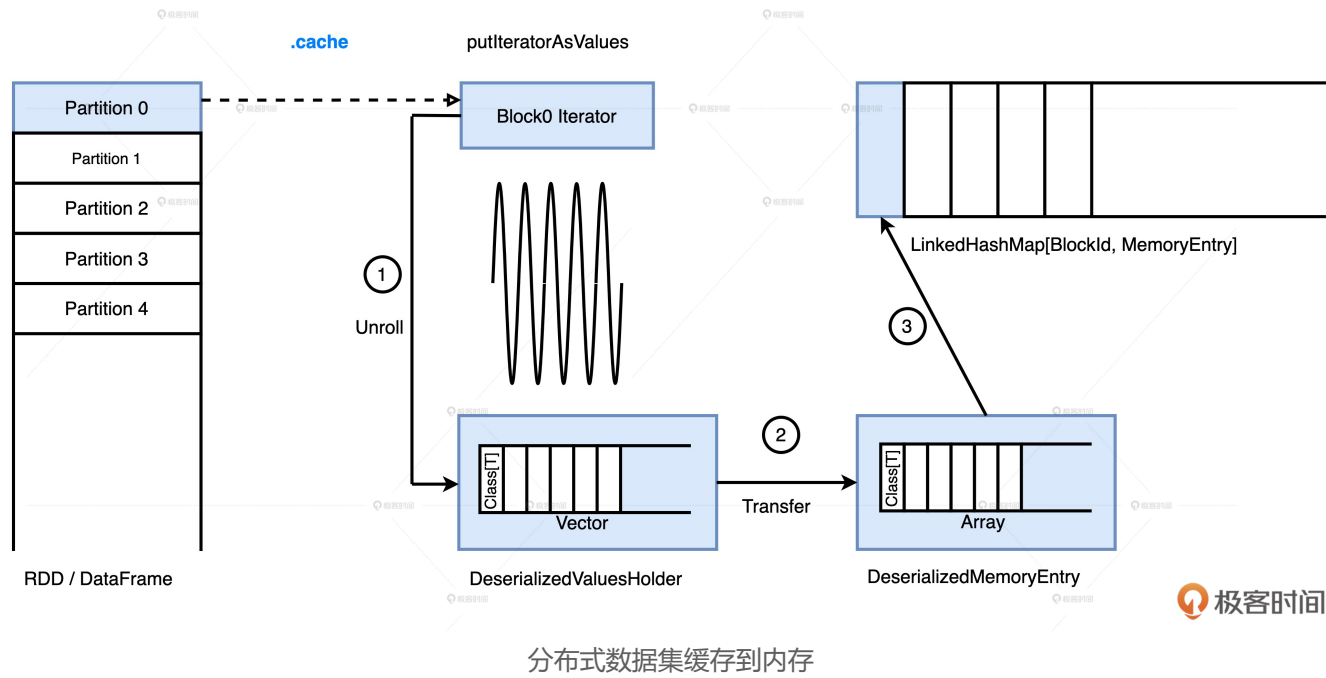
Cache存储级别

当我们将五花八门的存储级别拆解之后就会发现，它们不过是存储介质、存储形式和副本数量这 3 类不同基本元素的排列组合而已。我在上表中列出了目前 Spark 支持的所有存储级别，你可以通过它加深理解。

尽管缓存级别多得让人眼花缭乱，但实际上**最常用的只有两个：MEMORY\_ONLY 和 MEMORY\_AND\_DISK**，它们分别是 **RDD 缓存和 DataFrame 缓存的默认存储级别**。在日常的开发工作中，当你在 RDD 和 DataFrame 之上调用 `.cache` 函数时，Spark 默认采用的就是 MEMORY\_ONLY 和 MEMORY\_AND\_DISK。

### 缓存的计算过程

在 MEMORY\_AND\_DISK 模式下，Spark 会优先尝试把数据集全部缓存到内存，内存不足的情况下，再把剩余的数据落盘到本地。MEMORY\_ONLY 则不管内存是否充足，而是一股脑地把数据往内存里塞，即便内存不够也不会落盘。不难发现，**这两种存储级别都是先尝试把数据缓存到内存**。数据在内存中的存储过程我们在 [第 6 讲](#) 中讲过了，这里我们再一起回顾一下。



无论是 RDD 还是 DataFrame，它们的数据分片都是以迭代器 Iterator 的形式存储的。因此，要把数据缓存下来，我们先得把迭代器展开成实实在在的数据值，这一步叫做 Unroll，如步骤 1 所示。展开的对象值暂时存储在一个叫做 ValuesHolder 的数据结构里，然后转换为 MemoryEntry。转换的实现方式是 toArray，因此它不产生额外的内存开销，这一步转换叫做 Transfer，如步骤 2 所示。最终，MemoryEntry 和与之对应的 BlockID，以 Key、Value 的形式存储到哈希字典（LinkedHashMap）中，如图中的步骤 3 所示。

当分布式数据集所有的数据分片都从 Unroll 到 Transfer，再到注册哈希字典之后，数据在内存中的缓存过程就宣告完毕。

缓存的销毁过程

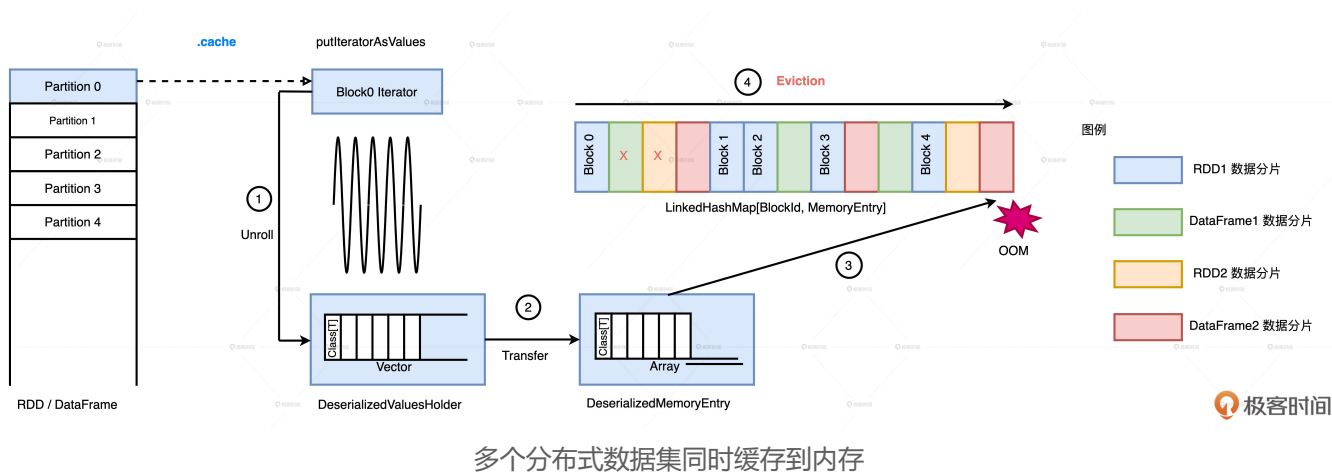
但是很多情况下，应用中数据缓存的需求会超过 Storage Memory 区域的空间供给。虽然缓存任务可以抢占 Execution Memory 区域的空间，但“出来混，迟早是要还的”，随着执行任务的推进，缓存任务抢占的内存空间还是要“吐”出来。这个时候，Spark 就要执行缓存的销毁过程。

你不妨把 Storage Memory 想象成一家火爆的网红餐厅，待缓存的数据分片是一位又一位等待就餐的顾客。当需求大于供给，顾客数量远超餐位数量的时候，Spark 自然要制定一些规则，来合理地“驱逐”那些尸位素餐的顾客，把位置腾出来及时服务那些排队等餐的人。

那么问题来了，Spark 基于什么规则“驱逐”顾客呢？接下来，我就以同时缓存多个分布式数据集的情况为例，带你去分析一下在内存受限的情况下会发生什么。

我们用一张图来演示这个过程，假设 MemoryStore 中存有 4 个 RDD/Data Frame 的缓存数据，这 4 个分布式数据集各自缓存了一些数据分片之后，Storage Memory 区域就被占满了。当 RDD1 尝试把第 6 个分片缓存到 MemoryStore 时，却发现内存不足，塞不进去了。

这种情况下，Spark 就会逐一清除一些“尸位素餐”的 MemoryEntry 来释放内存，从而获取更多的可用空间来存储新的数据分片。这个过程叫做 Eviction，它的中文翻译还是蛮形象的，就叫做驱逐，也就是把 MemoryStore 中那些倒霉的 MemoryEntry 驱逐出内存。



回到刚才的问题，Spark 是根据什么规则选中的这些倒霉蛋呢？这个规则叫作 LRU（Least Recently Used），基于这个算法，最近访问频率最低的那个家伙就是倒霉蛋。因为 [LRU](#) 是比较基础的数据结构算法，笔试、面试的时候经常会考，所以它的概念我就不多说了。

我们要知道的是，Spark 是如何实现 LRU 的。这里，Spark 使用了一个巧妙的数据结构：**LinkedHashMap**，这种数据结构天然地支持 LRU 算法。

LinkedHashMap 使用两个数据结构来维护数据，一个是传统的 HashMap，另一个是双向链表。HashMap 的用途在于快速访问，根据指定的 BlockId，HashMap 以  $O(1)$  的效率返回 MemoryEntry。双向链表则不同，它主要用于维护元素（也就是 BlockId 和 MemoryEntry 键值对）的访问顺序。凡是被访问过的元素，无论是插入、读取还是更新都会被放置到链表的尾部。因此，链表头部保存的刚好都是“最近最少访问”的元素。



如此一来，当内存不足需要驱逐缓存的数据块时，Spark 只利用 LinkedHashMap 就可以做到按照“最近最少访问”的原则，去依次驱逐缓存中的数据分片了。

除此之外，在存储系统那一讲，有同学问 MemoryStore 为什么使用 LinkedHashMap，而不用普通的 Map 来存储 BlockId 和 MemoryEntry 的键值对。我刚才说的就是答案了。

回到图中的例子，当 RDD1 试图缓存第 6 个数据分片，但可用内存空间不足时，Spark 会对 LinkedHashMap 从头至尾扫描，边扫描边记录 MemoryEntry 大小，当倒霉蛋的总大小超过第 6 个数据分片时，Spark 停止扫描。

有意思的是，**倒霉蛋的选取规则遵循“兔子不吃窝边草”，同属一个 RDD 的 MemoryEntry 不会被选中**。就像图中的步骤 4 展示的一样，第一个蓝色的 MemoryEntry 会被跳过，紧随其后打叉的两个 MemoryEntry 被选中。

因此，总结下来，在清除缓存的过程中，Spark 遵循两个基本原则：

LRU：按照元素的访问顺序，优先清除那些“最近最少访问”的 BlockId、MemoryEntry 键值对

兔子不吃窝边草：在清除的过程中，同属一个 RDD 的 MemoryEntry 拥有“赦免权”

## 退化为 MapReduce

尽管有缓存销毁这个环节的存在，Storage Memory 内存空间也总会耗尽，MemoryStore 也总会“驱无可驱”。这个时候，MEMORY\_ONLY 模式就会放弃剩余的数据分片。比如，在 Spark UI 上，你时常会看到 Storage Tab 中的缓存比例低于 100%。而我们从 Storage Tab 也可以观察到，在 MEMORY\_AND\_DISK 模式下，数据集在内存和磁盘中各占一部分比例。

这是因为对于 MEMORY\_AND\_DISK 存储级别来说，当内存不足以容纳所有的 RDD 数据分片的时候，Spark 会把尚未展开的 RDD 分片通过 DiskStore 缓存到磁盘中。DiskStore 的工作原理，我们在存储系统那一讲有过详细介绍，你可以回去看一看，我建议你结合 DiskStore 的知识把 RDD 分片在磁盘上的缓存过程推导出来。

因此，相比 **MEMORY\_ONLY**，**MEMORY\_AND\_DISK** 模式能够保证数据集 100% 地物化到存储介质。对于计算链条较长的 RDD 或是 DataFrame 来说，把数据物化到磁盘也是值得的。但是，我们也不能逢 RDD、DataFrame 就调用 `.cache`，因为在最差的情况下，Spark 的内存计算就会退化为 Hadoop MapReduce 根据磁盘的计算模式。

比如说，你用 DataFrame API 开发应用，计算过程涉及 10 次 DataFrame 之间的转换，每个 DataFrame 都调用 `.cache` 进行缓存。由于 Storage Memory 内存空间受限，MemoryStore 最多只能容纳两个 DataFrame 的数据量。因此，MemoryStore 会有 8 次以 DataFrame 为粒度的换进换出。最终，MemoryStore 存储的是访问频次最高的 DataFrame 数据分片，其他的数据分片全部被驱逐到了磁盘上。也就是说，平均下来，至少有 8 次 DataFrame 的转换都会将计算结果落盘，这不就是 Hadoop 的 MapReduce 计算模式吗？

当然，咱们考虑的是最差的情况，但这也能让我们体会到滥用 Cache 可能带来的隐患和危害了。

## Cache 的用武之地

既然滥用 Cache 危害无穷，那在什么情况下适合使用 Cache 呢？我建议你在做决策的时候遵循以下 2 条基本原则：


如果 RDD/DataFrame/Dataset 在应用中的引用次数为 1，就坚决不使用 Cache

如果引用次数大于 1，且运行成本占比超过 30%，应当考虑启用 Cache

第一条很好理解，我们详细说说第二条。这里咱们定义了一个新概念：**运行成本占比**。它指的是计算某个分布式数据集所消耗的总时间与作业执行时间的比值。我们来举个例子，假设我们有个数据分析的应用，端到端的执行时间为 1 小时。应用中有个 DataFrame 被引用了 2 次，从读取数据源，经过一系列计算，到生成这个 DataFrame 需要花费 12 分钟，那么这个 DataFrame 的运行成本占比应该算作： $12 * 2 / 60 = 40\%$ 。

你可能会说：“作业执行时间好算，直接查看 Spark UI 就好了，DataFrame 的运行时间怎么算呢？”这里涉及一个小技巧，我们可以从现有应用中把 DataFrame 的计算逻辑单拎出来，然后利用 Spark 3.0 提供的 Noop 来精确地得到 DataFrame 的运行时间。假设 df 是那个被引用 2 次的 DataFrame，我们就可以把 df 依赖的所有代码拷贝成一个新的作

业，然后在 df 上调用 Noop 去触发计算。Noop 的作用很巧妙，它只触发计算，而不涉及落盘与数据存储，因此，新作业的执行时间刚好就是 DataFrame 的运行时间。

 复制代码

```
1 //利用noop精确计算DataFrame运行时间
2 df.write
3   .format("noop")
4   .save()
```

你可能会觉得每次计算占比会很麻烦，但只要你对数据源足够了解、对计算 DataFrame 的中间过程心中有数了之后，其实不必每次都去精确地计算运行成本占比，尝试几次，你就能对分布式数据集的运行成本占比估摸得八九不离十了。

## Cache 的注意事项

弄清楚了应该什么时候使用 Cache 之后，我们再来说说 Cache 的注意事项。

首先，我们都知道，.cache是惰性操作，因此在调用.cache之后，需要先用 Action 算子触发缓存的物化过程。但是，我发现很多同学在选择 Action 算子的时候很随意，first、take、show、count 中哪个顺手就用哪个。

这肯定是不对的，**这 4 个算子中只有 count 才会触发缓存的完全物化，而 first、take 和 show 这 3 个算子只会把涉及的数据物化**。举个例子，show 默认只产生 20 条结果，如果我们在.cache 之后调用 show 算子，它只会缓存数据集中这 20 条记录。

选择好了算子之后，我们再来讨论一下怎么 Cache 这个问题。你可能会说：“这还用说吗？在 RDD、DataFrame 后面调用.cache不就得了”。还真没这么简单，我出一道选择题来考考你，如果给定包含数十列的 DataFrame df 和后续的数据分析，你应该采用下表中的哪种 Cache 方式？

 复制代码

```
1 val filePath: String = _
2 val df: DataFrame = spark.read.parquet(filePath)
3
4 //Cache方式一
5 val cachedDF = df.cache
6 //数据分析
```



```
7 cachedDF.filter(col2 > 0).select(col1, col2)
8 cachedDF.select(col1, col2).filter(col2 > 100)
9
10 //Cache方式二
11 df.select(col1, col2).filter(col2 > 0).cache
12 //数据分析
13 df.filter(col2 > 0).select(col1, col2)
14 df.select(col1, col2).filter(col2 > 100)
15
16 //Cache方式三
17 val cachedDF = df.select(col1, col2).cache
18 //数据分析
19 cachedDF.filter(col2 > 0).select(col1, col2)
20 cachedDF.select(col1, col2).filter(col2 > 100)
21
```

我们都知道，由于 Storage Memory 内存空间受限，因此 Cache 应该遵循**最小公共子集原则**，也就是说，开发者应该仅仅缓存后续操作必需的那些数据列。按照这个原则，实现方式 1 应当排除在外，毕竟 df 是一张包含数十列的宽表。

我们再来看第二种 Cache 方式，方式 2 缓存的数据列是 col1 和 col2，且 col2 数值大于 0。第一条分析语句只是把 filter 和 select 调换了顺序；第二条语句 filter 条件限制 col2 数值要大于 100，那么，这个语句的结果就是缓存数据的子集。因此，乍看上去，两条数据分析语句在逻辑上刚好都能利用缓存的数据内容。

但遗憾的是，这两条分析语句都会跳过缓存数据，分别去磁盘上读取 Parquet 源文件，然后从头计算投影和过滤的逻辑。这是为什么呢？究其缘由是，**Cache Manager 要求两个查询的 Analyzed Logical Plan 必须完全一致，才能对 DataFrame 的缓存进行复用。**

Analyzed Logical Plan 是比较初级的逻辑计划，主要负责 AST 查询语法树的语义检查，确保查询中引用的表、列等元信息的有效性。像谓词下推、列剪枝这些比较智能的推理，要等到制定 Optimized Logical Plan 才会生效。因此，即使是同一个查询语句，仅仅是调换了 select 和 filter 的顺序，在 Analyzed Logical Plan 阶段也会被判定为不同的逻辑计划。

因此，为了避免因为 Analyzed Logical Plan 不一致造成的 Cache miss，我们应该采用第三种实现方式，把我们想要缓存的数据赋值给一个变量，凡是在这个变量之上的分析操作，都会完全复用缓存数据。你看，缓存的使用可不仅仅是调用 .cache 那么简单。

除此之外，我们也应当及时清理用过的 Cache，尽早腾出内存空间供其他数据集消费，从而尽量避免 Eviction 的发生。一般来说，我们会用 `unpersist` 来清理弃用的缓存数据，它是 `cache` 的逆操作。`unpersist` 操作支持同步、异步两种模式：

异步模式：调用 `unpersist()` 或是 `unpersist(False)`

同步模式：调用 `unpersist(True)`

在异步模式下，Driver 把清理缓存的请求发送给各个 Executors 之后，会立即返回，并且继续执行用户代码，比如后续的任务调度、广播变量创建等等。在同步模式下，Driver 发送完请求之后，会一直等待所有 Executors 给出明确的结果（缓存清除成功还是失败）。各个 Executors 清除缓存的效率、进度各不相同，Driver 要等到最后一个 Executor 返回结果，才会继续执行 Driver 侧的代码。显然，同步模式会影响 Driver 的工作效率。因此，通常来说，在需要主动清除 Cache 的时候，我们往往采用异步的调用方式，也就是调用 `unpersist()` 或是 `unpersist(False)`。

## 小结

想要有效避免 Cache 的滥用，我们必须从 Cache 的工作原理出发，先掌握 Cache 的 3 个重要机制，分别是存储级别、缓存计算和缓存的销毁过程。

对于存储级别来说，实际开发中最常用到的有两个，`MEMORY_ONLY` 和 `MEMORY_AND_DISK`，它们分别是 RDD 缓存和 DataFrame 缓存的默认存储级别。

对于缓存计算来说，它分为 3 个步骤，第一步是 Unroll，把 RDD 数据分片的 Iterator 物化为对象值，第二步是 Transfer，把对象值封装为 `MemoryEntry`，第三步是把 `BlockId`、`MemoryEntry` 价值对注册到 `LinkedHashMap` 数据结构。

另外，当数据缓存需求远大于 Storage Memory 区域的空间供给时，Spark 利用 `LinkedHashMap` 数据结构提供的特性，会遵循 LRU 和兔子不吃窝边草这两个基本原则来清除内存空间：

LRU：按照元素的访问顺序，优先清除那些“最近最少访问”的 `BlockId`、`MemoryEntry` 键值对

兔子不吃窝边草：在清除的过程中，同属一个 RDD 的 `MemoryEntry` 拥有“赦免权”

其次，我们要掌握使用 Cache 的一般性原则和注意事项，我把它们总结为 3 条：

如果 RDD/DataFrame/Dataset 在应用中的引用次数为 1，我们就坚决不使用 Cache

如果引用次数大于 1，且运行成本占比超过 30%，我们就考虑启用 Cache（其中，运行成本占比的计算，可以利用 Spark 3.0 推出的 noop 功能）

Action 算子要选择 count 才能完全物化缓存数据，以及在调用 Cache 的时候，我们要把待缓存数据赋值给一个变量。这样一来，只要是在这个变量之上的分析操作都会完全复用缓存数据。

## 每日一练

1. 你能结合 DiskStore 的知识，推导出 MEMORY\_AND\_DISK 模式下 RDD 分片缓存到磁盘的过程吗？
2. 你觉得，为什么 Eviction 规则要遵循“兔子不吃窝边草”呢？如果允许同一个 RDD 的 MemoryEntry 被驱逐，有什么危害吗？
3. 对于 DataFrame 的缓存复用，Cache Manager 为什么没有采用根据 Optimized Logical Plan 的方式，你觉得难点在哪里？如果让你实现 Cache Manager 的话，你会怎么做？

期待在留言区看到你的思考和答案，如果你的朋友也正在为怎么使用 Cache 而困扰，也欢迎你把这一讲转发给他。我们下一讲见！

提建议

## 更多课程推荐

# Kafka 核心技术与实战

全面提升你的 Kafka 实战能力

胡夕

Apache Kafka Committer  
老虎证券技术总监



涨价倒计时 🕒

今日订阅 **¥89**，4月29日涨价至 **¥199**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 15 | 内存视角（一）：如何最大化内存的使用效率？

下一篇 17 | 内存视角（三）：OOM都是谁的锅？怎么破？

## 精选留言 (6)

写留言



Fendora范东\_

2021-04-19

还有几个疑问，

1. 驱逐memoryentry的时候，图上两个打叉，我理解如果驱逐一个内存还是不够，就驱逐了两个？
2. linkedhashmap是怎样区分不同RDD的block的，k是blockid，v是memoryentry。没看到所属RDD的属性...

展开 ∨

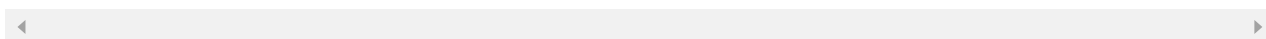
作者回复: 好问题~

1. 是的，一个空间不够，因此继续向后扫描，扫了两个之后，发现空间够了，就停止了。

2. 这个要赞一个~ ㄟ\_ㄟ, 思考很细致。这里咱们图省事, 我没有交代的特别清楚, 实际上, blockId不是String, 而是一个sealed class, 也就是一个类。这个类有不少属性, 其中一个是asRDDId, 这个函数就可以用来获取RDD Id, 从而区分扫描的Block, 是属于哪个RDD的, 从而实现“兔子不吃窝边草”~

3. 对, 现在的设计, 是放在Analyzed LP之后、Optimized LP之前, 相当于是一种逻辑优化, 就像你说的, 能省略掉一些Rules、或者选取一些更优的Rules。

关于Cache Manager的优化, 有个思路是SQL Re-write, 其实这个思路就来源于传统的DBMS, Spark SQL没有这个环节。如果能够用SQL Re-writer, 把Analyzed LP重写, 那么其实potentially, 很多的Analyzed LP都可以共享一份数据。Query Re-writer本身也并不复杂, 参考传统DBMS, 就可以很快实现出一个来。如果Cache Manager参考的, 不再是“纯字符串”的Analyzed LP, 而是重写之后的查询, 想必效果要好很多~



**Fendora范东\_**

2021-04-19

1.M\_A\_D在storage memory可用的情况下, 缓存block过程和M\_O过程一样, 区别在于storage\_memory不够情况下的处理。M\_O是通过LRU来驱逐block来获取可用缓存;而M\_A\_D是通过落盘, 落盘流程就是把unroll出来的block, 通过putBytes()方法直接进行落盘, BlockID为落盘文件名, 便于查找。

2.在M\_O模式下, 为啥不驱逐同一个RDD的memoryentry。我理解, 当前RDD正在缓...  
展开 ✓

作者回复: 答得都对, 具体细节刚刚在上一个thread都展开了, 可以看看哈~



**aof**

2021-05-05

1. 第一题, 在前面广播变量那一节的课后思考题已经回答过整个MemoryStore和DiskStore的缓存过程了哈哈

2. 因为同一个RDD的Block大概率在接下来就会被用到, 因为本身要缓存的就是同一个RDD的其他Block, 如果不这样做的话, 很可能要缓存的Block被缓存到内存了, 却发现要用的Block却被驱逐出内存了...

展开 ✓



**辰**

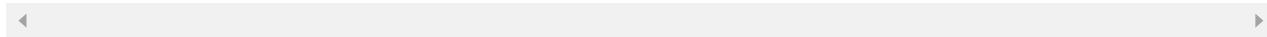


2021-04-21

以及缓存的完全物化的物化是指什么意思呢

展开 ∨

作者回复: 物化 (Materialization) : 把分布式数据集的Iterator迭代器, 展开并存储到内存或是磁盘的过程。

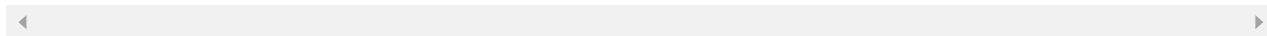
**辰**

2021-04-21

老师, 文中的三种cache方式, 我看都没有调用action算子 (排除第二种不会cache的场景), 是不是意味着都不会缓存落盘的操作

展开 ∨

作者回复: 文中的三种方式, 主要是为了区别示意哈, 目的是为了说清楚缓存复用。实际开发的时候, 肯定是需要Action算子来触发缓存计算的。

**Jay**

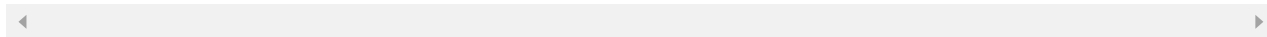
2021-04-19

老师, 如果用的是RDD, 文中的 “Cache方式二” 是不是就没有问题了?

展开 ∨

作者回复: 不是哈, RDD还不如DataFrame, 因为DF会走Spark SQL, Cache Manager是Spark SQL的组件之一。虽然CM效率不高、容易miss, 但是Spark SQL好歹有这么个组件帮他复用Cache。

RDD更惨, 没有哪个组件来帮他做 “复用” 这件事, 所以, 要想充分利用、复用RDD Cache, 你只能用第三种方式, 也就是变量赋值的方式, 明确地引用RDD Cache。



1