Spark存储原理

Spark的存储介质包括内存和磁盘，其存储采取了主从模式，即Master/Slave模式，整个存储模块使用了前面介绍的RPC的消息通信方式。其中Master负责整个应用程序运行期间的数据块元数据的管理和维护，而Slave一方面负责将本地数据块的状态信息上报给Master，另一方面接收从Master传过来的执行命令，如获取数据状态、删除RDD/数据块等命令。每个Slave都存在数据传输通道，根据需要在Slave之间进行远程数据的读取和写入整体架构如图所示：



根据Spark存储的整体架构图，下面根据数据生命周期过程进行消息通信：

* Block

RDD由不同分区组成，其转换和执行操作都是在独立的分区上各自进行，在本质上每个分区都是和数据块（Block）等价的。在Spark存储管理模块中存取数据的最小单位就是数据Block。在Spark存储管理模块中所管理的集中主要数据块：

1. RDD数据块：用户存储所缓存的RDD数据
2. Shuffle数据块：存储持久化Shuffle数据
3. 广播变量数据块：存储广播变量数据
4. 任务返回结果数据块：存储管理模块内部的任务返回结果，在Driver端获取该数据块即可。
5. 流数据块：在Spark Streaming中，用来存储所接收到的流式数据块

* Driver

在应用程序启动时，SparkContext会创建Driver端的SparkEnv，在该SparkEnv中实例化BlockManagerMaster，其内部创建消息通信的终端点BlockManagerMasterEndpoint。在SparkEnv中创建BlockManagerMaster的代码如下：

*val blockManagerMaster = new BlockManagerMaster(registerOrLookupEndpoint(*

*BlockManagerMaster.DRIVER\_ENDPOINT\_NAME,*

*new BlockManagerMasterEndpoint(rpcEnv, isLocal, conf, listenerBus)),*

*conf, isDriver)*

* Executor

在Executor启动时，也会创建SparkEnv，在该SparkEnv中实例化BlockManager和负责网络数据传输服务的BlockTransferService。在BlockManager初始化的过程中，一方面加入BlockManagerMasterEndpoint终端点的引用，另一方面会创建Executor消息通信的BlockManagerSlaveEndpoint终端点，并把该终端点的引用注册到Driver中，这样Dirver和Executor相互持有通信终端点的引用，可以在应用程序执行过程中进行消息通信。实例化BlockTransferService过程中，使用Netty的数据传输服务，由于该数据传输服务隐藏了集群间不同节点的消息传输操作，可类似于本地数据操作方式进行数据读写，简化网络间数据传输的复杂程度。在SparkEnv中创建BlockTransferService和BlockManager的代码如下：

*val blockTransferService =*

*new NettyBlockTransferService(conf, securityManager, bindAddress, advertiseAddress,*

*blockManagerPort, numUsableCores)*

*val blockManager = new BlockManager(executorId, rpcEnv, blockManagerMaster,*

*serializerManager, conf, memoryManager, mapOutputTracker, shuffleManager,*

*blockTransferService, securityManager, numUsableCores)*

其中BlockManager的初始化代码如下：

*def initialize(appId: String): Unit = {*

*//在Executor中启动远程数据传输服务，根据配置启动传输服务器BlockTransferService*

*//该服务启动后等待其他节点发送请求信息*

*blockTransferService.init(this)*

*shuffleClient.init(appId)*

*blockReplicationPolicy = {*

*val priorityClass = conf.get(*

*"spark.storage.replication.policy", classOf[RandomBlockReplicationPolicy].getName)*

*val clazz = Utils.classForName(priorityClass)*

*val ret = clazz.newInstance.asInstanceOf[BlockReplicationPolicy]*

*logInfo(s"Using $priorityClass for block replication policy")*

*ret*

*}*

*//获取BlockManager编号*

*val id =*

*BlockManagerId(executorId, blockTransferService.hostName, blockTransferService.port, None)*

*//把Executor的BlockManager注册到BlockManagerMaster中，其中包括其终端点*

*val idFromMaster = master.registerBlockManager(*

*id,*

*maxOnHeapMemory,*

*maxOffHeapMemory,*

*slaveEndpoint)*

*blockManagerId = if (idFromMaster != null) idFromMaster else id*

*//获取Shuffle服务编号，如果启动外部Shuffle服务，则加入外部Shuffle服务端口信息，否则使用BlockManager编号*

*shuffleServerId = if (externalShuffleServiceEnabled) {*

*logInfo(s"external shuffle service port = $externalShuffleServicePort")*

*BlockManagerId(executorId, blockTransferService.hostName, externalShuffleServicePort)*

*} else {*

*blockManagerId*

*}*

*//外部外部Shuffle服务启动并且为Executor节点，则注册外部Shuffle服务*

*if (externalShuffleServiceEnabled && !blockManagerId.isDriver) {*

*registerWithExternalShuffleServer()} }*

# 1.整体架构

流程如下所示：

1）BlockManager初始化后，向Driver端发送RegisterBlockManager消息，把Exector的BlockManager和其所包含的BlockManagerSlaveEndpoint引用注册到BlockManagerMaster中

*val idFromMaster = master.registerBlockManager(*

*id,*

*maxOnHeapMemory,*

*maxOffHeapMemory,*

*slaveEndpoint)*

其调用BlockManagerMasterEndpoint#receiveAndReply，源码如下：

*case RegisterBlockManager(blockManagerId, maxOnHeapMemSize, maxOffHeapMemSize, slaveEndpoint) =>*

*context.reply(register(blockManagerId, maxOnHeapMemSize, maxOffHeapMemSize, slaveEndpoint))*

register的操作如下：

*private def register(*

*idWithoutTopologyInfo: BlockManagerId,*

*maxOnHeapMemSize: Long,*

*maxOffHeapMemSize: Long,*

*slaveEndpoint: RpcEndpointRef): BlockManagerId = {*

*//构建BlockId*

*val id = BlockManagerId(*

*idWithoutTopologyInfo.executorId,*

*idWithoutTopologyInfo.host,*

*idWithoutTopologyInfo.port,*

*topologyMapper.getTopologyForHost(idWithoutTopologyInfo.host))*

*//将BlockManager加入到BlockManagerInfo中*

*if (!blockManagerInfo.contains(id)) {*

*blockManagerIdByExecutor.get(id.executorId) match {*

*case Some(oldId) =>*

*removeExecutor(id.executorId)*

*case None =>*

*}*

*blockManagerIdByExecutor(id.executorId) = id*

*blockManagerInfo(id) = new BlockManagerInfo(*

*id, System.currentTimeMillis(), maxOnHeapMemSize, maxOffHeapMemSize, slaveEndpoint)*

*}*

*//将BlockManager加入到Block listenerBus上*

*listenerBus.post(SparkListenerBlockManagerAdded(time, id, maxOnHeapMemSize + maxOffHeapMemSize,*

*Some(maxOnHeapMemSize), Some(maxOffHeapMemSize)))*

*id}*

2） 当BlockManager写入、更新或者删除数据完毕后，发送数据块的最新状态消息

UpdateBlockInfo给BlockManagerMasterEndpoint终端点，由其更新数据块的元数据。该终端点的元数据存放在BlockManagerMasterEndpoint的3个HashMap中，分别如下：

*//该HashMap中存放BlockManagerId与BlockManagerInfo的对应，其中BlockManagerInfo包含Executor内存使用情况、数据块的使用情况、已被缓存的数据块和Executor Endpoint的引用（通过该引用可以向Executor发送消息）*

*private val blockManagerInfo = new mutable.HashMap[BlockManagerId, BlockManagerInfo]*

*//该HashMap中存放ExecutorID和BlockManagerId的对应列表*

*private val blockManagerIdByExecutor = new mutable.HashMap[String, BlockManagerId]*

*//该HashMap存放BlockId和BlockManagerId序列所对应的列表，原因在于一个数据块可能存储多个副本，保存在多个Executor中*

*private val blockLocations = new JHashMap[BlockId, mutable.HashSet[BlockManagerId]]*

在更新数据块的元数据时，更新blockManagerInfo和blockLocations两个列表。

* 在处理blockManagerInfo时，传入的blockManagerId,blockId和storageLevel等参数，通过这些参数判断数据的操作时插入、更新还是删除操作。当插入或删除数据块时，会增加或者更新BlockManagerInfo该数据块的信息。如果是删除数据块，则会在BlockManagerInfo移除该数据块信息，这些操作结束后还会对该Executor内存使用信息进行更新。
* 在处理blockLocations时，根据blockId判断blockLoctions中是否包含该数据块，如果包含该数据块，则根据数据块的操作，进行数据更新时，更新该数据块所在的BlockManagerId信息，当进行数据删除时，则移除该BlockManagerID信息，在删除过程中判断该数据块对应的Executor是否为空，如果不包含该数据块，则进行的是数据添加操作，把该数据块对应的信息加入到blockLocations列表中。

3）应用程序数据存储后，在获取远程节点数据、获取RDD Action的首先位置等操作时需要根据数据块的编号查询数据块所在的位置，此时发送GetLocations或GetLocationsMultipleBl

ockIds等消息给BlockManagerMasterEndpoint终端点，通过对元数据查询获取数据块的位置信息。例如BlockManagerMasterEndpoint中的getLocations方法，该方法获取blockLocations中键值为数据块编号BlockId的所有BlockManagerId序列，代码如下所示：

*private def getLocations(blockId: BlockId): Seq[BlockManagerId] = {*

*if (blockLocations.containsKey(blockId)) blockLocations.get(blockId).toSeq else Seq.empty*

*}*

4）Spark提供删除RDD、数据块和广播变量等方式，当数据需要删除时，提交删除消息给BlockManagerSlaveEndpoint终端点，在该终端点发起删除操作，删除操作一方面需要删除Driver端元数据信息，另一方面需要发送消息给Executor，删除对应的物理数据。下面是RDD的unpersistRDD方法描述其删除过程，类调用关系如图：



首先在SparkContext中调用unpersistRDD方法，在该方法中发送removeRdd消息给BlockMan

agerMasterEndpoint终端点；然后，该终端点接收到消息时，从blockLocations列表中找出该RDD对应的数据存在BlockManagerId列表，查询完毕后，更新blockLocations和blockManagerInfo两个数据块元数据列表；最后把获取的BlockManagerId列表，发送消息给所在BlockManagerSlaveEndpoint终端点，通知其删除Executor上的RDD，删除是调用BlockManager的removeRdd方法，删除在Executor上的RDD所对应的数据块，其中:

BlockManagerMasterEndpoint终端点的removeRdd代码如下：

*private def removeRdd(rddId: Int): Future[Seq[Int]] = {*

*//在blockLocations和blockManagerInfo中删除该RDD的数据元信息*

*//首先，根据RDD编号获取该RDD存储的数据块信息*

*val blocks = blockLocations.asScala.keys.flatMap(\_.asRDDId).filter(\_.rddId == rddId)*

*blocks.foreach { blockId =>*

*//根据数据块信息找出数据块所在的BlockManagerId列表，遍历这些列表并删除*

*//BlockManager包含该数据块的元数据，同时删除blockLocations对应该数据块的元数据*

*val bms: mutable.HashSet[BlockManagerId] = blockLocations.get(blockId)*

*bms.foreach(bm => blockManagerInfo.get(bm).foreach(\_.removeBlock(blockId)))*

*blockLocations.remove(blockId)*

*}*

*//最后，范removeRdd消息给Executor,通知其删除RDD*

*val removeMsg = RemoveRdd(rddId)*

*Future.sequence(*

*blockManagerInfo.values.map { bm =>*

*bm.slaveEndpoint.ask[Int](removeMsg)*

*}.toSeq*

*)*

*}*

BlockManagerMasterEndpoint和Executor之间还进行数据的状态信息，和获取BlockManager副本等信息的交互。下图是Spark存储模块类之间的关系，在整个模块中BlockManager是其核心，不仅提供存储模块处理各种存储方式的读写方法，而且为Shuffle模块提供数据处理等操作接口：



BlockManager存在于Driver端和Executor中，在Driver端的BlockManager保存了数据的元数据信息，而在Executor的BlockManater根据接收到的信息进行如下操作：

* 当Executor的BlockManager接收到读取数据时，根据数据块所在的节点是否本地使用BlockManager不同的方法进行处理。如果在本地，在直接调用MemoryStor的DiskStore中的读取方法getValues/getBytes进行读取。如果是在远程，则调用BlockTransferService的服务进行获取远程节点上的数据。
* 当Executor的BlockManager接收到写入数据时，不需要创建副本，则调用BlockStore的接口方法进行处理，根据数据写入的存储类型，决定调用对应的写入方法

# 2.存储级别

Spark虽然是基于内存的计算，但RDD的数据集不仅可以存在内存中，还可以使用persist或cache方法显示的将RDD的数据集缓存到内存或者磁盘中。下面是RDD的persist的源代码：

*private def persist(newLevel: StorageLevel, allowOverride: Boolean): this.type = {*

*//如果RDD为非NONE的存储级别，该存储级别不能修改*

*if (storageLevel != StorageLevel.NONE && newLevel != storageLevel && !allowOverride) {*

*throw new UnsupportedOperationException(*

*"Cannot change storage level of an RDD after it was already assigned a level")*

*}*

*//当RDD原来的存储级别为NONE,可以对RDD进行持久化处理，在处理前需要先清除SparkContext中原来相关RDD相关的存储元数据，然后加入该RDD的持久化信息*

*if (storageLevel == StorageLevel.NONE) {*

*sc.cleaner.foreach(\_.registerRDDForCleanup(this))*

*sc.persistRDD(this)*

*}*

*storageLevel = newLevel*

*this*

*}*

RDD第一次被计算时，persist方法会根据参数StorageLevel的设置采取特定的缓存策略，当原存储级别为NONE或者新传递进来的存储级别值与原存储级别相等时才进行操作。由于persist操作是action操作的一种，只是改变原RDD的元数据信息，并没有进行数据的存储操作，真正进行的是在RDD的iterator方法中。对于cache方法而言，它只是persist方法的一个特例，即persist方法参数为MEMORY\_ONLY的情况。

在StorageLevel类中，根据useDisk,useMemory,useOffHeap,deserialized,replication 5个参数的组合，Spark 提供了12种存储级别的缓存策略，还可以将RDD持久化到内存、磁盘和外部存储系统，或者是以序列化的方式持久化到内存中，甚至可以在集群的不同节点之间存储多副本，代码如下：

*class StorageLevel private(*

*private var \_useDisk: Boolean,*

*private var \_useMemory: Boolean,*

*private var \_useOffHeap: Boolean,*

*private var \_deserialized: Boolean,*

*private var \_replication: Int = 1)*

这些存储策略都被包含在类StorageLevel中：

*object StorageLevel {*

*val NONE = new StorageLevel(false, false, false, false)*

*val DISK\_ONLY = new StorageLevel(true, false, false, false)*

*val DISK\_ONLY\_2 = new StorageLevel(true, false, false, false, 2)*

*val MEMORY\_ONLY = new StorageLevel(false, true, false, true)*

*val MEMORY\_ONLY\_2 = new StorageLevel(false, true, false, true, 2)*

*val MEMORY\_ONLY\_SER = new StorageLevel(false, true, false, false)*

*val MEMORY\_ONLY\_SER\_2 = new StorageLevel(false, true, false, false, 2)*

*val MEMORY\_AND\_DISK = new StorageLevel(true, true, false, true)*

*val MEMORY\_AND\_DISK\_2 = new StorageLevel(true, true, false, true, 2)*

*val MEMORY\_AND\_DISK\_SER = new StorageLevel(true, true, false, false)*

*val MEMORY\_AND\_DISK\_SER\_2 = new StorageLevel(true, true, false, false, 2)*

*val OFF\_HEAP = new StorageLevel(true, true, true, false, 1)*

可选用的存储级别如下表所示：

|  |  |
| --- | --- |
| 存储级别 | 描述 |
| NONE | 不进行数据 |
| MEMORY\_ONLY | 将RDD作为反序列化的对象存储在JVM中，如果RDD不能被内存装下，一些分区将不会缓存，并且在需要的时候进行重新计算，这是默认的级别 |
| MEMORY\_AND\_DISK | 将RDD作为反序列化的对象存储在JVM中，如果RDD不能被内存装下，超出的分区将被保存在硬盘上，并且在需要时读取 |
| MEMORY\_ONLY\_SER | 将RDD作为序列化对象进行存储（每一分区占用一个字节数据） |
| MEMORY\_AND\_DISK\_SER | 与MEMORY\_ONLY\_SER相似，但是把超出的内存分区存储到磁盘上，而不是在每次需要的时候重新计算 |
| DISK\_ONLY | 只是将RDD分区存储在磁盘上 |
| DISK\_ONLY\_2等带2的 | 与上述的存储级别一样，但是将每一个分区都复制到两个集群节点上 |
| OFF\_HEAP | 可以将RDD存储到分布式内存文件中，如Alluxio |

# 3.RDD存储调用

在分析存储模块原理时，需要明确RDD与数据块Block之间的关系，RDD包含多个Partition，每个Partition对应一个数据块Block，即每个RDD包含一个或者多个数据块Block，每个Block拥有唯一的编号BlockID，对应的数据块编号规则为：”rdd\_”+rddId+”\_”+splitIndex，其中splitIndex为该数据块对应的Partition序列号。

上节中在persist方法中并没有发生数据存储操作动作，实际发生数据操作是任务运行过程中，RDD调用iterator方法时发生的，在调用过程中，会根据数据块Block编号判断是否已经按照指定的存储级别进行存储，如果存在该数据块Block，则从本地或远程节点读取数据，如果不存在该数据块Block，则调用RDD的计算方法得出结果，并把结果按照指定的存储级别进行存储。RDD的iterator方法代码如下：

*final def iterator(split: Partition, context: TaskContext): Iterator[T] = {*

*if (storageLevel != StorageLevel.NONE) {*

*//如果存在存储级别，尝试读取内存数据进行迭代计算*

*getOrCompute(split, context)*

*} else {*

*//不存在存储级别，则直接读取数据进行迭代计算或读取检查点结果进行迭代计算*

*computeOrReadCheckpoint(split, context)*

*}*

*}*

其中调用的getOrCompute方法是存储逻辑的核心，代码如下：

*private[spark] def getOrCompute(partition: Partition, context: TaskContext): Iterator[T] = {*

*//通过RDD编号和Partition序号获取数据块Block的编号*

*val blockId = RDDBlockId(id, partition.index)*

*var readCachedBlock = true*

*//该方法由Executor调用，可使用SparkEnv替代sc.env，根据数据块Block编号先读取数据，然后更新数据，*

*//下面是读取数据的入口点*

*SparkEnv.get.blockManager.getOrElseUpdate(blockId, storageLevel, elementClassTag, () => {*

*//如果数据不在内存，则尝试读取检查点结果进行迭代计算*

*readCachedBlock = false*

*computeOrReadCheckpoint(partition, context)*

*}) match {*

*//对getOrElseUpdtae返回结果进行处理，该结果表示处理成功，记录结果度量信息*

*case Left(blockResult) =>*

*if (readCachedBlock) {*

*val existingMetrics = context.taskMetrics().inputMetrics*

*existingMetrics.incBytesRead(blockResult.bytes)*

*new InterruptibleIterator[T](context, blockResult.data.asInstanceOf[Iterator[T]]) {*

*override def next(): T = {*

*existingMetrics.incRecordsRead(1)*

*delegate.next()*

*}*

*}*

*} else {*

*new InterruptibleIterator(context, blockResult.data.asInstanceOf[Iterator[T]])*

*}*

*//对getOrElseUpdtae返回结果进行处理，该结果表示保存失败，例如数据太大无法放到内存中并且无法保存在磁盘中，*

*//把该结果返回给调用者，尤其决定处理*

*case Right(iter) =>*

*new InterruptibleIterator(context, iter.asInstanceOf[Iterator[T]])*

*}*

*}*

在getOrCompute调用getOrElseUpdate方法，该方法是存储读写数据的入口点：

*def getOrElseUpdate[T](*

*blockId: BlockId,*

*level: StorageLevel,*

*classTag: ClassTag[T],*

*makeIterator: () => Iterator[T]): Either[BlockResult, Iterator[T]] = {*

*//读取数据入口，尝试从本地或者远程读取数据*

*get[T](blockId)(classTag) match {*

*case Some(block) =>*

*return Left(block)*

*case \_ =>*

*// Need to compute the block.*

*}*

*//写数据入口*

*doPutIterator(blockId, makeIterator, level, classTag, keepReadLock = true) match {*

*case None =>*

*val blockResult = getLocalValues(blockId).getOrElse {*

*releaseLock(blockId)*

*throw new SparkException(s"get() failed for block $blockId even though we held a lock")*

*}*

*releaseLock(blockId)*

*Left(blockResult)*

*case Some(iter) =>*

*Right(iter)*

*} }*

# 4.读数据过程

BlockManager的get方法是读数据的入口点，在读取时分为本地读取和远程节点读完两个步骤。整个数据读取的类调用如下图：



本地读取使用getLocalValues方法，在该方法中根据不同的存储级别直接调用不同存储的实现方法，分为内存和磁盘两种读取方式；而远程节点读取使用getRemoteValues方法，在getRemoteValues方法中调用了getRemoteBytes方法，在方法中调用远程数据传输服务类BlockTransferService的fetchBlockSync进行处理，使用Netty的fetchBlocks方法获取数据。

## 4.1 内存读取

在getLocalValues方法中，读取内存中的数据根据返回的是封装成BlockResult类型还是数据流，分别调用MemoryStore的getValues和getBytes两种方法，代码如下：

*if (level.useMemory && memoryStore.contains(blockId)) {*

*val iter: Iterator[Any] = if (level.deserialized) {*

*//如果存储时使用反序列化，则直接读取内存中的数据*

*memoryStore.getValues(blockId).get*

*} else {*

*//如果存储是未使用反序列化，则内存中的数据后做反序列化处理*

*serializerManager.dataDeserializeStream(*

*blockId, memoryStore.getBytes(blockId).get.toInputStream())(info.classTag)*

*}*

*//数据读取完毕后，返回数据及数据块大小，读取方法等信息*

*val ci = CompletionIterator[Any, Iterator[Any]](iter, {*

*releaseLock(blockId, taskAttemptId)*

*})*

*Some(new BlockResult(ci, DataReadMethod.Memory, info.size))*

*}*

在MemoryStore的getValues和getBytes方法中，最终都是通过数据块编号获取内存中的数据，其代码为:

*val entry = entries.synchronized { entries.get(blockId) }*

*//entries <= new LinkedHashMap[BlockID,MemoryEntry[\_]](32,0.75f,true)*

Spark的内存存储其实使用LinkedHashMap来保存数据，在其中保存了记录的插入顺序，在遍历LinkedHashMap元素时，先得到的记录是先插入，其类似于FIFO。

## 4.2 磁盘读取

磁盘读取在getLocalValues方法中，调用的是DiskStore的getBytes方法，在读取磁盘中的数据后需要把这些数据缓存到内存中，其中BlockManager类中的getLocalValues方法代码片段如下：

*if (level.useDisk && diskStore.contains(blockId)) { //从磁盘中获取数据，由于保存到磁盘的数据都是序列化，读取得到的数据也是序列化*

*val diskData = diskStore.getBytes(blockId)*

*val iterToReturn: Iterator[Any] = {*

*if (level.deserialized) {*

*//如果存储级别需反序列化，则把读取的数据反序列化，然后存储到内存*

*val diskValues = serializerManager.dataDeserializeStream(*

*blockId,*

*diskData.toInputStream())(info.classTag)*

*maybeCacheDiskValuesInMemory(info, blockId, level, diskValues)*

*} else {*

*//如果存储级别不需要反序列化，则直接把这些序列化数据存储到内存中*

*val stream = maybeCacheDiskBytesInMemory(info, blockId, level, diskData)*

*.map { \_.toInputStream(dispose = false) }*

*.getOrElse { diskData.toInputStream() }*

*serializerManager.dataDeserializeStream(blockId, stream)(info.classTag)*

*}*

*}*

*//数据读取完之后，返回数据及数据块大小、读取方法等信息*

*val ci = CompletionIterator[Any, Iterator[Any]](iterToReturn, {*

*releaseLockAndDispose(blockId, diskData, taskAttemptId)*

*})*

*Some(new BlockResult(ci, DataReadMethod.Disk, info.size))}*

在Spark中由spark.local.dir设置的磁盘存储的一级目录，默认情况下设置一个一级目录，在每个一级目录下最多创建64个二级子目录。一级目录命名为spark-UUID.randomUUID，其中UUID.randomUUID为16位UUID，二级目录以数据命名，返回00~63，目录中文件的名字是数据块的名称blockId.name，磁盘的目录形式如下图所示：



二级子目录在启动时没有创建，而是当进行数据操作时才创建。在DiskStore中的getBytes方法中，调用DiskBlockManager的getFile方法获取数据块所在文件的句柄。该文件名为数据块的名称，文件所在的一级目录和二级子目录索引值通过文件名的哈希值取模获取，其代码如下：

*def getFile(filename: String): File = {*

*//根据文件名的哈希值获取一级目录和二级子目录索引值，其中一级目录索引值为哈希值与一级目录个数的模，*

*//而二级子目录索引值为哈希值与二级子目录个数的模*

*val hash = Utils.nonNegativeHash(filename)*

*val dirId = hash % localDirs.length*

*val subDirId = (hash / localDirs.length) % subDirsPerLocalDir*

*//先通过一级目录和二级子目录索引值获取该目录，然后判断该子目录是否存在*

*val subDir = subDirs(dirId).synchronized {*

*val old = subDirs(dirId)(subDirId)*

*if (old != null) { old*

*} else {*

*//如果不存在该目录则创建目录，范围0~63*

*val newDir = new File(localDirs(dirId), "%02x".format(subDirId))*

*if (!newDir.exists() && !newDir.mkdir()) {*

*throw new IOException(s"Failed to create local dir in $newDir.")*

*}*

*//判断该文件是否存在，不存在则创建*

*subDirs(dirId)(subDirId) = newDir*

*newDir*

*}*

*}*

*//通过文件的路径获取文件的句柄并返回*

*new File(subDir, filename)*

*}*

获取文件句柄后，读取整个文件内容，以RandomAccessFile的只读方式打开该文件。该文件的打开方式支持从偏移位置开始读取指定大小，如果该文件足够小直接读取，而较大文件则指定通道的文件区域直接映射到内存中进行读取，其代码如下：

*def getBytes(blockId: BlockId): BlockData = {*

*//获取数据块所在文件的句柄*

*val file = diskManager.getFile(blockId.name)*

*val blockSize = getSize(blockId)*

*securityManager.getIOEncryptionKey() match {*

*case Some(key) =>*

*//是否加密，根据文件获取加密数据*

*new EncryptedBlockData(file, blockSize, conf, key)*

*case \_ =>*

*//未加密，直接获取数据*

*new DiskBlockData(minMemoryMapBytes, maxMemoryMapBytes, file, blockSize)*

*}*

其中DiskBlockData是根据数据块所在的文件句柄，进行初始化，其代码如下：

*private class DiskBlockData(*

*minMemoryMapBytes: Long,*

*maxMemoryMapBytes: Long,*

*file: File,*

*blockSize: Long) extends BlockData {*

*override def toInputStream(): InputStream = new FileInputStream(file)*

*override def toNetty(): AnyRef = new DefaultFileRegion(file, 0, size)*

*override def toChunkedByteBuffer(allocator: (Int) => ByteBuffer): ChunkedByteBuffer = {*

*Utils.tryWithResource(open()) { channel =>*

*var remaining = blockSize*

*val chunks = new ListBuffer[ByteBuffer]()*

*while (remaining > 0) {*

*val chunkSize = math.min(remaining, maxMemoryMapBytes)*

*val chunk = allocator(chunkSize.toInt)*

*remaining -= chunkSize*

*JavaUtils.readFully(channel, chunk)*

*chunk.flip()*

*chunks += chunk*

*}*

*new ChunkedByteBuffer(chunks.toArray)*

*}*

*}*

*override def toByteBuffer(): ByteBuffer = {*

*require(blockSize < maxMemoryMapBytes,*

*s"can't create a byte buffer of size $blockSize" +*

*s" since it exceeds ${Utils.bytesToString(maxMemoryMapBytes)}.")*

*Utils.tryWithResource(open()) { channel =>*

*if (blockSize < minMemoryMapBytes) {*

*val buf = ByteBuffer.allocate(blockSize.toInt)*

*JavaUtils.readFully(channel, buf)*

*buf.flip()*

*buf*

*} else {*

*channel.map(MapMode.READ\_ONLY, 0, file.length)*

*}*

*}*

*}*

*override def size: Long = blockSize*

*override def dispose(): Unit = {}*

*private def open() = new FileInputStream(file).getChannel*

*}*

## 4.3 远程节点读取

如果从本地获取不到数据，就会从远程节点读取数据，Spark只提供了Netty远程读取方式，主要由下面两个类处理Netty远程数据读取：

1. NettyBlockTransferService，该类向Shuffle、存储模块提供数据存取的接口，接收到数据存取的命令时，通过Netty的RPC架构发送消息给指定节点，请求进行数据存取操作
2. NettyBlockRpcServer，当Executor启动时，同时会启动RCP监听器，当监听到消息时把消息传递到该类进行处理，消息的内容包括读取数据OpenBlocks和写入数据UploadBlock两种。

使用Netty处理远程数据读取流程如下：

1. Spark远程读取数据入口为getRemoteValues，然后调用getRemoteBytes方法，在该方法中调用getLocations方法先向BlockManagerMasterEndpoint终端点发送GetLocations消息，请求数据块所在的位置消息。

当Driver的终端点接收到请求消息时，根据数据块的编号获取该数据块所在的位置列表，根据是否是本地节点对位置进行排序，其中获取location的代码如下：

*private def getLocations(blockId: BlockId): Seq[BlockManagerId] = {*

*//向BlockManagerMasterEndpoint终端点发送消息，获取数据块所在节点信息*

*val locs = Random.shuffle(master.getLocations(blockId))*

*val (preferredLocs, otherLocs) = locs.partition { loc => blockManagerId.host == loc.host }*

*blockManagerId.topologyInfo match {*

*case None => preferredLocs ++ otherLocs*

*case Some(\_) =>*

*//从获取的节点信息中，优先读取本地节点数据*

*val (sameRackLocs, differentRackLocs) = otherLocs.partition {*

*loc => blockManagerId.topologyInfo == loc.topologyInfo*

*}*

*preferredLocs ++ sameRackLocs ++ differentRackLocs*

*}*

*}*

获取数据块的位置列表后，在BlockManager.getRemoteBytes方法中调用BlockTransferService提供的fetchBlockSync方法进行读取远程数据，具体代码如下：

*def getRemoteBytes(blockId: BlockId): Option[ChunkedByteBuffer] = {*

*......*

*//获取数据块位置列表*

*val locations = getLocations(blockId)*

*val maxFetchFailures = locations.size*

*var locationIterator = locations.iterator*

*while (locationIterator.hasNext) {*

*val loc = locationIterator.next()*

*//通过BlockTransferService提供的fetchBlockSync方法获取远程数据*

*val data = try {*

*blockTransferService.fetchBlockSync(*

*loc.host, loc.port, loc.executorId, blockId.toString).nioByteBuffer()*

*} catch {*

*......*

*}*

*//获取数据后，返回该数据块*

*if (data != null) {*

*return Some(new ChunkedByteBuffer(data))*

*}*

*}*

*None*

*}*

(2) 调用远程数据传输服务BlockTransferService的fetchBlockSync方法后，在该方法中继续调用fetchBlocks方法。该方法是一个抽象方法，实际上调用的是Netty远程数据服务NettyBlockTransferServer类中的fetchBlocks方法。在fetchBlocks方法中，根据远程节点地址和端口创建通信客户端TransportClient，通过该RPC客户端向指定节点发送读取数据消息：

*override def fetchBlocks(host: String,port: Int,execId: String,*

*blockIds: Array[String],listener: BlockFetchingListener,*

*tempShuffleFileManager: TempShuffleFileManager): Unit = {*

*try {*

*val blockFetchStarter = new RetryingBlockFetcher.BlockFetchStarter {*

*override def createAndStart(blockIds: Array[String], listener: BlockFetchingListener) {*

*//根据远程节点和端口创建通信客户端*

*val client = clientFactory.createClient(host, port)*

*//通过该客户端向指定节点发送获取数据信息*

*new OneForOneBlockFetcher(client, appId, execId, blockIds, listener,*

*transportConf, tempShuffleFileManager).start()*

*}*

*}*

*......*

*}*

其中发送读取消息是在OneForOneBlockFetcher类中实现，在该类中的构造函数定义了该消息：

*this.openMessage = new OpenBlocks(appId,execId,blockIds)*

然后在该类的start方法中向RPC客户端发送消息：

*public void start() {*

*//通过客户端发送读取数据块消息*

*client.sendRpc(openMessage.toByteBuffer(), new RpcResponseCallback() {*

*@Override*

*public void onSuccess(ByteBuffer response) {*

*......*

*}*

*@Override*

*public void onFailure(Throwable e) {*

*......*

*}*

*});*

*}*

（3）当远程节点的RPC服务端接收到客户端发送消息时，在NettyBlockRpcServer类中对消息进行匹配。如果是请求读取消息，则调用BlockManager的getBlockData方法读取该节点上的数据，读取的数据块封装为ManagedBuffer序列缓存在内存中，然后使用Netty提供的传输通道，把数据传递请求节点上，完成远程传输任务：

*override def receive(*

*client: TransportClient,*

*rpcMessage: ByteBuffer,*

*responseContext: RpcResponseCallback): Unit = {*

*val message = BlockTransferMessage.Decoder.fromByteBuffer(rpcMessage)*

*logTrace(s"Received request: $message")*

*message match {*

*case openBlocks: OpenBlocks =>*

*val blocksNum = openBlocks.blockIds.length*

*//调用blockManager的getBlockData读取该节点上的数据，读取的数据封装成*

*//byteBuffer序列缓存在内存中*

*val blocks = for (i <- (0 until blocksNum).view)*

*yield blockManager.getBlockData(BlockId.apply(openBlocks.blockIds(i)))*

*val streamId = streamManager.registerStream(appId, blocks.iterator.asJava)*

*logTrace(s"Registered streamId $streamId with $blocksNum buffers")*

*responseContext.onSuccess(new StreamHandle(streamId, blocksNum).toByteBuffer)*

*case uploadBlock: UploadBlock =>*

*// StorageLevel and ClassTag are serialized as bytes using our JavaSerializer.*

*val (level: StorageLevel, classTag: ClassTag[\_]) = {*

*serializer*

*.newInstance()*

*.deserialize(ByteBuffer.wrap(uploadBlock.metadata))*

*.asInstanceOf[(StorageLevel, ClassTag[\_])]*

*}*

*val data = new NioManagedBuffer(ByteBuffer.wrap(uploadBlock.blockData))*

*val blockId = BlockId(uploadBlock.blockId)*

*blockManager.putBlockData(blockId, data, level, classTag)*

*responseContext.onSuccess(ByteBuffer.allocate(0))*

*}*

*}*

# 5.写数据过程

BlockManager的doPutIterator方法是写数据的入口点，在该方法中，根据数据是否缓存到内存中进行处理，如果不缓存到内存中，则调用BlockManager的putIterator方法直接存储到磁盘；如果缓存到内存中，则判断数据存储级别是否进行了反序列化，如果设置反序列化，则说明获取的数据为值类型，调用putIteratorAsValues方法把数据存入内存，如果没有设置反序列化，则获取的数据为字节类型，调用putIteratorAsBytes方法把数据存入内存。在把数据存入内存过程中，需要判断在内存中展开该数据大小是否足够，当足够则调用BlockManager的putArray方法写入内存，否则数据写入到磁盘。

在写入数据完成时，一方面把数据块的元数据发送给Driver端的BlockManagerMasterEndpoint终端点，请求其更新数据元数据，另一方面判断是否需要创建副本，如果需要则调用replicate方法，把数据写到远程节点上，类似于读取远程节点数据，Spark提供Netty方法写数据，整个数据写入类调用关系如下图：



通过上面的方法调用图，BlockManager的doPutIterator方法中根据存储级别和数据类型确定调用的方法，当存储级别为内存时，调用MemoryStore的写入方法；当存储级别为硬盘时，调用DiskStore的写入方法。BlockManager.doPutIterator代码如下所示：

*private def doPutIterator[T](*

*blockId: BlockId,iterator: () => Iterator[T],level: StorageLevel,classTag: ClassTag[T],*

*tellMaster: Boolean = true,keepReadLock: Boolean = false): Option[PartiallyUnrolledIterator[T]] = {*

*//辅助类，用于获取数据块信息，并对写数据结果进行处理*

*doPut(blockId, level, classTag, tellMaster = tellMaster, keepReadLock = keepReadLock) { info =>*

*//将数据写入内存中*

*if (level.useMemory) {*

*//如果设置反序列化，则说明获取的数据为值类型，调用putIteratorAsValues方法将数据写入内存*

*if (level.deserialized) {*

*memoryStore.putIteratorAsValues(blockId, iterator(), classTag) match {*

*//数据写入内存成功，返回数据块大小*

*case Right(s) => size = s*

*case Left(iter) =>*

*// 写入内存失败，如果存储级别设置写入磁盘，则写到磁盘，否则返回结果*

*if (level.useDisk) {*

*diskStore.put(blockId) { channel =>*

*val out = Channels.newOutputStream(channel)*

*serializerManager.dataSerializeStream(blockId, out, iter)(classTag)*

*}*

*size = diskStore.getSize(blockId)*

*} else {*

*iteratorFromFailedMemoryStorePut = Some(iter)*

*}*

*}*

*} else { // 没有设置反序列化，则获取的数据为字节类型，调用putIteratorAsBytes方法把数据存入内存*

*memoryStore.putIteratorAsBytes(blockId, iterator(), classTag, level.memoryMode) match {*

*case Right(s) => size = s*

*case Left(partiallySerializedValues) =>*

*// 写入内存失败，如果存储级别设置写入磁盘，则写到磁盘，否则返回结果*

*if (level.useDisk) {*

*logWarning(s"Persisting block $blockId to disk instead.")*

*diskStore.put(blockId) { channel =>*

*val out = Channels.newOutputStream(channel)*

*partiallySerializedValues.finishWritingToStream(out)*

*}*

*size = diskStore.getSize(blockId)*

*} else {*

*iteratorFromFailedMemoryStorePut = Some(partiallySerializedValues.valuesIterator)*

*}*

*}*

*}*

*//调用DiskStore的put方法写入到磁盘中*

*} else if (level.useDisk) {*

*diskStore.put(blockId) { channel =>*

*val out = Channels.newOutputStream(channel)*

*serializerManager.dataSerializeStream(blockId, out, iterator())(classTag)*

*}*

*size = diskStore.getSize(blockId)*

*}*

*val putBlockStatus = getCurrentBlockStatus(blockId, info)*

*val blockWasSuccessfullyStored = putBlockStatus.storageLevel.isValid*

*if (blockWasSuccessfullyStored) {*

*//成功写入，则把该数据块的元数据发送给driver端*

*info.size = size*

*if (tellMaster && info.tellMaster) {*

*reportBlockStatus(blockId, putBlockStatus)*

*}*

*addUpdatedBlockStatusToTaskMetrics(blockId, putBlockStatus)*

*//如果需要创建副本，则根据数据块编号获取数据复制到其他节点上*

*if (level.replication > 1) {*

*val remoteStartTime = System.currentTimeMillis*

*val bytesToReplicate = doGetLocalBytes(blockId, info)*

*val remoteClassTag = if (!serializerManager.canUseKryo(classTag)) {*

*scala.reflect.classTag[Any]*

*} else {*

*classTag*

*}*

*try {*

*replicate(blockId, bytesToReplicate, level, remoteClassTag)*

*} finally {*

*bytesToReplicate.dispose()*

*}*

*}*

*}*

*iteratorFromFailedMemoryStorePut*

*}*

*}*

在Spark中写入数据分为内存和磁盘两种方式，其对应写入过程如下：

## 5.1 写入内存

下图是Spark内存结构图，内存大致分为两部分：下半部分为已经使用的内存，这些内存存放在entries中，该entries由不同数据块生成的MemoryEntry构成；上半部分为可用内存，用于尝试展开数据块，这些数据块的线程并不是一下子把数据展开到内存中（展开动作是在内存中占位置，而没有真正写入），而是采取步步为营的策略，在每个步中都会先检查内存大小是否足够，如果内存大小不够，则尝试把内存中的数据写入磁盘中，需要释放空间用来存放新写入的数据。而在读数据时了解内存中的数据是以LinkedHashMap保存，其保存了记录的插入顺序，在计算释放空间时以类似FIFO的顺序进行计算。当计算释放空间足够时，则把内存中释放的数据写入磁盘并返回内存足够的结果，当计算出释放所有空间都不足时（但属于同一个RDD中的数据块不能被释放），则返回内存不足的结果。Spark的内存管理相对简单，内存的替换算法类似于FIFO，当机器有足够的内存时，可以明显减少内存释放次数，提供系统处理速度。

如果数据展开成功，则需要把这些数据写入内存中，在写入之前会再次判断内存中展开的数据与数据块估计存储在内存中的大小。如果小于或者等于，则需要多分配它们之间的差值，然后尝试写入到内存中；如果大于，则直接把数据写入到内存中。

在内存处理类MemoryStore中，存入两种写入方法，分别为putIteratorAsValues，针对值类型的数据写入和putIteratorAsBytes针对字节码数据的写入，这两个方法区别在于写入内存的数据类型不同，但是写入内存过程基本类似，下面是putIteratorAsValues的写入过程：

1. 在数据块展开前，为该展开线程获取初始化内存，该内存大小为unrollMemoryThreshold，获取完毕后返回是否成功的结果keepUnrolling。
2. 如果Iterator[T]存在元素且keepUnrolling为真，则继续向前遍历Iterator[T]，内存展开元素的数量elementsUnrolled自增1。如果遍历Iterator[T]到头或者KeepUnrolling为假，则调到步骤4
3. 当memoryCheckPeriod即16次展开动作后，进行一次检查展开的内存大小是否超过当前分配的内存，没有查过则继续展开，不足则根据增长因子计算需要增加的内存大小，然后根据该大小申请增加的内存大小。如果展开大小\*内存增长因子-当前分配的内存大小。如果申请成功，则把内存大小加入到已使用内存中，则该展开线程获取的内存大小为当前展开大小\*内存增长因子
4. 判断数据块是否在内存成功展开，如果展开失败，则记录内存不足并去除，如果展开成功，则继续进行下一步骤
5. 先估算该数据块在内存中存储的大小，然后比较数据块展开的内存和数据块在内存中存储的大小，如果数据展开的内存<=数据块存储的大小，说明展开的内存大小不足以存储数据块，需要申请他们之间的差值，如果申请成功，则调用transferUnrollToStorage方法处理；数据块展开的内存>数据块存储的大小，说明展开的内存大小足以存储数据块。那么先示范多余内存，然后调用transferUnrollToStorage方法处理。
6. 在transferUnrolllToStorage方法中释放该数据块在内存展开的空间，然后判断内存释放足够用于写入数据，如果有足够的内存，则把数据块放到内存entries中，否则返回内存不足，写入内存失败的消息。

*private[storage] def putIteratorAsValues[T](*

*blockId: BlockId,*

*values: Iterator[T],*

*classTag: ClassTag[T]): Either[PartiallyUnrolledIterator[T], Long] = {*

*// 在内存中展开元素的数量*

*var elementsUnrolled = 0*

*// 是否存在足够的内存用于继续展开该数据块，true表示有，false表示内存不足*

*var keepUnrolling = true*

*// 每个展开线程初始化内存大小，这里设置unrollMemoryThreshold*

*val initialMemoryThreshold = unrollMemoryThreshold*

*// 数据块在内存展开时，设置每经过几次展开动作去检查是否申请内存，默认17*

*val memoryCheckPeriod = conf.get(UNROLL\_MEMORY\_CHECK\_PERIOD)*

*// 当前线程保留用于处理展开操作保留的内存大小，初始值initialMemoryThreshold*

*var memoryThreshold = initialMemoryThreshold*

*//内存增长因子，每次请求的内存带下，该因子乘以vector大小，减去memoryThreshold*

*val memoryGrowthFactor = conf.get(UNROLL\_MEMORY\_GROWTH\_FACTOR)*

*// 展开该数据块已使用内存大小*

*var unrollMemoryUsedByThisBlock = 0L*

*// 用于跟踪该数据块展开所使用的内存大小*

*var vector = new SizeTrackingVector[T]()(classTag)*

*// 在数据展开前，根据设置是否为该线程尝试初始化内存*

*keepUnrolling =*

*reserveUnrollMemoryForThisTask(blockId, initialMemoryThreshold, MemoryMode.ON\_HEAP)*

*//获取失败记录日志，获取成功，则把大小加入已使用内存中*

*if (!keepUnrolling) {*

*logWarning(s"Failed to reserve initial memory threshold of " +*

*s"${Utils.bytesToString(initialMemoryThreshold)} for computing block $blockId in memory.")*

*} else {*

*unrollMemoryUsedByThisBlock += initialMemoryThreshold*

*}*

*// 在内存中安全展开该数据块，定期判断是否超过分配内存大小*

*while (values.hasNext && keepUnrolling) {*

*vector += values.next()*

*if (elementsUnrolled % memoryCheckPeriod == 0) {*

*// 每memoryCheckPeriod即16次展开动作，进行一次检查展开的内存大小是否超过当前已经分配的内存大小*

*val currentSize = vector.estimateSize()*

*//展开所需的内存与该线程分配大小比较，内存充足则继续展开，不足则根据增长因子计算*

*//需要增加的内存大小，然后根据该大小申请*

*if (currentSize >= memoryThreshold) {*

*//获取申请增加的内存大小，当前展开大小\*内存增长因子-当前分配的内存大小*

*val amountToRequest = (currentSize \* memoryGrowthFactor - memoryThreshold).toLong*

*//申请需要增加的内存，申请成功，则把内存带下加入已使用内存中*

*keepUnrolling =*

*reserveUnrollMemoryForThisTask(blockId, amountToRequest, MemoryMode.ON\_HEAP)*

*if (keepUnrolling) {*

*unrollMemoryUsedByThisBlock += amountToRequest*

*}*

*// New threshold is currentSize \* memoryGrowthFactor*

*memoryThreshold += amountToRequest*

*}*

*}*

*elementsUnrolled += 1*

*}*

*if (keepUnrolling) {*

*// 成功在内存中展开数据块，估算该数据块在内存中存储的大小*

*val arrayValues = vector.toArray*

*vector = null*

*val entry =*

*new DeserializedMemoryEntry[T](arrayValues, SizeEstimator.estimate(arrayValues), classTag)*

*val size = entry.size*

*//定义内部方法，在该方法中释放该数据块在内存展开的空间，然后判断内存释放足够用于写入数据*

*def transferUnrollToStorage(amount: Long): Unit = {*

*// Synchronize so that transfer is atomic*

*memoryManager.synchronized {*

*releaseUnrollMemoryForThisTask(MemoryMode.ON\_HEAP, amount)*

*val success = memoryManager.acquireStorageMemory(blockId, amount, MemoryMode.ON\_HEAP)*

*assert(success, "transferring unroll memory to storage memory failed")*

*}*

*}*

*// 判断内存释放有足够空间保存该数据块*

*val enoughStorageMemory = {*

*//判断数据块展开的内存大小和数据块存储的大小*

*if (unrollMemoryUsedByThisBlock <= size) {*

*//展开的内存<=数据块存储大小，说明展开内存的大小不足以存储数据块*

*//需要申请他们之间的差值，如果申请成功，则调用transferUnrollToStorage方法处理*

*val acquiredExtra =*

*memoryManager.acquireStorageMemory(*

*blockId, size - unrollMemoryUsedByThisBlock, MemoryMode.ON\_HEAP)*

*if (acquiredExtra) {*

*transferUnrollToStorage(unrollMemoryUsedByThisBlock)*

*}*

*acquiredExtra*

*} else { // unrollMemoryUsedByThisBlock > size*

*//数据块展开的内存>数据块存储的大小，说明展开的内存大小足以存储数据块*

*//那么先释放多余的内存，然后调用transferUnroolToStorage方法处理*

*val excessUnrollMemory = unrollMemoryUsedByThisBlock - size*

*releaseUnrollMemoryForThisTask(MemoryMode.ON\_HEAP, excessUnrollMemory)*

*transferUnrollToStorage(size)*

*true*

*}*

*}*

*if (enoughStorageMemory) {*

*//如果有足够的内存，则把数据块放到内存的entries中，并返回占用内存大小*

*entries.synchronized {*

*entries.put(blockId, entry)*

*}*

*logInfo("Block %s stored as values in memory (estimated size %s, free %s)".format(*

*blockId, Utils.bytesToString(size), Utils.bytesToString(maxMemory - blocksMemoryUsed)))*

*Right(size)*

*} else {*

*//内存不足，则返回该数据块在内存部分展开消息及大小等信息*

*Left(new PartiallyUnrolledIterator(*

*this,*

*MemoryMode.ON\_HEAP,*

*unrollMemoryUsedByThisBlock,*

*unrolled = arrayValues.toIterator,*

*rest = Iterator.empty))*

*}*

*} else {*

*//内存不足无法展开*

*Left(new PartiallyUnrolledIterator(*

*this,*

*MemoryMode.ON\_HEAP,*

*unrollMemoryUsedByThisBlock,*

*unrolled = vector.iterator,*

*rest = values))*

*}*

*}*

## 5.2写入磁盘

Spark写入磁盘的方法调用DiskStore.put方法，该方法提供写入文件的回调方法writeFunc，在该方法中先写入文件句柄，然后把数据序列化为数据流，最后根据回调方法把数据写入文件中，其处理代码如下：

*def put(blockId: BlockId)(writeFunc: WritableByteChannel => Unit): Unit = {*

*......*

*//获取需要写入的文件句柄，参见外部存储系统的读过程*

*val file = diskManager.getFile(blockId)*

*val out = new CountingWritableChannel(openForWrite(file))*

*var threwException: Boolean = true*

*try {*

*//使用回调方法，写入前需要把值类型数据序列化成数据流*

*writeFunc(out)*

*blockSizes.put(blockId.name, out.getCount)*

*threwException = false*

*} finally {...}*

*}*