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)》**

以往的系列：

- **《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》**

前言

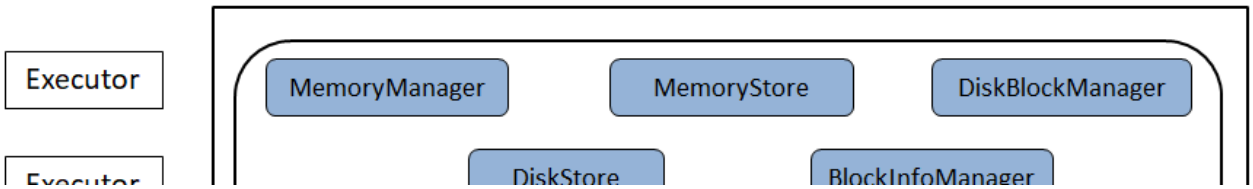
在之前的课中我们讲了Spark的RDD以及整个Spark系统中的一些关键角色：**《【Spark重点难点】你从未深入理解的RDD和关键角色》**。

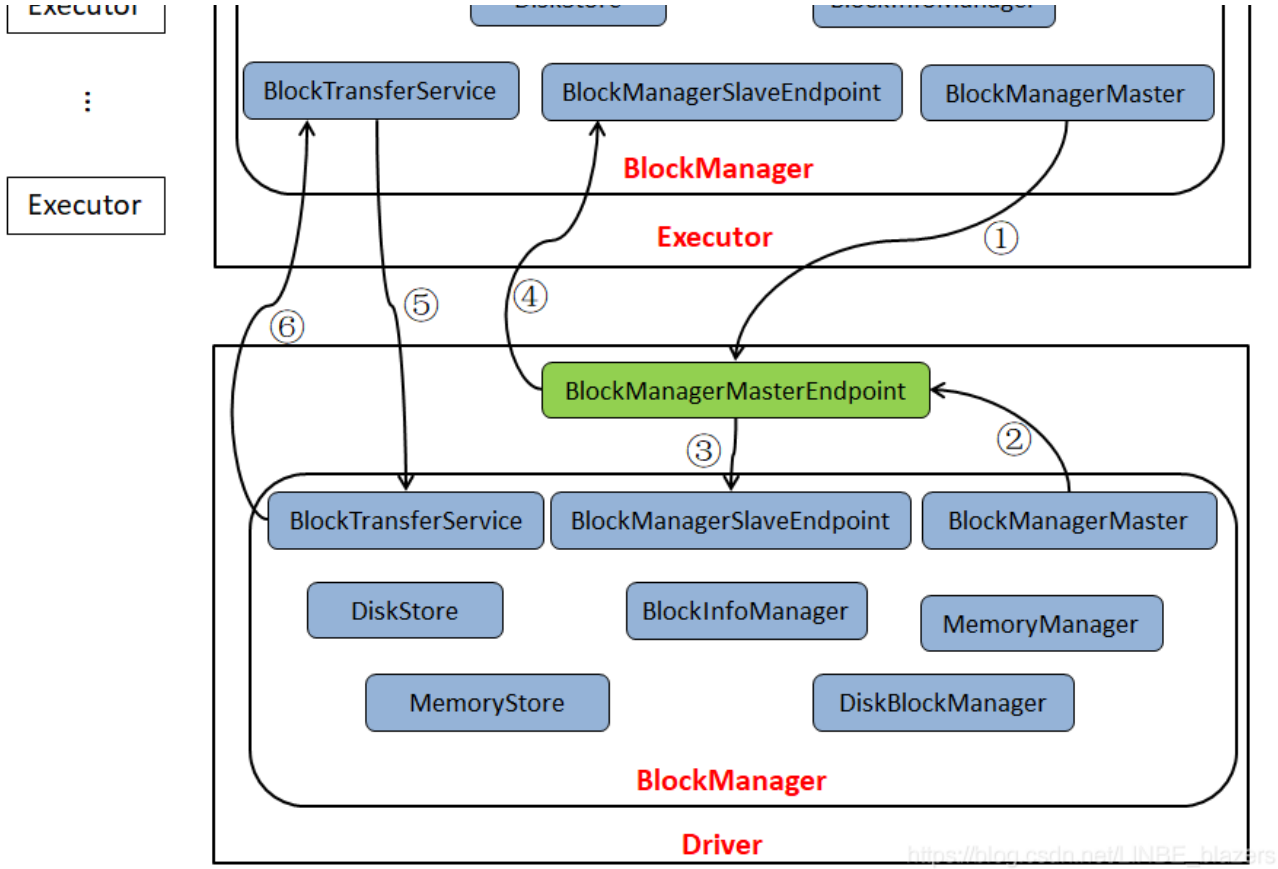
以及Spark中非常重要的一个概念Shuffle:**《【Spark重点难点】你以为的Shuffle和真正的Shuffle》**

无论是在提交任务还是执行任务的过程中，Spark存储体系永远是绕不过去的坎。

Spark为了避免类似Hadoop读写磁盘的IO操作成为性能瓶颈，优先将配置信息、计算结果等数据存入内存，当内存存储不下的时候，可选择性的将计算结果输出到磁盘，为了保证性能，默认都是存储到内存的，这样极大的提高了Spark的计算效率。

我们先用一张图来概括一下Spark的存储体系：





整体体系中重要的角色包括：

- `BlockManager` 是整体存储体系中核心模块
- `DiskBlockManager` 磁盘管理器
- `MemoryStore` 内存存储
- `DiskStore` 磁盘存储

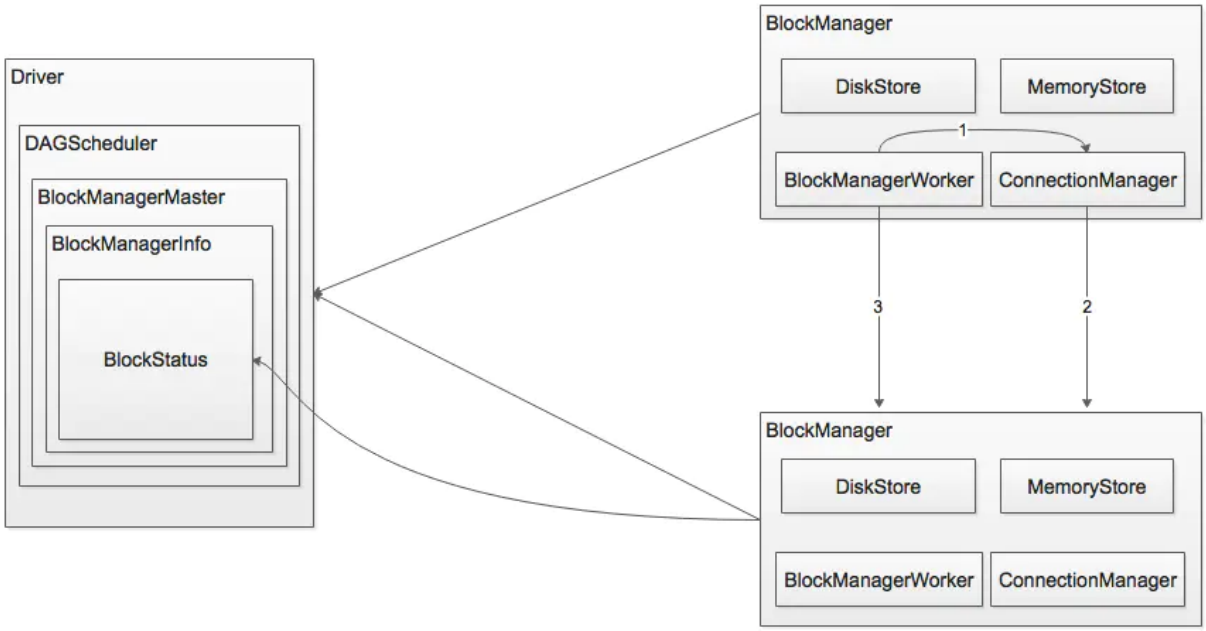
接下来我们依次看看这些角色都是用来做什么的。

BlockManager

BlockManager运行在每个节点上（包括Driver和Executor）。

他提供对本地或远端节点上的内存、磁盘及堆外内存中Block的管理。 存储体系从狭义上来说指的就是 `BlockManager` ，从广义上来说，则包括整个Spark集群中的各个 `BlockManager` 、 `BlockInfoManager` 、 `DiskBlockManager` 、 `DiskStore` 、 `MemoryManager` 、 `MemoryStore` 、对集群中的所有 `BlockManager` 进行管理的 `BlockManagerMaster` 及各个节点上对外提供Block上传与下载服务的 `BlockTransferService` 。

BlockManager 的结构是 Maser-Slave 架构，Master就是Driver上的 BlockManagerMaster ，Slave就是每个Executor上的 BlockManager 。 BlockManagerMaster 负责接受Executor上的 BlockManager 的注册以及管理 BlockManager 的元数据信息。



工作原理

在DAGShceduler中有一个 BlockManagerMaster 对象，该对象的工作就是负责管理全局所有 BlockManager 的元数据，当集群中有 BlockManager 注册完成的时候，其会向 BlockManagerMaster 发送自己元数据信息； BlockManagerMaster 会为 BlockManager 创建一个属于这个 BlockManager 的 BlockManagerInfo ，用于存放 BlockManager 的信息。

在创建 SparkContext 的时候，会调用 SparkEnv.blockManager.initialize 方法实例化 BlockManager 对象，在创建 Executor 对象的时候也会创建 BlockManager 。

当我们的Spark程序启动的时候，首先会创建 SparkContext 对象，在创建 SparkContext 对象的时候就会调用 _env.blockManager.initialize(_applicationId) 创建 BlockManager 对象，这个 BlockManager 就是Driver上的 BlockManager ，它负责管理集群中Executor上的 BlockManager 。

创建BlockManager的关键方法如下，完整的源代码你可以在 BlockManager 这个类中看到。

```
def initialize(appId: String): Unit = {
    //初始化BlockTransferService，其实是它的子类NettyBlockTransferService是下了init方法
    //该方法的作用就是初始化传输服务，通过传输服务可以从不同的节点上拉取Block数据
    blockTransferService.init(this)
    shuffleClient.init(appId)

    //设置block的复制分片策略，由spark.storage.replication.policy指定
    blockReplicationPolicy = {
      val priorityClass = conf.get(
        "spark.storage.replication.policy", classOf[RandomBlockReplicationPolicy]
      val clazz = Utils.classForName(priorityClass)
      val ret = clazz.newInstance.asInstanceOf[BlockReplicationPolicy]
      logInfo(s"Using $priorityClass for block replication policy")
      ret
    }

    //根据给定参数为对对应的Executor封装一个BlockManagerId对象（块存储的唯一标识）
    //executorID：executor的Id，blockTransferService.hostName：传输Block数据的服务的主
    //blockTransferService.port：传输Block数据的服务的主机名
    val id = BlockManagerId(executorId, blockTransferService.hostName, blockTrans

    //调用BlockManagerMaster的registerBlockManager方法向Driver上的BlockManagerMaster
    val idFromMaster = master.registerBlockManager(
      id,
      maxMemory,
      slaveEndpoint)
    //更新BlockManagerId
    blockManagerId = if (idFromMaster != null) idFromMaster else id

    //判断是否开了外部shuffle服务
    shuffleServerId = if (externalShuffleServiceEnabled) {
      logInfo(s"external shuffle service port = $externalShuffleServicePort")
      BlockManagerId(executorId, blockTransferService.hostName, externalShuffleSe
    } else {
      blockManagerId
    }

    // 如果开启了外部shuffle服务，并且该节点是Driver的话就调用registerWithExternalShuffle
    //将BlockManager注册在本地
    if (externalShuffleServiceEnabled && !blockManagerId.isDriver) {
      registerWithExternalShuffleServer()
    }

    logInfo(s"Initialized BlockManager: $blockManagerId")
  }
```

那么 BlockManager 又是如何存储数据的呢？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。

MemoryStore

MemoryStore 负责将Block存储到内存。Spark通过将广播数据、RDD、Shuffle数据存储到内存，减少了对磁盘I/O的依赖，提高了程序的读写效率。

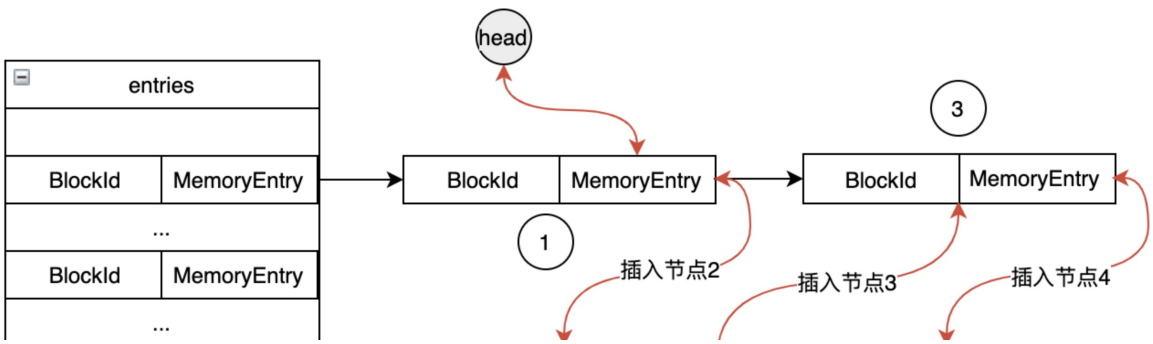
MemoryStore 类实现了一个简单的基于块数据的内存数据库，用来管理需要写入到内存中的块数据。可以按序列化或非序列化的形式存放块数据，存放这两种块数据的数据结构是不同的，但都必须实现 MemoryEntry 这个接口。也就是说： MemoryStore 管理的是以 MemoryEntry 为父接口的内存对象。

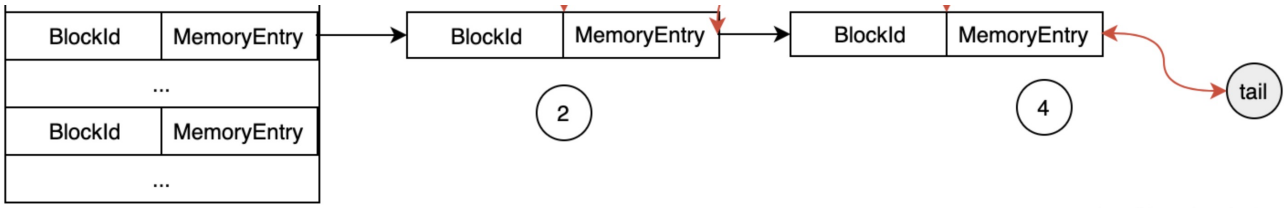
MemoryEntry 是 MemoryStore 中的管理的成员结构。它是一个接口，有两种实现：一种是 DeserializedMemoryEntry ，**用来保存非序列化块数据**；一种是 SerializedMemoryEntry ，**用来保存序列化块数据**。

MemoryStore 如何管理这些 MemoryEntry 对象呢？在当前版本， MemoryStore 通过一个 LinkedHashMap 结构来管理内存对象。也就是说， MemoryStore 是一个 MemoryEntry 类型的 LinkedHashMap 。Spark选择 LinkedHashMap 作为内存管理的数据结构与内存块的淘汰机制有很大的关系。

MemoryStore的数据结构

MemoryStore 通过以 MemoryEntry 对象为元素的 LinkedHashMap 来管理块数据。 LinkedHashMap 是一个有序的HashMap，这样可以按插入顺序来对元素进行管理，此时各个节点构成了一个双向链表。





MemoryStore使用LinkedHashMap按访问元素的先后顺序把访问过的元素放到双向链表的末尾。这其实就形成了一个LRU队列（Least Recently Used队列）。这正是官方文档中提到的：**缓存数据是不可靠的，当内存不够时，会按LRU算法来淘汰内存块。**

需要注意的是，LinkedHashMap是非并发结构，所以在进行其元素的读写操作时，必须加锁。

MemoryEntry的数据结构

MemoryEntry的成员变量有三个：块数据的大小，内存模式（堆内还是堆外），块数据的类标识。MemoryEntry的代码实现如下：

```
// 代码位置：org.apache.spark.storage.memory

private sealed trait MemoryEntry[T] {
  // 块数据大小
  def size: Long
  // 内存模式：ON_HEAP(堆内)，OFF_HEAP(堆外)
  def memoryMode: MemoryMode
  // 数据的类标识
  def classTag: ClassTag[T]
}
```

每个MemoryEntry对象的大小由size来确定。并且可以被保存在 `ON_HEAP(堆内)` 或者 `OFF_HEAP(堆外)` 。

淘汰内存数据

当执行任务或缓存数据空闲内存不足时，可能会释放一部分存储内存，如果对应的RDD的存储级别设置了useDisk，则会把内存中的数据持久化到磁盘上。可以参考： `MemoryStore#evictBlocksToFreeSpace` ：

```
private[spark] def evictBlocksToFreeSpace(
                    blockId: Option[BlockId],
                    space: Long,
                    memoryMode: MemoryMode
): Long = {...}
```

其中的blockId是数据块的id，每个id都对应一个内存块。需要淘汰内存块时，只需要从 LinkedHashMap 的头部选择一个进行删除即可。这就是上面我们提到的LRU内存数据淘汰机制。

DiskStore

DiskStore是BlockStore的另一个实现类，负责管理磁盘数据。简单的说， DiskStore 就是通过 DiskBlockManager 来实现 Block 和相应磁盘文件的映射关系，从而将Block存储到磁盘的文件中。

下面是整体 DiskStore 的类实现：

```
private[spark] class DiskStore(
    conf: SparkConf,
    diskManager: DiskBlockManager,
    securityManager: SecurityManager) extends Logging { // SecurityManager用于提供对

  // 读取磁盘中的Block时，是直接读取还是使用FileChannel的内存镜像映射方法读取的阈值。由spark.s
  private val minMemoryMapBytes = conf.get(config.STORAGE_MEMORY_MAP_THRESHOLD)
  // 使用内存映射读取文件的最大阈值，由配置项spark.storage.memoryMapLimitForTests指定。它是
  private val maxMemoryMapBytes = conf.get(config.MEMORY_MAP_LIMIT_FOR_TESTS)
  // 维护块ID与其对应大小之间的映射关系的ConcurrentHashMap。
  private val blockSizes = new ConcurrentHashMap[BlockId, Long]()
  ...
}
```

我们可以看到 DiskStore 的属性有以下几项：

- conf：即SparkConf
- diskManager：即磁盘Block管理器DiskBlockManager
- minMemoryMapBytes：读到磁盘中的Block时，是直接读取还是使用FileChannel的内存镜像映射方法读取的阈值

此外， `DiskStore` 提供了下面的方法进行操作：

- getSize：获取给定的BlockId所对应Block的大小。
- contains：判断本地磁盘存储路径下是否包含给定BlockId所对应的Block文件。
- remove：删除给定BlockId所对应的Block文件。
- putBytes：用于将BlockId所对应的Block写入磁盘，Block的内容已经封装为ChunkedByteBuffer。
- getBytes：读取给定BlockId所对应的Block，并封装为ChunkedByteBuffer返回。

借用吴磊老师的一句话： `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。`

关于BlockStore的实现还有一种叫做TachyonStore，是基于Tachyon内存分布式文件系统级别的持久化，我们在这里就不做介绍了。感兴趣的读者可以网上搜索一些资料来看。

> Hi，我是王知无，一个大数据领域的原创作者。 放心关注我，获取更多行业的一手消息。