## 第 4 章 存储体系

天行健，君子以自强不息；地势坤，君子以厚德载物。

——《易经》

本章导读

从本章开始，我们一起来学习Spark的核心知识，掌握好这些内容是阅读全书的关键。无论是Spark的初始化阶段还是任务提交、执行阶段，始终离不开存储体系。笔者将所有涉及存储的内容都集中在本章，以便于对存储体系有好的抽象，也便于读者对其内容有更宏观的认识。

Spark为了避免Hadoop读写磁盘的I/O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，这极大地提升了系统的执行效率。正是因为这一关键决策，才让Spark能在大数据应用中表现出优秀的计算能力。

### 4.1 存储体系概述

4.1.1 块管理器BlockManager的实现

块管理器BlockManager是Spark存储体系中的核心组件，因此本章内容主要围绕BlockManager展开。Driver Application和Executor都会创建BlockManager，BlockManager的实现见代码清单4-1。

代码清单4-1 BlockManager的实现

val diskBlockManager = new DiskBlockManager(this, conf) private val blockInfo = new TimeStampedHashMap[BlockId, BlockInfo] // Actual storage of where blocks are kept private var tachyonInitialized = false private[spark] val memoryStore = new MemoryStore(this, maxMemory) private[spark] val diskStore = new DiskStore(this, diskBlockManager) private[spark] lazy val tachyonStore: TachyonStore = { val storeDir = conf.get("spark.tachyonStore.baseDir", "/tmp\_spark\_tachyon") val appFolderName = conf.get("spark.tachyonStore.folderName") val tachyonStorePath = s"$storeDir/$appFolderName/${this.executorId}" val tachyonMaster = conf.get("spark.tachyonStore.url", "tachyon://localhost:19998") val tachyonBlockManager = new TachyonBlockManager(this, tachyonStorePath, tachyonMaster) tachyonInitialized = true new TachyonStore(this, tachyonBlockManager) } private[spark] val shuffleClient = if (externalShuffleServiceEnabled) { val transConf = SparkTransportConf.fromSparkConf(conf, numUsableCores) new ExternalShuffleClient(transConf, securityManager, securityManager.isAuthenticationEnabled()) } else { blockTransferService } private val slaveActor = actorSystem.actorOf( Props(new BlockManagerSlaveActor(this, mapOutputTracker)), name = "BlockManagerActor" + BlockManager.ID\_GENERATOR.next) private val metadataCleaner = new MetadataCleaner( MetadataCleanerType.BLOCK\_MANAGER, this.dropOldNonBroadcastBlocks, conf) private val broadcastCleaner = new MetadataCleaner( MetadataCleanerType.BROADCAST\_VARS, this.dropOldBroadcastBlocks, conf) private lazy val compressionCodec: CompressionCodec = CompressionCodec.createCodec(conf)

上面代码中声明的BlockInfo：TimeStampedHashMap[BlockId，BlockInfo]，用于Block-Manager缓存BlockId及对应的BlockInfo。从代码清单4-1看到，BlockManager主要由以下部分组成：

·shuffle客户端ShuffleClient；

·BlockManagerMaster（对存在于所有Executor上的BlockManager统一管理）；

·磁盘块管理器DiskBlockManager；

·内存存储MemoryStore；

·磁盘存储DiskStore；

·Tachyon存储TachyonStore；

·非广播Block清理器metadataCleaner和广播Block清理器broadcastCleaner；

·压缩算法实现CompressionCodec。

BlockManager要生效，必须要初始化，它的初始化方法见代码清单4-2。BlockManager的初始化步骤如下：

1）BlockTransferService的初始化和ShuffleClient的初始化（具体参见4.2节）。Shuffle-Client默认是BlockTransferService，当有外部的ShuffleService时，调用外部ShuffleService的初始化方法。

2）BlockManagerId和ShuffleServerId的创建。当有外部的ShuffleService时，创建新的BlockManagerId，否则ShuffleServerId默认使用当前BlockManager的BlockManagerId。

3）向BlockManagerMaster注册BlockManagerId，具体实现见4.3.3节（当有外部的ShuffleService时，还需要向BlockManagerMaster注册ShuffleServerId）。

代码清单4-2 BlockManager的初始化

def initialize(appId: String): Unit = { blockTransferService.init(this) shuffleClient.init(appId) blockManagerId = BlockManagerId( executorId, blockTransferService.hostName, blockTransferService.port) shuffleServerId = if (externalShuffleServiceEnabled) { BlockManagerId(executorId, blockTransferService.hostName, externalShuffleServicePort) } else { blockManagerId } master.registerBlockManager(blockManagerId, maxMemory, slaveActor) // Register Executors' configuration with the local shuffle service, if one should exist. if (externalShuffleServiceEnabled && !blockManagerId.isDriver) { registerWithExternalShuffleServer() } }

4.1.2 Spark存储体系架构

在详细介绍存储体系之前，我们先用图4-1说明Spark存储体系的架构。

这里对图4-1中的调用关系做个说明：

·记号①表示Executor的BlockManager与Driver的BlockManager进行消息通信，例如，注册BlockManager、更新Block信息、获取Block所在的BlockManager、删除Executor等；

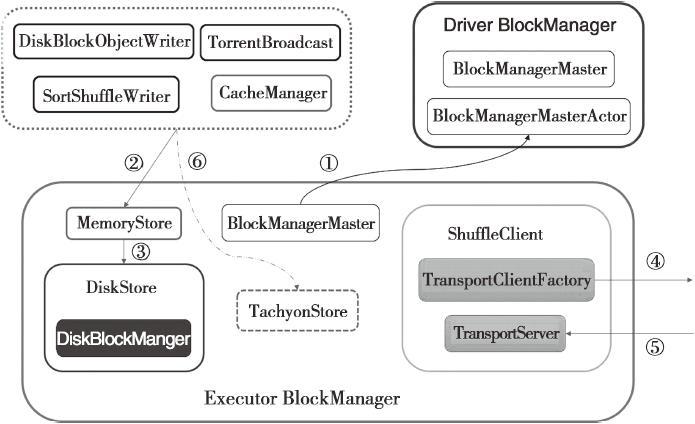


图4-1 Spark存储体系架构

·记号②表示对BlockManager的读操作（例如get、doGetLocal以及BlockManager内部进行的MemoryStore、DiskStore、TachyonStore的getBytes、getValues等操作）和写操作（例如doPut、putSingle、putBytes以及BlockManager内部进行的MemoryStore、DiskStore、TachyonStore的putBytes、putArray、putIterator等操作）；

·记号③表示当MemoryStore的内存不足时，写入DiskStore，而DiskStore实际依赖于DiskBlockManager；

·记号④表示通过访问远端节点的Executor的BlockManager中的TransportServer提供的RPC服务下载或者上传Block；

·记号⑤表示远端节点的Executor的BlockManager访问本地Executor的BlockManager中的TransportServer提供的RPC服务下载或者上传Block；

·记号⑥表示当存储体系选择Tachyon作为存储时，对于BlockManager的读写操作实际调用了TachyonStore的putBytes、putArray、putIterator、getBytes、getValues等。

Spark目前支持HDFS、Amazon S3两种主流分布式存储系统，还使用也诞生于UCBerkeley的AMP实验室的Tachyon这种高效的分布式文件系统作为缓存。

Spark定义了抽象类BlockStore，用于制定所有存储类型的规范。目前BlockStore的具体实现包括MemoryStore、DiskStore和TachyonStore。BlockStore的继承体系如图4-2所示。

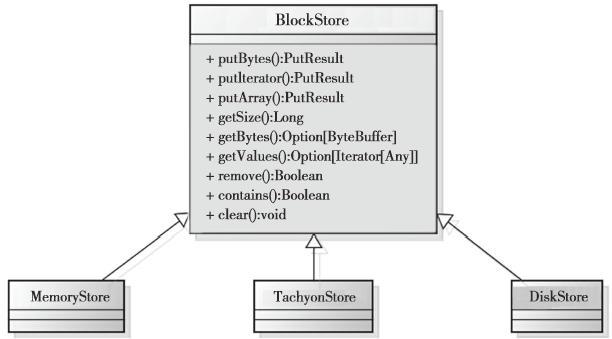


图4-2 BlockStore继承体系

### 4.2 shuffle服务与客户端

读者可能奇怪：为什么需要把由Netty实现的网络服务组件也放到存储体系里面？这是由于Spark是分布式部署的，每个Task最终都运行在不同的机器节点上。map任务的输出结果直接存储到map任务所在机器的存储体系中，reduce任务极有可能不在同一机器上运行，所以需要远程下载map任务的中间输出。因此将ShuffleClient放到存储体系是最合适的。

ShuffleClient并不像它的名字一样，是shuffle的客户端，它不光是将shuffle文件上传到其他Executor或者下载到本地的客户端，也提供了可以被其他Executor访问的shuffle服务。读到这里，熟悉Hadoop YARN的读者可能已经发现Spark与Hadoop一样，都采用Netty作为shuffle server。从代码清单4-1可知，当有外部的ShuffleClient时，新建ExternalShuffleClient，否则默认为BlockTransferService。BlockTransferService只有在其init方法被调用，即被初始化后才提供服务。以默认的NettyBlockTransferService的init方法为例，见代码清单4-3。NettyBlockTransferService的初始化步骤如下：

1）创建RpcServer；

2）构造TransportContext；

3）创建RPC客户端工厂TransportClientFactory；

4）创建Netty服务器TransportServer，可以修改属性spark.blockManager.port（默认为0，表示随机选择）改变TransportServer的端口。

代码清单4-3 NettyBlockTransferService的初始化

override def init(blockDataManager: BlockDataManager): Unit = { val (rpcHandler: RpcHandler, bootstrap: Option[TransportClientBootstrap]) = { val nettyRpcHandler = new NettyBlockRpcServer(serializer, blockDataManager) if (!authEnabled) { (nettyRpcHandler, None) } else { (new SaslRpcHandler(nettyRpcHandler, securityManager), Some(new SaslClientBootstrap(transportConf, conf.getAppId, securityManager))) } } transportContext = new TransportContext(transportConf, rpcHandler) clientFactory = transportContext.createClientFactory(bootstrap.toList) server = transportContext.createServer(conf.getInt("spark.blockManager.port", 0)) appId = conf.getAppId logInfo("Server created on " + server.getPort) }

接下来我们逐步讲解Block的RPC服务，构造TransportContext，创建RPC客户端工厂TransportClientFactory，创建Netty服务器TransportServer的实现。此外还会介绍reduce任务是如何拉取map任务中间结果的（即shuffle过程的数据传输）。

4.2.1 Block的RPC服务

当map任务与reduce任务处于不同节点时，reduce任务需要从远端节点下载map任务的中间输出，因此NettyBlockRpcServer提供打开，即下载Block文件的功能；一些情况下，为了容错，需要将Block的数据备份到其他节点上，所以NettyBlockRpcServer还提供了上传Block文件的RPC服务，NettyBlockRpcServer的实现见代码清单4-4。

代码清单4-4 NettyBlockRpcServer的实现

class NettyBlockRpcServer( serializer: Serializer, blockManager: BlockDataManager) extends RpcHandler with Logging { private val streamManager = new OneForOneStreamManager() override def receive( client: TransportClient, messageBytes: Array[Byte], responseContext: RpcResponseCallback): Unit = { val message = BlockTransferMessage.Decoder.fromByteArray(messageBytes) logTrace(s"Received request: $message") message match { case openBlocks: OpenBlocks => val blocks: Seq[ManagedBuffer] = openBlocks.blockIds.map(BlockId.apply).map(blockManager.getBlockData) val streamId = streamManager.registerStream(blocks.iterator) logTrace(s"Registered streamId $streamId with ${blocks.size} buffers") responseContext.onSuccess(new StreamHandle(streamId, blocks.size).toByteArray) case uploadBlock: UploadBlock => val level: StorageLevel = serializer.newInstance().deserialize(ByteBuffer.wrap(uploadBlock.metadata)) val data = new NioManagedBuffer(ByteBuffer.wrap(uploadBlock.blockData)) blockManager.putBlockData(BlockId(uploadBlock.blockId), data, level) responseContext.onSuccess(new Array[Byte](0)) } } override def getStreamManager(): StreamManager = streamManager }

4.2.2 构造传输上下文TransportContext

TransportContext用于维护传输上下文，它的构造器如下。

public TransportContext(TransportConf conf, RpcHandler rpcHandler) { this.conf = conf; this.rpcHandler = rpcHandler; this.encoder = new MessageEncoder(); this.decoder = new MessageDecoder(); }

TransportContext既可以创建Netty服务，也可以创建Netty访问客户端。TransportContext的组成如下：

·TransportConf：主要控制Netty框架提供的shuffle的I/O交互的客户端和服务端线程数量；

·RpcHandler：负责shuffle的I/O服务端在接收到客户端的RPC请求后，提供打开Block或者上传Block的RPC处理，此处即为NettyBlockRpcServer；

·decoder：在shuffle的I/O服务端对客户端传来的ByteBuf进行解析，防止丢包和解析错误；

·encoder：在shuffle的I/O客户端对消息内容进行编码，防止服务端丢包和解析错误。

00128.jpeg

问题

为什么需要MessageEncoder和MessageDecoder？因为在基于流的传输里（比如TCP/IP），接收到的数据首先会被存储到一个socket接收缓冲里。不幸的是，基于流的传输并不是一个数据包队列，而是一个字节队列。即使发送了2个独立的数据包，操作系统也不会作为2个消息处理，而仅仅认为是一连串的字节。因此不能保证远程写入的数据会被准确地读取。举个例子，假设操作系统的TCP/TP协议栈已经接收了3个数据包：ABC、DEF、GHI。由于基于流传输的协议的这种统一的性质，在应用程序读取数据时很可能性被分成下面的片段：AB、CDEFG、H、I。因此，接收方不管是客户端还是服务端，都应该把接收到的数据整理成一个或者多个更有意义并且让程序的逻辑更好理解的数据。

4.2.3 RPC客户端工厂TransportClientFactory

TransportClientFactory是创建Netty客户端TransportClient的工厂类，TransportClient用于向Netty服务端发送RPC请求。TransportContext的createClientFactory方法用于创建TransportClientFactory，实现如下。

public TransportClientFactory createClientFactory(List bootstraps) { return new TransportClientFactory(this, bootstraps); }

从代码清单4-5可以看到，TransportClientFactory由以下部分组成：

·clientBootstraps：用于缓存客户端列表；

·connectionPool：用于缓存客户端连接；

·numConnectionsPerPeer：节点之间取数据的连接数，可以使用属性spark.shuffle.io.numConnectionsPerPeer来配置，默认为1；

·socketChannelClass：客户端channel被创建时使用的类，可以使用属性spark.shuffle.io.mode来配置，默认为NioSocketChannel；

·workerGroup：根据Netty的规范，客户端只有work组，所以此处创建workerGroup，实际是NioEventLoopGroup；

·pooledAllocator：汇集ByteBuf但对本地线程缓存禁用的分配器。

TransportClientFactory里大量使用了NettyUtils，关于NettyUtils的具体实现，请看附录G。

代码清单4-5 TransportClientFactory的实现

public TransportClientFactory( TransportContext context, List clientBootstraps) { this.context = Preconditions.checkNotNull(context); this.conf = context.getConf(); this.clientBootstraps = Lists.newArrayList(Preconditions.checkNotNull(clientBootstraps)); this.connectionPool = new ConcurrentHashMap (); this.numConnectionsPerPeer = conf.numConnectionsPerPeer(); this.rand = new Random(); IOMode ioMode = IOMode.valueOf(conf.ioMode()); this.socketChannelClass = NettyUtils.getClientChannelClass(ioMode); this.workerGroup = NettyUtils.createEventLoop(ioMode, conf.clientThreads(), "shuffle-client"); this.pooledAllocator = NettyUtils.createPooledByteBufAllocator( conf.preferDirectBufs(), false /\* allowCache \*/, conf.clientThreads()); }

00130.jpeg

提示

NIO是指Java中New IO的简称，其特点包括：为所有的原始类型提供（Buffer）缓存支持；字符集编码解码解决方案；提供一个新的原始I/O抽象Channel，支持锁和内存映射文件的文件访问接口；提供多路非阻塞式（non-bloking）的高伸缩性网络I/O。其具体使用属于Java语言的范畴，本文不过多介绍。

4.2.4 Netty服务器TransportServer

TransportServer提供了Netty实现的服务器端，用于提供RPC服务（比如上传、下载等）。创建TransportServer的代码如下。

public TransportServer createServer(int port) { return new TransportServer(this, port); }

TransportServer的构造器实现如下。

public TransportServer(TransportContext context, int portToBind) { this.context = context; this.conf = context.getConf(); init(portToBind); }

上面代码中的init方法用于对TransportServer初始化，通过使用Netty框架的Event-LoopGroup、ServerBootstrap等API创建shuffle的I/O交互的服务端，init的主要代码见代码清单4-6。

代码清单4-6 init的主要代码

private void init(int portToBind) { IOMode ioMode = IOMode.valueOf(conf.ioMode()); EventLoopGroup bossGroup = NettyUtils.createEventLoop(ioMode, conf.serverThreads(), "shuffle-server"); EventLoopGroup workerGroup = bossGroup; PooledByteBufAllocator allocator = NettyUtils.createPooledByteBufAllocator( conf.preferDirectBufs(), true /\* allowCache \*/, conf.serverThreads()); bootstrap = new ServerBootstrap() .group(bossGroup, workerGroup) .channel(NettyUtils.getServerChannelClass(ioMode)) .option(ChannelOption.ALLOCATOR, allocator) .childOption(ChannelOption.ALLOCATOR, allocator); bootstrap.childHandler(new ChannelInitializer () { @Override protected void initChannel(SocketChannel ch) throws Exception { context.initializePipeline(ch); } }); channelFuture = bootstrap.bind(new InetSocketAddress(portToBind)); channelFuture.syncUninterruptibly(); port = ((InetSocketAddress) channelFuture.channel().localAddress()).getPort(); }

ServerBootstrap的childHandler方法调用了TransportContext的initializePipeline。initia-lizePipeline中创建了TransportChannelHandler，并将它绑定到SocketChannel的pipeline的handler中，见代码清单4-7。

代码清单4-7 initializePipeline方法的实现

public TransportChannelHandler initializePipeline(SocketChannel channel) { try { TransportChannelHandler channelHandler = createChannelHandler(channel); channel.pipeline() .addLast("encoder", encoder) .addLast("frameDecoder", NettyUtils.createFrameDecoder()) .addLast("decoder", decoder) .addLast("handler", channelHandler); return channelHandler; } catch (RuntimeException e) { logger.error("Error while initializing Netty pipeline", e); throw e; } } private TransportChannelHandler createChannelHandler(Channel channel) { TransportResponseHandler responseHandler = new TransportResponseHandler(channel); TransportClient client = new TransportClient(channel, responseHandler); TransportRequestHandler requestHandler = new TransportRequestHandler(channel, client,rpcHandler); return new TransportChannelHandler(client, responseHandler, requestHandler); }

00133.jpeg

注意

本节很多代码都是通过使用Netty API来实现的，有兴趣的读者可以去http://netty.io/查阅API的使用。一些读者可能注意到Spark在使用Netty时，都是用Java作为编程语言，实际上也可以使用Scala作为编程语言的。这个问题，笔者不了解其发生的背景，如果有读者知道，希望能及时通知笔者。

4.2.5 获取远程shuffle文件

NettyBlockTransferService的fetchBlocks方法用于获取远程shuffle文件，实际是利用NettyBlockTransferService中创建的Netty服务，见代码清单4-8。

代码清单4-8 获取远端节点上的shuffle文件

override def fetchBlocks( host: String, port: Int, execId: String, blockIds: Array[String], listener: BlockFetchingListener): Unit = { val blockFetchStarter = new RetryingBlockFetcher.BlockFetchStarter { override def createAndStart(blockIds: Array[String], listener: BlockFetchingListener) { val client = clientFactory.createClient(host, port) new OneForOneBlockFetcher(client, appId, execId, blockIds.toArray, listener).start() } } val maxRetries = transportConf.maxIORetries() if (maxRetries > 0) { new RetryingBlockFetcher(transportConf, blockFetchStarter, blockIds, listener).start() } else { blockFetchStarter.createAndStart(blockIds, listener) } }

4.2.6 上传shuffle文件

NettyBlockTransferService的uploadBlock方法用于上传shuffle文件到远程Executor，实际也是利用NettyBlockTransferService中创建的Netty服务，见代码清单4-9。Netty-BlockTransferService上传Block的步骤如下：

1）创建Netty服务的客户端，客户端连接的hostname和port正是我们随机选择的BlockManager的hostname和port。

2）将Block的存储级别StorageLevel序列化。

3）将Block的ByteBuffer转化为数组，便于序列化。

4）将appId、execId、blockId、序列化的StorageLevel、转换为数组的Block封装为UploadBlock，并将UploadBlock序列化为字节数组。

5）最终调用Netty客户端的sendRpc方法将字节数组上传，回调函数RpcResponse-Callback根据RPC的结果更改上传状态。

代码清单4-9 上传Block到远端节点

override def uploadBlock( hostname: String, port: Int, execId: String, blockId: BlockId, blockData: ManagedBuffer, level: StorageLevel): Future[Unit] = { val result = Promise[Unit]() val client = clientFactory.createClient(hostname, port) // StorageLevel is serialized as bytes using our JavaSerializer. Everything else is encoded // using our binary protocol. val levelBytes = serializer.newInstance().serialize(level).array() // Convert or copy nio buffer into array in order to serialize it. val nioBuffer = blockData.nioByteBuffer() val array = if (nioBuffer.hasArray) { nioBuffer.array() } else { val data = new Array[Byte](nioBuffer.remaining()) nioBuffer.get(data) data } client.sendRpc(new UploadBlock(appId, execId, blockId.toString, levelBytes, array).toByteArray, new RpcResponseCallback { override def onSuccess(response: Array[Byte]): Unit = { logTrace(s"Successfully uploaded block $blockId") result.success() } override def onFailure(e: Throwable): Unit = { logError(s"Error while uploading block $blockId", e) result.failure(e) } }) result.future }

### 4.3 BlockManagerMaster对BlockManager的管理

Driver上的BlockManagerMaster对存在于Executor上的BlockManager统一管理，比如Executor需要向Driver发送注册BlockManager、更新Executor上Block的最新信息、询问所需要Block目前所在的位置以及当Executor运行结束需要将此Executor移除等。但是Driver与Executor却位于不同机器中，该怎么实现呢？Driver上的BlockManagerMaster会持有BlockManagerMasterActor，所有Executor也会从ActorSystem中获取BlockManagerMasterActor的引用，所有Executor与Driver关于BlockManager的交互都依赖于它。

4.3.1 BlockManagerMasterActor

BlockManagerMasterActor只存在于Driver上。Executor从ActorSystem获取BlockManager-MasterActor的引用，然后给BlockManagerMasterActor发送消息，实现和Driver交互。BlockManagerMasterActor的实现见代码清单4-10。

代码清单4-10 BlockManagerMasterActor的实现

private val blockManagerInfo = new mutable.HashMap[BlockManagerId, BlockManagerInfo] private val blockManagerIdByExecutor = new mutable.HashMap[String, BlockManagerId] private val blockLocations = new JHashMap[BlockId, mutable.HashSet[BlockManagerId]] private val akkaTimeout = AkkaUtils.askTimeout(conf) val slaveTimeout = conf.getLong("spark.storage.blockManagerSlaveTimeoutMs", math.max(conf.getInt("spark.executor.heartbeatInterval", 10000) \* 3, 45000)) val checkTimeoutInterval = conf.getLong("spark.storage.blockManagerTimeoutIntervalMs", 60000) var timeoutCheckingTask: Cancellable = null override def preStart() { import context.dispatcher timeoutCheckingTask = context.system.scheduler.schedule(0.seconds, checkTimeoutInterval.milliseconds, self, ExpireDeadHosts) super.preStart() } override def receiveWithLogging = { case RegisterBlockManager(blockManagerId, maxMemSize, slaveActor) => register(blockManagerId, maxMemSize, slaveActor) sender ! true case UpdateBlockInfo( blockManagerId, blockId, storageLevel, deserializedSize, size, tachyonSize) => updateBlockInfo(blockManagerId, blockId, storageLevel, deserializedSize, size, tachyonSize) case GetLocations(blockId) => sender ! getLocations(blockId) case GetLocationsMultipleBlockIds(blockIds) => sender ! getLocationsMultipleBlockIds(blockIds) case GetActorSystemHostPortForExecutor(executorId) => sender ! getActorSystemHostPortForExecutor(executorId) case GetMemoryStatus => sender ! memoryStatus case GetStorageStatus => sender ! storageStatus case GetBlockStatus(blockId, askSlaves) => sender ! blockStatus(blockId, askSlaves) case GetMatchingBlockIds(filter, askSlaves) => sender ! getMatchingBlockIds(filter, askSlaves) case RemoveRdd(rddId) => sender ! removeRdd(rddId) case RemoveShuffle(shuffleId) => sender ! removeShuffle(shuffleId) case RemoveBroadcast(broadcastId, removeFromDriver) => sender ! removeBroadcast(broadcastId, removeFromDriver) case RemoveBlock(blockId) => removeBlockFromWorkers(blockId) sender ! true case RemoveExecutor(execId) => removeExecutor(execId) sender ! true case StopBlockManagerMaster => sender ! true if (timeoutCheckingTask != null) {timeoutCheckingTask.cancel()} context.stop(self) case ExpireDeadHosts => expireDeadHosts() case BlockManagerHeartbeat(blockManagerId) => sender ! heartbeatReceived(blockManagerId) case other => logWarning("Got unknown message: " + other)

上面代码展示了BlockManagerMasterActor维护的很多缓存数据结构：

·blockManagerInfo：缓存所有的BlockManagerId及其BlockManager的信息；

·blockManagerIdByExecutor：缓存executorId与其拥有的BlockManagerId之间的映射关系；

·blockLocations：缓存Block与BlockManagerId的映射关系。

在代码清单4-10中，receiveWithLogging作为匹配BlockManagerMasterActor接收到消息的偏函数；属性spark.storage.blockManagerSlaveTimeoutMs和spark.executor.heartbeatInterval共同决定Slave节点，即BlockManager的超时时间；属性spark.storage.blockManagerTimeout-IntervalMs指定检查BlockManager超时的时间间隔。

4.3.2 询问Driver并获取回复方法

在Executor的BlockManagerMaster中，所有与Driver上BlockManagerMaster的交互方法最终都调用了askDriverWithReply，可见它是一个最基础的方法，它的代码如下。

private def askDriverWithReply[T](message: Any): T = { AkkaUtils.askWithReply(message, driverActor, AKKA\_RETRY\_ATTEMPTS, AKKA\_RETRY\_INTERVAL\_MS, timeout) }

此外，tell方法作为askDriverWithReply的代理也经常被调用，代码如下。

private def tell(message: Any) { if (!askDriverWithReply[Boolean](message)) { throw new SparkException("BlockManagerMasterActor returned false, expected true.") } }

askDriverWithReply调用了AkkaUtils.askWithReply方法。askWithReply方法实际使用Actor-System向BlockManagerMasterActor发送任何消息。发送每条消息的最大尝试次数是3次，间隔为3000毫秒，请求超时时间是30秒，具体实现见附录G。BlockManagerMasterActor接收到消息后将由receiveWithLogging函数匹配，并处理具体的逻辑。

4.3.3 向BlockManagerMaster注册BlockManagerId

Executor或者Driver自身的BlockManager在初始化时，需要向Driver的BlockManager注册BlockManager信息，代码如下。

def registerBlockManager(blockManagerId: BlockManagerId, maxMemSize: Long, slaveActor: ActorRef) { logInfo("Trying to register BlockManager") tell(RegisterBlockManager(blockManagerId, maxMemSize, slaveActor)) logInfo("Registered BlockManager") }

从上面代码看到，消息内容包括BlockManagerId、最大内存、BlockManagerSlaveActor。消息体带有BlockManagerSlaveActor是为了便于接受BlockManagerMasterActor回复的消息。这些信息被封装为RegisterBlockManager，并调用刚刚在4.3.2节介绍的tell方法。根据之前的分析，RegisterBlockManager消息会被BlockManagerMasterActor匹配并执行register方法注册BlockManager，并在register方法执行结束后向发送者BlockManagerSlaveActor发送一个简单的消息true。注册BlockManager的实现见代码清单4-11。

代码清单4-11 注册BlockManager的实现

private def register(id: BlockManagerId, maxMemSize: Long, slaveActor: ActorRef) { val time = System.currentTimeMillis() if (!blockManagerInfo.contains(id)) { blockManagerIdByExecutor.get(id.executorId) match { case Some(oldId) => // A block manager of the same executor already exists, so remove it (assumed dead) logError("Got two different block manager registrations on same executor - " + s" will replace old one $oldId with new one $id") removeExecutor(id.executorId) case None => } logInfo("Registering block manager %s with %s RAM, %s".format( id.hostPort, Utils.bytesToString(maxMemSize), id)) blockManagerIdByExecutor(id.executorId) = id blockManagerInfo(id) = new BlockManagerInfo( id, System.currentTimeMillis(), maxMemSize, slaveActor) } listenerBus.post(SparkListenerBlockManagerAdded(time, id, maxMemSize)) }

register方法确保blockManagerInfo持有消息中的blockManagerId及对应信息，并且确保每个Executor最多只能有一个blockManagerId，旧的blockManagerId会被移除。最后向listenerBus中post（推送）一个SparkListenerBlockManagerAdded事件。

00135.jpeg

注意

此处以注册BlockManager为例，演示了askDriverWithReply和tell的使用。

### 4.4 磁盘块管理器DiskBlockManager

4.4.1 DiskBlockManager的构造过程

BlockManager初始化时会创建DiskBlockManager，DiskBlockManager的构造步骤如下：

1）调用createLocalDirs方法创建本地文件目录，然后创建二维数组subDirs，用来缓存一级目录localDirs及二级目录，其中二级目录的数量根据配置spark.diskStore.subDirectories获取，默认为64。以笔者本地为例，创建的目录为：C：\Users\{username}\AppData\Local\Temp\spark-016f279f-4060-4065-b0cb-c7fad1f616ae\spark-58cdc43a-a39d-4b49-b357-f5ce9cc5c051，其中spark-016f279f-4060-4065-b0cb-c7fad1f616ae是一级目录，spark-58cdc43a-a39d-4b49-b357-f5ce9cc5c051是二级目录，见代码清单4-12。

代码清单4-12 创建本地文件目录的creatLocalDirs方法

private[spark] val subDirsPerLocalDir = blockManager.conf.getInt("spark.diskStore.subDirectories", 64) private[spark] val localDirs: Array[File] = createLocalDirs(conf) if (localDirs.isEmpty) { logError("Failed to create any local dir.") System.exit(ExecutorExitCode.DISK\_STORE\_FAILED\_TO\_CREATE\_DIR) } private val subDirs = Array.fill(localDirs.length)(new Array[File](subDirsPerLocalDir)) private def createLocalDirs(conf: SparkConf): Array[File] = { Utils.getOrCreateLocalRootDirs(conf).flatMap { rootDir => try { val localDir = Utils.createDirectory(rootDir, "blockmgr") logInfo(s"Created local directory at $localDir") Some(localDir) } catch { case e: IOException => logError(s"Failed to create local dir in $rootDir. Ignoring this directory.", e) None } } }

00138.jpeg

注意

createLocalDirs方法具体创建目录的过程实际调用了Utils的getOrCreateLocalRootDirs和createDirectory方法。有关Utils的使用请参见附录A。

2）添加运行时环境结束时的钩子，用于在进程关闭时创建线程，通过调用Disk-BlockManager的stop方法，清除一些临时目录，见代码清单4-13。

代码清单4-13 addShutdownHook的实现

addShutdownHook() private def addShutdownHook() { Runtime.getRuntime.addShutdownHook(new Thread("delete Spark local dirs") { override def run(): Unit = Utils.logUncaughtExceptions { logDebug("Shutdown hook called") DiskBlockManager.this.stop() } }) } /\*\* Cleanup local dirs and stop shuffle sender. \*/ private[spark] def stop() { // Only perform cleanup if an external service is not serving our shuffle files. if (!blockManager.externalShuffleServiceEnabled || blockManager.blockManagerId.isDriver) { localDirs.foreach { localDir => if (localDir.isDirectory() && localDir.exists()) { try { if (!Utils.hasRootAsShutdownDeleteDir(localDir)) Utils.deleteRecursively(localDir) } catch { case e: Exception => logError(s"Exception while deleting local spark dir: $localDir", e) } } } } }

DiskBlockManager为什么要创建二级目录结构？这是因为二级目录用于对文件进行散列存储，散列存储可以使所有文件都随机存放，写入或删除文件更方便，存取速度快，节省空间。

4.4.2 获取磁盘文件方法getFile

很多代码中都使用DiskBlockManager的getFile方法，获取磁盘上的文件，通过对getFile的分析，能够掌握Spark磁盘散列文件存储的实现机制。getFile方法的实现见代码清单4-14，其处理步骤如下：

1）根据文件名计算哈希值。

2）根据哈希值与本地文件一级目录的总数求余数，记为dirId。

3）根据哈希值与本地文件一级目录的总数求商数，此商数与二级目录的数目再求余数，记为subDirId。

4）如果dirId/subDirId目录存在，则获取dirId/subDirId目录下的文件，否则新建dirId/subDirId目录。

代码清单4-14 getFile方法的实现

def getFile(blockId: BlockId): File = getFile(blockId.name) def getFile(filename: String): File = { val hash = Utils.nonNegativeHash(filename) val dirId = hash % localDirs.length val subDirId = (hash / localDirs.length) % subDirsPerLocalDir var subDir = subDirs(dirId)(subDirId) if (subDir == null) { subDir = subDirs(dirId).synchronized { val old = subDirs(dirId)(subDirId) if (old != null) { old } else { val newDir = new File(localDirs(dirId), "%02x".format(subDirId)) newDir.mkdir() subDirs(dirId)(subDirId) = newDir newDir } } } new File(subDir, filename) }

4.4.3 创建临时Block方法createTempShuffleBlock

当ShuffleMapTask运行结束需要把中间结果临时保存，此时就调用createTempShuffle-Block方法创建临时的Block，并返回TempShuffleBlockId与其文件的对偶，见代码清单4-15。TempShuffleBlockId的生成规则："temp\_shuffle\_"后加上UUID字符串。

代码清单4-15 createTempShuffleBlock方法的实现

def createTempShuffleBlock(): (TempShuffleBlockId, File) = { var blockId = new TempShuffleBlockId(UUID.randomUUID()) while (getFile(blockId).exists()) { blockId = new TempShuffleBlockId(UUID.randomUUID()) } (blockId, getFile(blockId)) }

### 4.5 磁盘存储DiskStore

当MemoryStore没有足够空间时，就会使用DiskStore将块存入磁盘。DiskStore继承自BlockStore，并实现了getBytes、putBytes、putArray、putIterator等方法。

4.5.1 NIO读取方法getBytes

getBytes方法通过DiskBlockManager的getFile方法获取文件。然后使用NIO将文件读取到ByteBuffer，见代码清单4-16。

代码清单4-16 getBytes的实现

private def getBytes(file: File, offset: Long, length: Long): Option[ByteBuffer] = { val channel = new RandomAccessFile(file, "r").getChannel try { // For small files, directly read rather than memory map if (length < minMemoryMapBytes) { val buf = ByteBuffer.allocate(length.toInt) channel.position(offset) while (buf.remaining() != 0) { if (channel.read(buf) == -1) { throw new IOException("Reached EOF before filling buffer\n" + s"offset=$offset\nfile=${file.getAbsolutePath}\nbuf.remaining= ${buf.remaining}") } } buf.flip() Some(buf) } else { Some(channel.map(MapMode.READ\_ONLY, offset, length)) } } finally { channel.close() } } override def getBytes(blockId: BlockId): Option[ByteBuffer] = { val file = diskManager.getFile(blockId.name) getBytes(file, 0, file.length) }

4.5.2 NIO写入方法putBytes

putBytes方法的作用是通过DiskBlockManager的getFile方法获取文件。然后使用NIO的Channel将ByteBuffer写入文件，见代码清单4-17。

代码清单4-17 putBytes的实现

override def putBytes(blockId: BlockId, \_bytes: ByteBuffer, level: StorageLevel): PutResult = { val bytes = \_bytes.duplicate() logDebug(s"Attempting to put block $blockId") val startTime = System.currentTimeMillis val file = diskManager.getFile(blockId) val channel = new FileOutputStream(file).getChannel while (bytes.remaining > 0) { channel.write(bytes) } channel.close() val finishTime = System.currentTimeMillis logDebug("Block %s stored as %s file on disk in %d ms".format( file.getName, Utils.bytesToString(bytes.limit), finishTime - startTime)) PutResult(bytes.limit(), Right(bytes.duplicate())) }

4.5.3 数组写入方法putArray

putArray内部实际调用了putIterator，代码如下。

override def putArray( blockId: BlockId, values: Array[Any], level: StorageLevel, returnValues: Boolean): PutResult = { putIterator(blockId, values.toIterator, level, returnValues) }

putIterator的实现见代码清单4-18，其处理步骤如下：

1）使用了DiskBlockManager的getFile获取blockId对应的Block文件，并封装为FileOutputStream。

2）调用BlockManager的dataSerializeStream方法，将FileOutputStream序列化并压缩。dataSerializeStream的实现见4.8.12节。

3）如果需要返回写入的数据（即returnValues等于true），则将写入的文件使用getBytes读取为ByteBuffer，与文件的长度一并封装到PutResult中并返回，否则只返回文件长度。

代码清单4-18 putIterator的实现

val startTime = System.currentTimeMillis val file = diskManager.getFile(blockId) val outputStream = new FileOutputStream(file) try { try { blockManager.dataSerializeStream(blockId, outputStream, values) } finally { outputStream.close() } } catch { case e: Throwable => if (file.exists()) { file.delete() } throw e } val length = file.length val timeTaken = System.currentTimeMillis - startTime if (returnValues) { val buffer = getBytes(blockId).get PutResult(length, Right(buffer)) } else { PutResult(length, null) }

### 4.6 内存存储MemoryStore

MemoryStore负责将没有序列化的Java对象数组或者序列化的ByteBuffer存储到内存中。我们先来看看MemoryStore的数据结构，见代码清单4-19。

代码清单4-19 MemoryStore的数据结构

private[spark] class MemoryStore(blockManager: BlockManager, maxMemory: Long) extends BlockStore(blockManager) { private val conf = blockManager.conf private val entries = new LinkedHashMap[BlockId, MemoryEntry](32, 0.75f, true) @volatile private var currentMemory = 0L // Ensure only one thread is putting, and if necessary, dropping blocks at any given time private val accountingLock = new Object private val unrollMemoryMap = mutable.HashMap[Long, Long]() private val maxUnrollMemory: Long = { val unrollFraction = conf.getDouble("spark.storage.unrollFraction", 0.2) (maxMemory \* unrollFraction).toLong } private val unrollMemoryThreshold: Long = conf.getLong("spark.storage.unrollMemoryThreshold", 1024 \* 1024) def freeMemory: Long = maxMemory - currentMemory

根据代码清单4-19，我们先来用一张图来说明MemoryStore的内存模型，见图4-3。

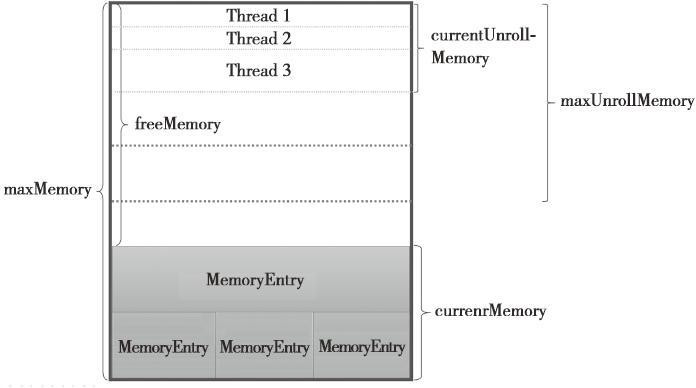


图4-3 MemoryStore的内存模型

从图4-3中看出，整个MemoryStore的存储分为两块：一块是被很多MemoryEntry占据的内存currentMemory，这些MemoryEntry实际是通过entries（即LinkedHashMap[BlockId，MemoryEntry）持有的；另一块是unrollMemoryMap通过占座方式占用的内存current-UnrollMemory。所谓占座，好比教室里空着的座位，有人在座位上放上书本，以防在需要坐的时候，却发现没有位置了。比起人的行为，unrollMemoryMap占座的出发点却是“高尚”的，这样可以防止在向内存真正写入数据时，内存不足发生溢出。每个线程实际占用的空间，其实是vector（即SizeTrackingVector）占用的大小，但是unrollMemoryMap的大小会稍大些。

这里把代码清单4-19中的一些概念，结合图4-3再解释下：

·maxUnrollMemory：当前Driver或者Executor最多展开的Block所占用的内存，可以修改属性spark.storage.unrollFraction改变大小；

·maxMemory：当前Driver或者Executor的最大内存；

·currentMemory：当前Driver或者Executor已经使用的内存；

·freeMemory：当前Driver或者Executor未使用的内存，freeMemory＝maxMemory－currentMemory；

·currentUnrollMemory：unrollMemoryMap中所有展开的Block的内存之和，即当前Driver或者Executor中所有线程展开的Block的内存之和；

·unrollMemoryMap：当前Driver或者Executor中所有线程展开的Block都存入此Map中，key为线程Id，value为线程展开的所有块的内存的大小总和。

MemoryStore继承自BlockStore，并实现了getBytes、putBytes、putArray、putIterator、getValues等方法。下面逐个介绍MemoryStore中实现的各个方法。

4.6.1 数据存储方法putBytes

如果Block可以被反序列化（即存储级别StorageLevel.Deserialized等于ture），那么先对Block序列化，然后调用putIterator；否则调用tryToPut方法，见代码清单4-20。

代码清单4-20 putBytes的实现

override def putBytes(blockId: BlockId, \_bytes: ByteBuffer, level: StorageLevel): PutResult = { // Work on a duplicate - since the original input might be used elsewhere. val bytes = \_bytes.duplicate() bytes.rewind() if (level.deserialized) { val values = blockManager.dataDeserialize(blockId, bytes) putIterator(blockId, values, level, returnValues = true) } else { val putAttempt = tryToPut(blockId, bytes, bytes.limit, deserialized = false) PutResult(bytes.limit(), Right(bytes.duplicate()), putAttempt.droppedBlocks) } }

4.6.2 Iterator写入方法putIterator详解

MemoryStore的putIterator方法的实现见代码清单4-21。调用unrollSafely将块在内存中安全展开，如果返回数据的类型匹配Left（arrayValues），则说明内存足够并调用putArray方法写入内存；如果返回数据的类型匹配Right（iteratorValues），则说明内存不足并写入硬盘或者放弃。diskStore.putIterator的实现见4.5.4节。unrollSafely方法将在4.6.3节说明。

代码清单4-21 MemoryStore.putIterator的实现

private[storage] def putIterator( blockId: BlockId, values: Iterator[Any], level: StorageLevel, returnValues: Boolean, allowPersistToDisk: Boolean): PutResult = { val droppedBlocks = new ArrayBuffer[(BlockId, BlockStatus)] val unrolledValues = unrollSafely(blockId, values, droppedBlocks) unrolledValues match { case Left(arrayValues) => // Values are fully unrolled in memory, so store them as an array val res = putArray(blockId, arrayValues, level, returnValues) droppedBlocks ++= res.droppedBlocks PutResult(res.size, res.data, droppedBlocks) case Right(iteratorValues) => // Not enough space to unroll this block; drop to disk if applicable if (level.useDisk && allowPersistToDisk) { logWarning(s"Persisting block $blockId to disk instead.") val res = blockManager.diskStore.putIterator(blockId, iteratorValues, level, returnValues) PutResult(res.size, res.data, droppedBlocks) } else { PutResult(0, Left(iteratorValues), droppedBlocks) } } }

为了防止写入内存的数据过大，导致内存溢出，Spark采用了一种优化方案：在正式写入内存之前，先用逻辑方式申请内存，如果申请成功，再写入内存，这个过程称为安全展开。表4-1列出了安全展开方法unrollSafely中一些变量及算法的定义。

表4-1 unrollSafely中一些变量及算法的定义



有了上述概念，下面我们列出展开内存的步骤：

1）申请memoryThreshold的初始大小为initialMemoryThreshold。

2）如果Iterator[Any]中有元素并且keepUnrolling为true，则向vector中添加Iterator-[Any]中的对象，elementsUnrolled自增1。如果Iterator[Any]中没有元素或者keepUnrolling不等于true，则跳转至第4）步。

3）如果elementsUnrolled%memoryCheckPeriod==0，则开始检查currentSize是否已经比memoryThreshold大？假如currentSize已经超过了memoryThreshold，则需要再申请内存，申请内存大小amountToRequest=currentSize\*memoryGrowthFactor–memoryThreshold。如果申请失败，但是maxUnrollMemory>currentUnrollMemory，则要求释放当前Driver或者Executor的其他内存，具体释放过程在4.6.4节讲解。释放内存必然伴随着其他Block被移入硬盘或者彻底清除，这些块的状态会在释放后返回。此时再次申请内存，memoryThreshold增加的大小为amountToRequest。返回第2）步。

4）根据是否将Block完整地放入内存，以数组或者迭代器形式返回vector的数据。

5）最后在finally语句块里，还会计算本次展开块实际占用的空间amountToRelease，并更新unrollMemoryMap中当前线程占用的内存大小，并减去amountToRelease。如果unrollMemoryMap中当前线程占用的内存大小小于等于0，则从unrollMemoryMap中完全清除此线程的数据。

对展开Block的算法就描述到这里，读者可以结合图4-3进行分析。unrollSafely的实现见代码清单4-22。

代码清单4-22 unrollSafely的实现

def unrollSafely( blockId: BlockId, values: Iterator[Any], droppedBlocks: ArrayBuffer[(BlockId, BlockStatus)]) : Either[Array[Any], Iterator[Any]] = { // Number of elements unrolled so far var elementsUnrolled = 0 // Whether there is still enough memory for us to continue unrolling this block var keepUnrolling = true // Initial per-thread memory to request for unrolling blocks (bytes). Exposed for testing. val initialMemoryThreshold = unrollMemoryThreshold // How often to check whether we need to request more memory val memoryCheckPeriod = 16 // Memory currently reserved by this thread for this particular unrolling operation var memoryThreshold = initialMemoryThreshold // Memory to request as a multiple of current vector size val memoryGrowthFactor = 1.5 // Previous unroll memory held by this thread, for releasing later (only at the very end) val previousMemoryReserved = currentUnrollMemoryForThisThread // Underlying vector for unrolling the block var vector = new SizeTrackingVector[Any] // Request enough memory to begin unrolling keepUnrolling = reserveUnrollMemoryForThisThread(initialMemoryThreshold) // Unroll this block safely, checking whether we have exceeded our threshold periodically try { while (values.hasNext && keepUnrolling) { vector += values.next() if (elementsUnrolled % memoryCheckPeriod == 0) { // If our vector's size has exceeded the threshold, request more memory val currentSize = vector.estimateSize() if (currentSize >= memoryThreshold) { val amountToRequest = (currentSize \* memoryGrowthFactor - memoryThreshold).toLong // Hold the accounting lock, in case another thread concurrently puts a block that // takes up the unrolling space we just ensured here accountingLock.synchronized { if (!reserveUnrollMemoryForThisThread(amountToRequest)) { // If the first request is not granted, try again after ensuring free space // If there is still not enough space, give up and drop the partition val spaceToEnsure = maxUnrollMemory - currentUnrollMemory if (spaceToEnsure > 0) { val result = ensureFreeSpace(blockId, spaceToEnsure) droppedBlocks ++= result.droppedBlocks } keepUnrolling = reserveUnrollMemoryForThisThread(amountToRequest) } } // New threshold is currentSize \* memoryGrowthFactor memoryThreshold += amountToRequest } } elementsUnrolled += 1 } if (keepUnrolling) { // We successfully unrolled the entirety of this block Left(vector.toArray) } else { // We ran out of space while unrolling the values for this block logUnrollFailureMessage(blockId, vector.estimateSize()) Right(vector.iterator ++ values) } } finally { // If we return an array, the values returned do not depend on the underlying vector and // we can immediately free up space for other threads. Otherwise, if we return an iterator, // we release the memory claimed by this thread later on when the task finishes. if (keepUnrolling) { val amountToRelease = currentUnrollMemoryForThisThread - previousMemoryReserved releaseUnrollMemoryForThisThread(amountToRelease) } } }

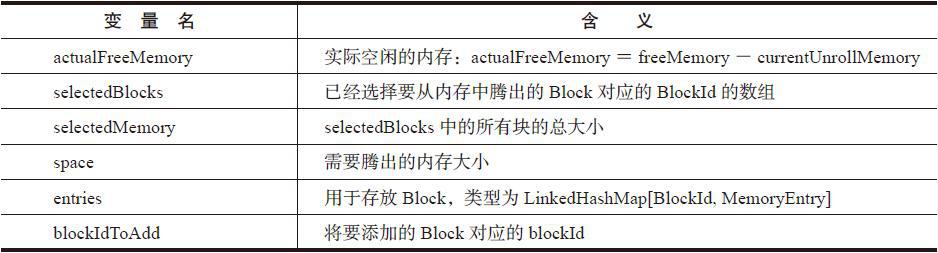
unrollSafely多次用到reserveUnrollMemoryForThisThread，以便给当前线程申请逻辑内存，它的实现如下。

def reserveUnrollMemoryForThisThread(memory: Long): Boolean = { accountingLock.synchronized { val granted = freeMemory > currentUnrollMemory + memory if (granted) { val threadId = Thread.currentThread().getId unrollMemoryMap(threadId) = unrollMemoryMap.getOrElse(threadId, 0L) + memory } granted } }

4.6.4 确认空闲内存方法ensureFreeSpace

ensureFreeSpace方法用于确认是否有足够内存，如果不足，会释放被MemoryEntry占用的内存。为了叙述方便，通过表4-2说明一些变量。

表4-2 ensureFreeSpace中的一些变量含义



有了上述概念，我们现在来看ensureFreeSpace的实现，见代码清单4-23。ensureFree-Space的处理步骤如下：

1）space不能超过maxMemory的限制，否则返回。

2）如果actualFreeMemory大于等于space，说明此时已经有充足的内存，不需要释放内存空间，直接返回。如果actualFreeMemory小于space，则说明空闲空间不足，需要释放一部分已经占用的内存。

3）当actualFreeMemory+selectedMemory ，则将blockId加入selectedBlocks，selectedMemory增加MemoryEntry的大小。

4）当actualFreeMemory＋selectedMemory≥space时，则说明可以腾出足够的内存空间。此时将selectedBlocks中所有的blockId对应于entries里的MemoryEntry取出，通过判断MemoryEntry是否可以反序列化，分别转换为Array[Any]或者ByteBuffer，调用blockManager的dropFromMemory方法，从内存中移除blockId及MemoryEntry，此方法最终返回移除的Block的状态。dropFromMemory的实现将在4.8.1节说明。

代码清单4-23 ensureFreeSpace方法的实现

private def ensureFreeSpace( blockIdToAdd: BlockId, space: Long): ResultWithDroppedBlocks = { logInfo(s"ensureFreeSpace($space) called with curMem=$currentMemory, maxMem=$maxMemory") val droppedBlocks = new ArrayBuffer[(BlockId, BlockStatus)] if (space > maxMemory) { logInfo(s"Will not store $blockIdToAdd as it is larger than our memory limit") return ResultWithDroppedBlocks(success = false, droppedBlocks) } // Take into account the amount of memory currently occupied by unrolling blocks val actualFreeMemory = freeMemory - currentUnrollMemory if (actualFreeMemory < space) { val rddToAdd = getRddId(blockIdToAdd) val selectedBlocks = new ArrayBuffer[BlockId] var selectedMemory = 0L entries.synchronized { val iterator = entries.entrySet().iterator() while (actualFreeMemory + selectedMemory < space && iterator.hasNext) { val pair = iterator.next() val blockId = pair.getKey if (rddToAdd.isEmpty || rddToAdd != getRddId(blockId)) { selectedBlocks += blockId selectedMemory += pair.getValue.size } } } if (actualFreeMemory + selectedMemory >= space) { logInfo(s"${selectedBlocks.size} blocks selected for dropping") for (blockId <- selectedBlocks) { val entry = entries.synchronized { entries.get(blockId) } if (entry != null) { val data = if (entry.deserialized) { Left(entry.value.asInstanceOf[Array[Any]]) } else { Right(entry.value.asInstanceOf[ByteBuffer].duplicate()) } val droppedBlockStatus = blockManager.dropFromMemory(blockId, data) droppedBlockStatus.foreach { status => droppedBlocks += ((blockId, status)) } } } return ResultWithDroppedBlocks(success = true, droppedBlocks) } else { logInfo(s"Will not store $blockIdToAdd as it would require dropping another block " + "from the same RDD") return ResultWithDroppedBlocks(success = false, droppedBlocks) } } ResultWithDroppedBlocks(success = true, droppedBlocks) }

4.6.5 内存写入方法putArray

内存写入方法putArray首先对对象大小进行估算，然后写入内存。如果unrollSafely返回的数据匹配Left（arrayValues），根据前面的分析知道，整个Block是可以一次性放入内存的。此时调用putArray方法，见代码清单4-24。

代码清单4-24 putArray的实现

override def putArray( blockId: BlockId, values: Array[Any], level: StorageLevel, returnValues: Boolean): PutResult = { if (level.deserialized) { val sizeEstimate = SizeEstimator.estimate(values.asInstanceOf[AnyRef]) val putAttempt = tryToPut(blockId, values, sizeEstimate, deserialized = true) PutResult(sizeEstimate, Left(values.iterator), putAttempt.droppedBlocks) } else { val bytes = blockManager.dataSerialize(blockId, values.iterator) val putAttempt = tryToPut(blockId, bytes, bytes.limit, deserialized = false) PutResult(bytes.limit(), Right(bytes.duplicate()), putAttempt.droppedBlocks) } }

SizeEstimator.estimate用来估算对象的大小，遍历对象及其属性。估算对象大小见代码清单4-25。

代码清单4-25 估算对象大小的方法estimate

def estimate(obj: AnyRef): Long = estimate(obj, new IdentityHashMap[AnyRef, AnyRef]) private def estimate(obj: AnyRef, visited: IdentityHashMap[AnyRef, AnyRef]): Long = { val state = new SearchState(visited) state.enqueue(obj) while (!state.isFinished) { visitSingleObject(state.dequeue(), state) } state.size } private def visitSingleObject(obj: AnyRef, state: SearchState) { val cls = obj.getClass if (cls.isArray) { visitArray(obj, cls, state) } else if (obj.isInstanceOf[ClassLoader] || obj.isInstanceOf[Class[\_]]) { } else { val classInfo = getClassInfo(cls) state.size += classInfo.shellSize for (field <- classInfo.pointerFields) { state.enqueue(field.get(obj)) } } }

4.6.6 尝试写入内存方法tryToPut

根据4.6.1节的内容我们知道，当Block不支持序列化时，会调用tryToPut方法。在介绍MemoryStore的putArray方法时，也最终使用此方法。

tryToPut的实现见代码清单4-26。可以看到tryToPut也调用了ensureFreeSpace方法，记得在调用tryToPut之前，已经在unrollSafely方法中调用过ensureFreeSpace了，这难道是重复调用了同一份代码吗？不，因为在展开阶段，即便内存充足，当真正写入数据时依然可能内存不足，所以需要再次确认空闲内存是否充足。

如果内存充足或者迁移其他内存Block后有足够内存，则会创建MemoryEntry对象，并将此对象与其blockId放入entries中，currentMemory会上浮估算的大小size。如果此时内存不足，还要把此blockId对应的MemoryEntry对象迁移到磁盘或者清除。

代码清单4-26 tryToPut的实现

private def tryToPut( blockId: BlockId, value: Any, size: Long, deserialized: Boolean): ResultWithDroppedBlocks = { var putSuccess = false val droppedBlocks = new ArrayBuffer[(BlockId, BlockStatus)] accountingLock.synchronized { val freeSpaceResult = ensureFreeSpace(blockId, size) val enoughFreeSpace = freeSpaceResult.success droppedBlocks ++= freeSpaceResult.droppedBlocks if (enoughFreeSpace) { val entry = new MemoryEntry(value, size, deserialized) entries.synchronized { entries.put(blockId, entry) currentMemory += size } val valuesOrBytes = if (deserialized) "values" else "bytes" logInfo("Block %s stored as %s in memory (estimated size %s, free %s)".format( blockId, valuesOrBytes, Utils.bytesToString(size), Utils.bytesToString(freeMemory))) putSuccess = true } else { // Tell the block manager that we couldn't put it in memory so that it can drop it to // disk if the block allows disk storage. val data = if (deserialized) { Left(value.asInstanceOf[Array[Any]]) } else { Right(value.asInstanceOf[ByteBuffer].duplicate()) } val droppedBlockStatus = blockManager.dropFromMemory(blockId, data) droppedBlockStatus.foreach { status => droppedBlocks += ((blockId, status)) } } } ResultWithDroppedBlocks(putSuccess, droppedBlocks) }

4.6.7 获取内存数据方法getBytes

getBytes方法用于从entries中获取MemoryEntry，见代码清单4-27。

代码清单4-27 getBytes的实现

override def getBytes(blockId: BlockId): Option[ByteBuffer] = { val entry = entries.synchronized { entries.get(blockId) } if (entry == null) { None } else if (entry.deserialized) { Some(blockManager.dataSerialize(blockId, entry.value.asInstanceOf [Array[Any]].iterator)) } else { Some(entry.value.asInstanceOf[ByteBuffer].duplicate()) // Doesn't actually copy the data } }

从代码清单4-27看到，如果MemoryEntry支持反序列化，则将MemoryEntry的value反序列化后反回，否则对MemoryEntny的value复制后返回。

4.6.8 获取数据方法getValues

getValues也用于从内存中获取数据，即从entries中获取MemoryEntry，并将blockId和value返回，见代码清单4-28。

代码清单4-28 getValues的实现

override def getValues(blockId: BlockId): Option[Iterator[Any]] = { val entry = entries.synchronized { entries.get(blockId) } if (entry == null) { None } else if (entry.deserialized) { Some(entry.value.asInstanceOf[Array[Any]].iterator) } else { val buffer = entry.value.asInstanceOf[ByteBuffer].duplicate() // Doesn't actually copy data Some(blockManager.dataDeserialize(blockId, buffer)) } }

### 4.7 Tachyon存储TachyonStore

为什么要使用Tachyon？原因如下：

·Spark的ShuffleMapTask和ResultTask被划分到不同Stage，ShuffleMapTask执行完毕将中间结果输出到本地磁盘文件系统（如HDFS），然后下一Stage中的ResultTask通过shuffleClient下载ShuffleMapTask的输出到本地磁盘文件系统，这种基于磁盘的读写效率较低；

·Spark的计算引擎与存储体系都位于Executor的同一进程中，当计算执行崩溃出错后，存储体系缓存的数据也会全部丢失；

·不同的Spark任务可能会访问同样的数据，例如两个任务都要访问HDFS中的某些Block，每个任务都要自己去磁盘加载数据到内存中。这导致数据被重复加载到内存，数据对象太多会导致Java GC时间过长等问题。

4.7.1 Tachyon简介

Tachyon也诞生于UCBerkeley的AMP实验室，是以内存为中心的高容错的分布式文件系统，能够为集群框架（比如Spark、Map-Reduce等）提供可靠的内存级的文件共享服务。从软件栈的层次来看，Tachyon是位于现有大数据计算框架和大数据存储系统之间的独立的一层，如图4-4所示。它利用底层文件系统作为备份，对于上层应用来说，Tachyon就是一个分布式文件系统。

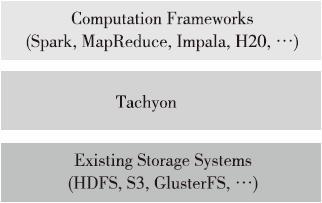


图4-4 Tachyon与计算框架、存储系统的层次关系

Tachyon属于伯克利大数据分析软件栈（Berkeley Data Analytics Stack）中的存储层软件，如图4-5所示。

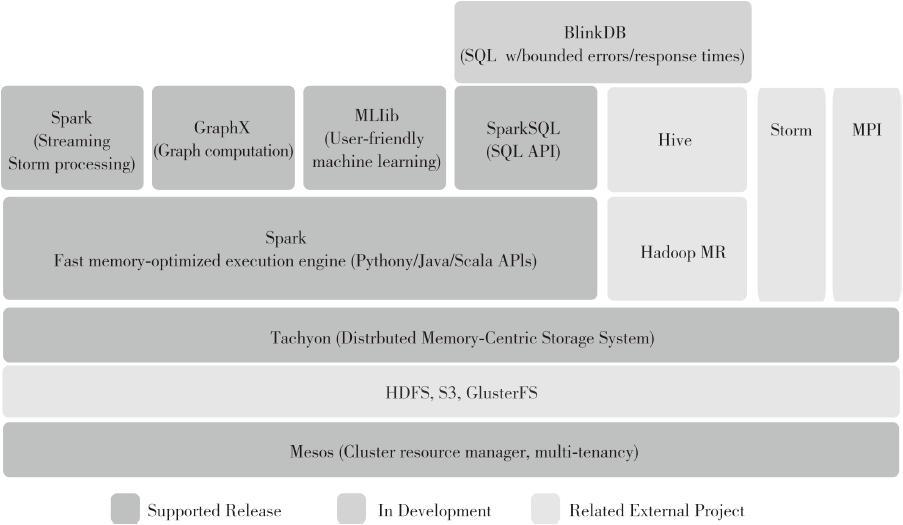


图4-5 伯克利大数据分析软件栈

Tachyon的整体架构如图4-6所示。

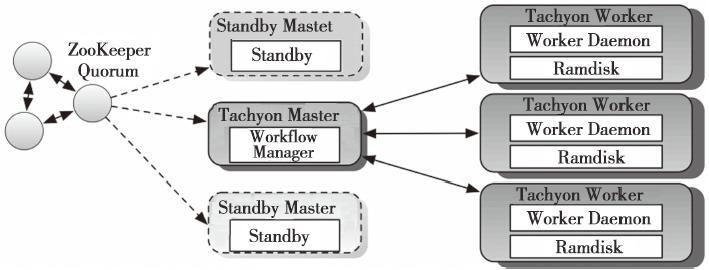


图4-6 Tachyon的整体架构

Tachyon也采用了Master-Worker的架构。Tachyon Master支持ZooKeeper进行容错，用于管理全部文件的元数据信息，同时也监控各个Tachyon Worker的状态。每个Tachyon Worker启动一个守护进程，管理本地的Ramdisk，Ramdisk中存储了具体的文件数据。Ramdisk实际是Tachyon集群的内存部分。

Tachyon的容错处理是怎样的？Tachyon内存中的数据如果不落地岂不是依然有丢失问题？Tachyon采用与Spark的RDD相类似（都是基于RDD不可变性以及粗粒度操作）的方法，它利用lineage信息（lineage-based recovery）和异步记录的checkpoint来恢复数据，所以可以放心大胆地使用Tachyon管理的内存。

00167.jpeg

注意

更多Tachyon的信息，请访问http://tachyon-project.org/。

4.7.2 TachyonStore的使用

Spark源码自带例子SparkTachyonHdfsLR演示了如何使用Tachyon，此例子在计算过程中将RDD持久化到Tachyon，代码如下。

val inputPath = args(0) val sparkConf = new SparkConf().setAppName("SparkTachyonHdfsLR") val conf = new Configuration() val sc = new SparkContext(sparkConf, InputFormatInfo.computePreferredLocations( Seq(new InputFormatInfo(conf, classOf[org.apache.hadoop.mapred.TextInputFormat], inputPath)) )) val lines = sc.textFile(inputPath) val points = lines.map(parsePoint \_).persist(StorageLevel.OFF\_HEAP) val ITERATIONS = args(1).toInt // Initialize w to a random value var w = DenseVector.fill(D){2 \* rand.nextDouble - 1} println("Initial w: " + w) for (i <- 1 to ITERATIONS) { println("On iteration " + i) val gradient = points.map { p => p.x \* (1 / (1 + exp(-p.y \* (w.dot(p.x)))) - 1) \* p.y }.reduce(\_ + \_) w -= gradient } println("Final w: " + w) sc.stop()

4.7.3 写入Tachyon内存的方法putIntoTachyonStore

TachyonStore也实现了BlockStore的getSize、putBytes、putArray、putIterator、getValues、getBytes等方法。其中putBytes、putArray、putIterator实际都调用了putIntoTachyonStore，PutIntoTachyonStore用于将数据写入Tachyon的分布式内存中。putIntoTachyonStore的实现见代码清单4-29。

代码清单4-29 putIntoTachyonStore的实现

private def putIntoTachyonStore( blockId: BlockId, bytes: ByteBuffer, returnValues: Boolean): PutResult = { val byteBuffer = bytes.duplicate() byteBuffer.rewind() logDebug(s"Attempting to put block $blockId into Tachyon") val startTime = System.currentTimeMillis val file = tachyonManager.getFile(blockId) val os = file.getOutStream(WriteType.TRY\_CACHE) os.write(byteBuffer.array()) os.close() val finishTime = System.currentTimeMillis logDebug("Block %s stored as %s file in Tachyon in %d ms".format( blockId, Utils.bytesToString(byteBuffer.limit), finishTime - startTime)) if (returnValues) { PutResult(bytes.limit(), Right(bytes.duplicate())) } else { PutResult(bytes.limit(), null) } }

4.7.4 获取序列化数据方法getBytes

getValues方法实际也调用了getBytes，getBytes的实现见代码清单4-30。其中Tachyon-BlockManager的getFile的代码实现与DiskBlockManager非常类似，有兴趣的读者可以自己找资料阅读。

代码清单4-30 getBytes的实现

override def getBytes(blockId: BlockId): Option[ByteBuffer] = { val file = tachyonManager.getFile(blockId) if (file == null || file.getLocationHosts.size == 0) { return None } val is = file.getInStream(ReadType.CACHE) assert (is != null) try { val size = file.length val bs = new Array[Byte](size.asInstanceOf[Int]) ByteStreams.readFully(is, bs) Some(ByteBuffer.wrap(bs)) } catch { case ioe: IOException => logWarning(s"Failed to fetch the block $blockId from Tachyon", ioe) None } finally { is.close() } }

### 4.8 块管理器BlockManager

已经介绍了BlockManager中的主要组件，现在来看看BlockManager自身的实现。

4.8.1 移出内存方法dropFromMemory

当内存不足时，可能需要腾出部分内存空间。dropFromMemory实现了这个功能，见代码清单4-31，它的处理步骤如下。

1）从blockInfo：TimeStampedHashMap[BlockId，BlockInfo]中检查是否存在要迁移的blockId。如果存在，从BlockInfo中获取Block的StorageLevel。

2）如果StorageLevel允许存入硬盘，并且DiskStore中不存在此文件，那么调用DiskStore的putArray或者putBytes方法，将此Block存入硬盘。

3）从MemoryStore中清除此BlockId对应的Block。

4）使用getCurrentBlockStatus方法获取Block的最新状态。如果此Block的tellMaster属性为true，则调用reportBlockStatus方法给BlockManagerMasterActor报告状态。report-BlockStatus在4.8.2节描述。

5）从blockInfo中清除此BlockId，并返回Block的状态。

代码清单4-31 移出内存的实现

def dropFromMemory( blockId: BlockId, data: Either[Array[Any], ByteBuffer]): Option[BlockStatus] = { logInfo(s"Dropping block $blockId from memory") val info = blockInfo.get(blockId).orNull // If the block has not already been dropped if (info != null) { info.synchronized { var blockIsUpdated = false val level = info.level // Drop to disk, if storage level requires if (level.useDisk && !diskStore.contains(blockId)) { logInfo(s"Writing block $blockId to disk") data match { case Left(elements) => diskStore.putArray(blockId, elements, level, returnValues = false) case Right(bytes) => diskStore.putBytes(blockId, bytes, level) } blockIsUpdated = true } // Actually drop from memory store val droppedMemorySize = if (memoryStore.contains(blockId)) memoryStore.getSize(blockId) else 0L val blockIsRemoved = memoryStore.remove(blockId) if (blockIsRemoved) { blockIsUpdated = true } else { logWarning(s"Block $blockId could not be dropped from memory as it does not exist") } val status = getCurrentBlockStatus(blockId, info) if (info.tellMaster) { reportBlockStatus(blockId, info, status, droppedMemorySize) } if (!level.useDisk) { // The block is completely gone from this node; forget it so we can put() it again later. blockInfo.remove(blockId) } if (blockIsUpdated) { return Some(status) } } } None }

4.8.2 状态报告方法reportBlockStatus

reportBlockStatus用于向BlockManagerMasterActor报告Block的状态并且重新注册BlockManager。reportBlockStatus的实现见代码清单4-32。它的处理步骤如下。

1）调用tryToReportBlockStatus方法，tryToReportBlockStatus调用了BlockManagerMaster的updateBlockInfo方法向BlockManagerMasterActor发送UpdateBlockInfo消息更新Block占用的内存大小、磁盘大小、存储级别等信息。

2）如果BlockManager还没有向BlockManagerMasterActor注册，则调用asyncReregister方法，asyncReregister调用了reregister，最后reregister实际调用了BlockManagerMaster的registerBlockManager方法和reportAllBlocks方法，reportAllBlocks方法实际也是调用了tryToReportBlockStatus。

代码清单4-32 状态报告的实现

private def reportBlockStatus(blockId: BlockId, info: BlockInfo, status: BlockStatus, droppedMemorySize: Long = 0L): Unit = { val needReregister = !tryToReportBlockStatus(blockId, info, status, droppedMemorySize) if (needReregister) { logInfo(s"Got told to re-register updating block $blockId") asyncReregister() } logDebug(s"Told master about block $blockId") } private def tryToReportBlockStatus(blockId: BlockId, info: BlockInfo, status: BlockStatus, droppedMemorySize: Long = 0L): Boolean = { if (info.tellMaster) { val storageLevel = status.storageLevel val inMemSize = Math.max(status.memSize, droppedMemorySize) val inTachyonSize = status.tachyonSize val onDiskSize = status.diskSize master.updateBlockInfo( blockManagerId, blockId, storageLevel, inMemSize, onDiskSize, inTachyonSize) } else { true } } def reregister(): Unit = { logInfo("BlockManager re-registering with master") master.registerBlockManager(blockManagerId, maxMemory, slaveActor) reportAllBlocks() } private def asyncReregister(): Unit = { asyncReregisterLock.synchronized { if (asyncReregisterTask == null) { asyncReregisterTask = Future[Unit] { reregister() asyncReregisterLock.synchronized { asyncReregisterTask = null } } } } }

RegisterBlockManager消息的处理已在4.3节介绍过，不再赘述。我们来看看Block-ManagerMaster的updateBlockInfo，代码如下。

val res = askDriverWithReply[Boolean]( UpdateBlockInfo(blockManagerId, blockId, storageLevel, memSize, diskSize, tachyonSize)) logInfo("Updated info of block " + blockId) res

从上面updateBlockInfo方法的实现中发现它也调用了我们熟悉的askDriverWithReply方法，只不过消息是UpdateBlockInfo。BlockManagerMasterActor接收后会调用updateBlockInfo方法更新blockManagerInfo及blockLocations等信息，比较简单，读者可自行分析。如果不熟悉askDriverWithReply，可以回顾4.3.2节。

4.8.3 单对象块写入方法putSingle

putSingle方法用于将由一个对象构成的Block写入存储系统。putSingle经过层层调用，实际调用了doPut方法，见代码清单4-33。doPut方法将在4.8.5节说明。

代码清单4-33 putSingle的实现

def putSingle( blockId: BlockId, value: Any, level: StorageLevel, tellMaster: Boolean = true): Seq[(BlockId, BlockStatus)] = { putIterator(blockId, Iterator(value), level, tellMaster) } def putIterator( blockId: BlockId, values: Iterator[Any], level: StorageLevel, tellMaster: Boolean = true, effectiveStorageLevel: Option[StorageLevel] = None): Seq[(BlockId, BlockStatus)] = { require(values != null, "Values is null") doPut(blockId, IteratorValues(values), level, tellMaster, effective-StorageLevel) }

4.8.4 序列化字节块写入方法putBytes

putBytes方法用于将序列化字节组成的Block写入存储系统，putBytes实际也调用了doPut方法，见代码清单4-34。

代码清单4-34 putBytes的实现

def putBytes( blockId: BlockId, bytes: ByteBuffer, level: StorageLevel, tellMaster: Boolean = true, effectiveStorageLevel: Option[StorageLevel] = None): Seq[(BlockId, BlockStatus)] = { require(bytes != null, "Bytes is null") doPut(blockId, ByteBufferValues(bytes), level, tellMaster, effective-StorageLevel) }

4.8.5 数据写入方法doPut

putSingle、putBytes等方法真正的数据写入实际由doPut实现，doPut的处理流程如图4-7所示。

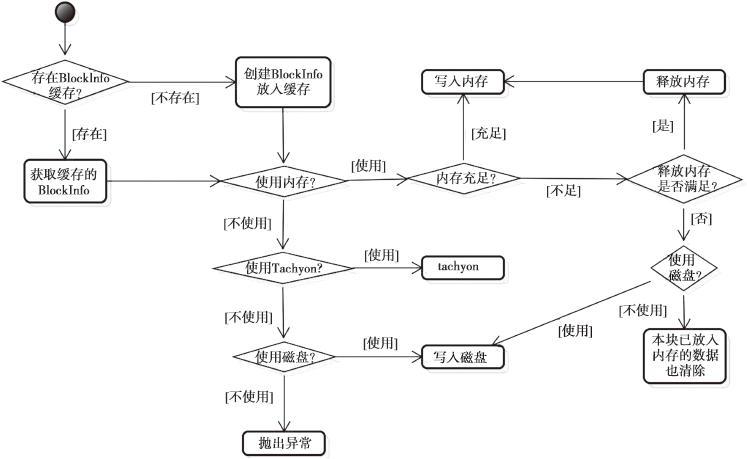


图4-7 doPut的处理流程

doPut的处理步骤如下。

1）获取putBlockInfo。如果blockInfo中已经缓存了BlockInfo，则使用缓存的BlockInfo，否则使用新建的BlockInfo。获取PutBlockInfo的实现，见代码清单4-35。

代码清单4-35 获取putBlockInfo的代码实现

val updatedBlocks = new ArrayBuffer[(BlockId, BlockStatus)] val putBlockInfo = { val tinfo = new BlockInfo(level, tellMaster) // Do atomically ! val oldBlockOpt = blockInfo.putIfAbsent(blockId, tinfo) if (oldBlockOpt.isDefined) { if (oldBlockOpt.get.waitForReady()) { logWarning(s"Block $blockId already exists on this machine; not re-adding it") return updatedBlocks } oldBlockOpt.get } else { tinfo } }

2）获取块最终使用的存储级别putLevel，根据putLevel判断块写入的BlockStore，从代码清单4-36可以看出，优先使用MemoryStore，其次是TachyonStore和DiskStore。依据data的实际包装类型，分别调用BlockStore不同的方法，如putIterator、putArray、putBytes等。

代码清单4-36 获取块最终使用的存储级别

// The level we actually use to put the block val putLevel = effectiveStorageLevel.getOrElse(level) val (returnValues, blockStore: BlockStore) = { if (putLevel.useMemory) { // Put it in memory first, even if it also has useDisk set to true; // We will drop it to disk later if the memory store can't hold it. (true, memoryStore) } else if (putLevel.useOffHeap) { // Use tachyon for off-heap storage (false, tachyonStore) } else if (putLevel.useDisk) { // Don't get back the bytes from put unless we replicate them (putLevel.replication > 1, diskStore) } else { assert(putLevel == StorageLevel.NONE) throw new BlockException( blockId, s"Attempted to put block $blockId without specifying storage level!") } } // Actually put the values val result = data match { case IteratorValues(iterator) => blockStore.putIterator(blockId, iterator, putLevel, returnValues) case ArrayValues(array) => blockStore.putArray(blockId, array, putLevel, returnValues) case ByteBufferValues(bytes) => bytes.rewind() blockStore.putBytes(blockId, bytes, putLevel) } size = result.size result.data match { case Left (newIterator) if putLevel.useMemory => valuesAfterPut = newIterator case Right (newBytes) => bytesAfterPut = newBytes case \_ => }

3）写入完毕后，将写入操作导致从内存drop掉的Block更新到updatedBlocks：Array-Buffer[（BlockId，BlockStatus）]中。使用getCurrentBlockStatus获取写入Block的状态。将putBlockInfo设置为允许其他线程读取，调用reportBlockStatus将当前Block的信息更新到BlockManagerMasterActor，最后将putBlockInfo添加到updatedBlocks中，见代码清单4-37。updatedBlocks中的Block的状态由于都发生了变化，所以都需要向BlockManagerMasterActor发送updateBlockInfo消息。

代码清单4-37 整理需要更新的Block

// Keep track of which blocks are dropped from memory if (putLevel.useMemory) { result.droppedBlocks.foreach { updatedBlocks += \_ } } val putBlockStatus = getCurrentBlockStatus(blockId, putBlockInfo) if (putBlockStatus.storageLevel != StorageLevel.NONE) { marked = true putBlockInfo.markReady(size) if (tellMaster) { reportBlockStatus(blockId, putBlockInfo, putBlockStatus) } updatedBlocks += ((blockId, putBlockStatus))

4）如果putLevel.replication大于1，即为了容错考虑，数据的备份数量大于1的时候，需要将Block的数据备份到其他节点上，见代码清单4-38。备份工作由replicate完成，请阅读4.8.6节。

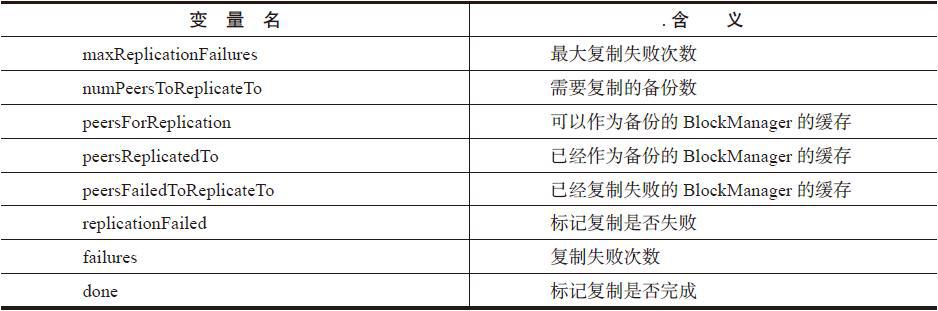
代码清单4-38 Block的备份

if (putLevel.replication > 1) { data match { case ByteBufferValues(bytes) => if (replicationFuture != null) { Await.ready(replicationFuture, Duration.Inf) } case \_ => val remoteStartTime = System.currentTimeMillis // Serialize the block if not already done if (bytesAfterPut == null) { if (valuesAfterPut == null) { throw new SparkException( "Underlying put returned neither an Iterator nor bytes! This shouldn't happen.") } bytesAfterPut = dataSerialize(blockId, valuesAfterPut) } replicate(blockId, bytesAfterPut, putLevel) logDebug("Put block %s remotely took %s" .format(blockId, Utils.getUsedTimeMs(remoteStartTime))) } } BlockManager.dispose(bytesAfterPut) updatedBlocks

4.8.6 数据块备份方法replicate

在介绍replicate方法之前，先对其中的一些定义进行解释，见表4-3。

表4-3 replicate中的定义



为了容灾，peersForReplication中缓存的BlockManager不应当是当前的BlockManager。获取其他所有BlockManager的方法是getPeers，见代码清单4-39。getPeers方法中的定义见表4-4。

代码清单4-39 getPeers的实现

private def getPeers(forceFetch: Boolean): Seq[BlockManagerId] = { peerFetchLock.synchronized { val cachedPeersTtl = conf.getInt("spark.storage.cachedPeersTtl", 60 \* 1000) // milliseconds val timeout = System.currentTimeMillis - lastPeerFetchTime > cachedPeersTtl if (cachedPeers == null || forceFetch || timeout) { cachedPeers = master.getPeers(blockManagerId).sortBy(\_.hashCode) lastPeerFetchTime = System.currentTimeMillis logDebug("Fetched peers from master: " + cachedPeers.mkString("[", ",", "]")) } cachedPeers } }

表4-4 getPeers方法中的定义



当cachedPeers为空或者forceFetch为true或者当前时间超时，则会调用BlockManager-Master的getPeers方法，从BlockManagerMasterActor获取最新的BlockManagerId。

BlockManagerMaster的getPeers方法也调用了我们熟悉的askDriverWithReply，代码如下。

def getPeers(blockManagerId: BlockManagerId): Seq[BlockManagerId] = { askDriverWithReply[Seq[BlockManagerId]](GetPeers(blockManagerId)) }

BlockManagerMasterActor匹配GetPeers消息将执行getPeers方法，getPeers从block-ManagerInfo中过滤掉Driver的BlockManager和当前的Executor的BlockManager，将其余的BlockManagerId都返回，见代码清单4-40。

代码清单4-40 BlockManagerMasterActor.getPeers的实现

private def getPeers(blockManagerId: BlockManagerId): Seq[BlockManagerId] = { val blockManagerIds = blockManagerInfo.keySet if (blockManagerIds.contains(blockManagerId)) { blockManagerIds.filterNot { \_.isDriver }.filterNot { \_ == blockManagerId }.toSeq } else { Seq.empty } }

replicate方法的实现见代码清单4-41。replicate有个内部函数getRandomPeer，用于随机获取BlockManagerId。由于random：Random（blockId.hashCode）使用blockId的哈希值，这样就保证在同一个节点上多次尝试复制同一个Block，保证它始终被复制到同一批节点上。特别注意的是，当复制失败并且再次尝试时，会强制从BlockManagerMasterActor获取所有最新的BlockManagerId。并且从peersForReplication中排除peersReplicatedTo和peersFailed-ToReplicateTo，即排除已经使用和已经复制失败的BlockManager的BlockManagerId。

代码清单4-41 replicate方法的实现

private def replicate(blockId: BlockId, data: ByteBuffer, level: StorageLevel): Unit = { // 常量、变量定义省略 val random = new Random(blockId.hashCode) def getRandomPeer(): Option[BlockManagerId] = { if (replicationFailed) { peersForReplication.clear() peersForReplication ++= getPeers(forceFetch = true) peersForReplication --= peersReplicatedTo peersForReplication --= peersFailedToReplicateTo } if (!peersForReplication.isEmpty) { Some(peersForReplication(random.nextInt(peersForReplication.size))) } else { None } }

现在我们正式讲解备份复制的过程，见代码清单4-42。通过图4-8能够帮助我们理解其处理步骤。

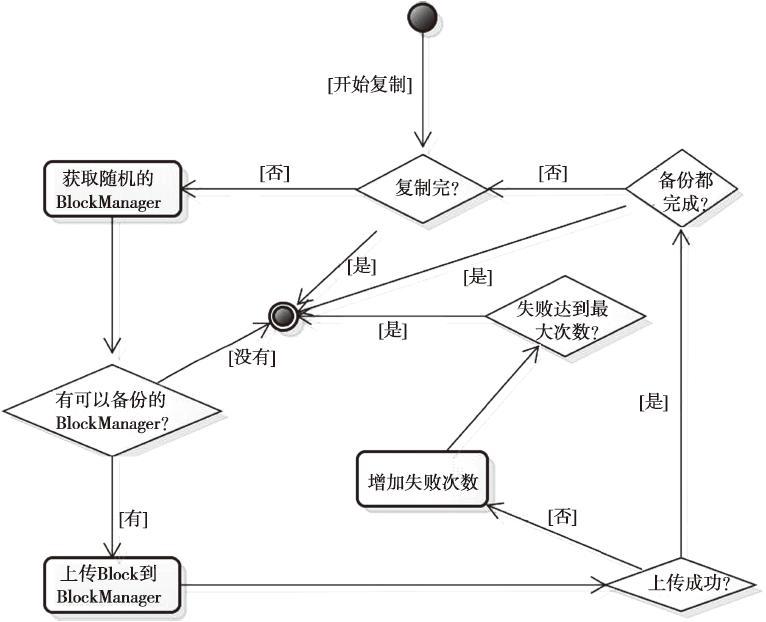


图4-8 Block备份复制的过程

代码清单4-42 备份复制的实现

while (!done) { getRandomPeer() match { case Some(peer) => try { val onePeerStartTime = System.currentTimeMillis data.rewind() logTrace(s"Trying to replicate $blockId of ${data.limit()} bytes to $peer") blockTransferService.uploadBlockSync( peer.host, peer.port, peer.executorId, blockId, new NioManaged-Buffer(data), tLevel) logTrace(s"Replicated $blockId of ${data.limit()} bytes to $peer in %s ms" .format(System.currentTimeMillis - onePeerStartTime)) peersReplicatedTo += peer peersForReplication -= peer replicationFailed = false if (peersReplicatedTo.size == numPeersToReplicateTo) { done = true // specified number of peers have been replicated to } } catch { case e: Exception => logWarning(s"Failed to replicate $blockId to $peer, failure #$failures", e) failures += 1 replicationFailed = true peersFailedToReplicateTo += peer if (failures > maxReplicationFailures) { // too many failures in replcating to peers done = true } } case None => // no peer left to replicate to done = true } }

从replicate的代码实现，总结复制的过程如下。

1）调用getRandomPeer随机获取BlockManager。

2）上传Block到BlockManager。

3）如果上传成功，则将此BlockManager添加到peersReplicatedTo，而从peersFor-Replication中移除，设置replicationFailed等于false，done等于true；如果上传过程出现异常，则将此BlockManager添加到peersFailedToReplicateTo，failures自增，设置replicationFailed等于true，done等于false。

如果上传失败，以上过程会迭代多次，直到失败次数failures超过最大失败次数maxRep-licationFailures。

异步上传方法uploadBlockSync实际是通过调用blockTransferService.uploadBlock来完成的，代码如下。NettyBlockTransferService的uploadBlock方法已在4.2.6节介绍过。

def uploadBlockSync(hostname: String, port: Int, execId: String, blockId: BlockId, blockData: ManagedBuffer, level: StorageLevel): Unit = { Await.result(uploadBlock(hostname, port, execId, blockId, blockData, level), Duration.Inf) }

4.8.7 创建DiskBlockObjectWriter的方法getDiskWriter

getDiskWriter用于创建DiskBlockObjectWriter，见代码清单4-43。属性spark.shuffle.sync决定写操作是同步还是异步。其中有关wrapForCompression方法的内容请阅读4.8.12节。

代码清单4-43 创建DiskBlockObjectWriter

def getDiskWriter( blockId: BlockId, file: File, serializer: Serializer, bufferSize: Int, writeMetrics: ShuffleWriteMetrics): BlockObjectWriter = { val compressStream: OutputStream => OutputStream = wrapForCompression(blockId, \_) val syncWrites = conf.getBoolean("spark.shuffle.sync", false) new DiskBlockObjectWriter(blockId, file, serializer, bufferSize, compressStream, syncWrites, writeMetrics) }

4.8.8 获取本地Block数据方法getBlockData

getBlockData用于从本地获取Block的数据，见代码清单4-44。

代码清单4-44 从本地获取Block

override def getBlockData(blockId: BlockId): ManagedBuffer = { if (blockId.isShuffle) { shuffleManager.shuffleBlockManager.getBlockData(blockId.asInstanceOf [ShuffleBlockId]) } else { val blockBytesOpt = doGetLocal(blockId, asBlockResult = false) .asInstanceOf[Option[ByteBuffer]] if (blockBytesOpt.isDefined) { val buffer = blockBytesOpt.get new NioManagedBuffer(buffer) } else { throw new BlockNotFoundException(blockId.toString) } } }

getBlockData的处理实现如下。

1）如果Block是ShuffleMapTask的输出，那么多个partition的中间结果都写入了同一个文件，怎样读取不同partition的中间结果？IndexShuffleBlockManager的getBlockData方法解决了这个问题，请阅读4.13节。

2）如果Block是ResultTask的输出，则使用doGetLocal来获取本地中间结果数据，请参阅4.8.9节。

4.8.9 获取本地shuffle数据方法doGetLocal

当reduce任务与map任务处于同一节点时，不需要远程拉取，只需调取doGetLocal方法从本地获取中间处理结果即可。doGetLocal的实现见代码清单4-45，其处理步骤如下：

1）如果Block允许使用内存，则调用MemoryStore的getValues或者getBytes方法获取。getValues和getBytes参阅4.6节。

2）如果Block允许使用Tachyon，则调用TachyonStore的getBytes方法获取，请参阅4.7.4节。

3）如果Block允许使用DiskStore，则调用DiskStore的getBytes方法获取，请参阅4.5.1节。

代码清单4-45 doGetLocal的实现

private def doGetLocal(blockId: BlockId, asBlockResult: Boolean): Option[Any] = { val info = blockInfo.get(blockId).orNull if (info != null) { info.synchronized { // 省略部分代码 // Look for the block in memory if (level.useMemory) { logDebug(s"Getting block $blockId from memory") val result = if (asBlockResult) { memoryStore.getValues(blockId).map(new BlockResult(\_, DataReadMethod.Memory, info.size)) } else { memoryStore.getBytes(blockId) } result match { case Some(values) => return result case None => logDebug(s"Block $blockId not found in memory") } } // Look for the block in Tachyon if (level.useOffHeap) { logDebug(s"Getting block $blockId from tachyon") if (tachyonStore.contains(blockId)) { tachyonStore.getBytes(blockId) match { case Some(bytes) => if (!asBlockResult) { return Some(bytes) } else { return Some(new BlockResult( dataDeserialize(blockId, bytes), DataReadMethod.Memory, info.size)) } case None => logDebug(s"Block $blockId not found in tachyon") } } } // Look for block on disk, potentially storing it back in memory if required if (level.useDisk) { logDebug(s"Getting block $blockId from disk") val bytes: ByteBuffer = diskStore.getBytes(blockId) match { case Some(b) => b case None => throw new BlockException( blockId, s"Block $blockId not found on disk, though it should be") } assert(0 == bytes.position()) // 次要代码此处省略

4.8.10 获取远程Block数据方法doGetRemote

doGetRemote用于从远端节点上获取Block数据，见代码清单4-46。其处理步骤如下：

1）向BlockManagerMasterActor发送GetLocations消息获取Block数据存储的Block-ManagerId。如果Block数据复制份数多于1个，则会返回多个BlockManagerId，对这些Block-ManagerId洗牌，避免总是从一个远程BlockManager获取Block数据。发送GetLocations消息使用了getLocations方法，代码如下。

def getLocations(blockId: BlockId): Seq[BlockManagerId] = { askDriverWithReply[Seq[BlockManagerId]](GetLocations(blockId)) }

2）根据返回的BlockManagerId信息，使用BlockTransferService远程同步获取Block数据。

代码清单4-46 获取远程Block数据

private def doGetRemote(blockId: BlockId, asBlockResult: Boolean): Option[Any] = { require(blockId != null, "BlockId is null") val locations = Random.shuffle(master.getLocations(blockId)) for (loc <- locations) { logDebug(s"Getting remote block $blockId from $loc") val data = blockTransferService.fetchBlockSync( loc.host, loc.port, loc.executorId, blockId.toString).nioByteBuffer() if (data != null) { if (asBlockResult) { return Some(new BlockResult( dataDeserialize(blockId, data), DataReadMethod.Network, data.limit())) } else { return Some(data) } } logDebug(s"The value of block $blockId is null") } logDebug(s"Block $blockId not found") None }

get方法用于通过BlockId获取Block。get方法在实现上首先从本地获取，如果没有则去远程获取，见代码清单4-47。

代码清单4-47 get方法的实现

def get(blockId: BlockId): Option[BlockResult] = { val local = getLocal(blockId) if (local.isDefined) { logInfo(s"Found block $blockId locally") return local } val remote = getRemote(blockId) if (remote.isDefined) { logInfo(s"Found block $blockId remotely") return remote } None }

getLocal方法实际调用了doGetLocal方法，代码如下。

def getLocal(blockId: BlockId): Option[BlockResult] = { logDebug(s"Getting local block $blockId") doGetLocal(blockId, asBlockResult = true).asInstanceOf[Option[BlockResult]] }

getRemote方法实际调用了doGetRemote方法，代码如下。

def getRemote(blockId: BlockId): Option[BlockResult] = { logDebug(s"Getting remote block $blockId") doGetRemote(blockId, asBlockResult = true).asInstanceOf[Option[BlockResult]] }

4.8.12 数据流序列化方法dataSerializeStream

如果写入存储体系的数据本身是序列化的，那么读取时应该对其反序列化。data-SerializeStream方法使用compressionCodec对文件输入流进行压缩和序列化处理，见代码清单4-48。compressionCodec的代码分析见4.11节。

代码清单4-48 dataSerializeStream的实现

def dataSerializeStream( blockId: BlockId, outputStream: OutputStream, values: Iterator[Any], serializer: Serializer = defaultSerializer): Unit = { val byteStream = new BufferedOutputStream(outputStream) val ser = serializer.newInstance() ser.serializeStream(wrapForCompression(blockId, byteStream)).writeAll(values).close() } def wrapForCompression(blockId: BlockId, s: OutputStream): OutputStream = { if (shouldCompress(blockId)) compressionCodec.compressedOutputStream(s) else s }

### 4.9 metadataCleaner和broadcastCleaner

为了有效利用磁盘空间和内存，metadataCleaner和broadcastCleaner分别用于清除blockInfo（TimeStampedHashMap[BlockId，BlockInfo]）中很久不用的非广播和广播Block信息。

00194.jpeg

注意

此处的metadataCleaner与3.3节介绍的metadataCleaner命名相同，作用却不一样，读者不要混淆。

3.3节中已经介绍过metadataCleaner，由此知道每个metadataCleaner的关键在于函数参数cleanupFunc：（Long）=>Unit。此处的metadataCleaner的函数参数是dropOldNonBroadcast-Blocks，broadcastCleaner的函数参数是dropOldBroadcastBlocks。

private def dropOldNonBroadcastBlocks(cleanupTime: Long): Unit = { logInfo(s"Dropping non broadcast blocks older than $cleanupTime") dropOldBlocks(cleanupTime, !\_.isBroadcast) } private def dropOldBroadcastBlocks(cleanupTime: Long): Unit = { logInfo(s"Dropping broadcast blocks older than $cleanupTime") dropOldBlocks(cleanupTime, \_.isBroadcast) }

这两个函数都调用了dropOldBlocks，dropOldBlocks的实现见代码清单4-49。遍历blockInfo，将很久不用的Block从MemoryStore、DiskStore、TachyonStore中清除。

代码清单4-49 删除很久不用的Block

private def dropOldBlocks(cleanupTime: Long, shouldDrop: (BlockId => Boolean)): Unit = { val iterator = blockInfo.getEntrySet.iterator while (iterator.hasNext) { val entry = iterator.next() val (id, info, time) = (entry.getKey, entry.getValue.value, entry.getValue.timestamp) if (time < cleanupTime && shouldDrop(id)) { info.synchronized { val level = info.level if (level.useMemory) { memoryStore.remove(id) } if (level.useDisk) { diskStore.remove(id) } if (level.useOffHeap) { tachyonStore.remove(id) } iterator.remove() logInfo(s"Dropped block $id") } val status = getCurrentBlockStatus(id, info) reportBlockStatus(id, info, status) } } }

### 4.10 缓存管理器CacheManager

CacheManager用于缓存RDD某个分区计算后的中间结果。读者可能误以为RDD都缓存在CacheManager的某个存储部分中，实际上CacheManager只是对BlockManager的代理，真正的缓存依然使用BlockManager。在任务迭代计算的过程中，当判断存储级别使用了缓存，就会调用CacheManager的getOrCompute方法。getOrCompute的实现见代码清单4-50。

代码清单4-50 getOrCompute的实现

def getOrCompute[T]( rdd: RDD[T], partition: Partition, context: TaskContext, storageLevel: StorageLevel): Iterator[T] = { val key = RDDBlockId(rdd.id, partition.index) logDebug(s"Looking for partition $key") blockManager.get(key) match { case Some(blockResult) => // Partition is already materialized, so just return its values context.taskMetrics.inputMetrics = Some(blockResult.inputMetrics) new InterruptibleIterator(context, blockResult.data.asInstanceOf[Iterator[T]]) case None => // Acquire a lock for loading this partition // If another thread already holds the lock, wait for it to finish return its results val storedValues = acquireLockForPartition[T](key) if (storedValues.isDefined) { return new InterruptibleIterator[T](context, storedValues.get) } // Otherwise, we have to load the partition ourselves try { logInfo(s"Partition $key not found, computing it") val computedValues = rdd.computeOrReadCheckpoint(partition, context) // If the task is running locally, do not persist the result if (context.isRunningLocally) { return computedValues } // Otherwise, cache the values and keep track of any updates in block statuses val updatedBlocks = new ArrayBuffer[(BlockId, BlockStatus)] val cachedValues = putInBlockManager(key, computedValues, storageLevel, updatedBlocks) val metrics = context.taskMetrics val lastUpdatedBlocks = metrics.updatedBlocks.getOrElse(Seq[(BlockId, BlockStatus)]()) metrics.updatedBlocks = Some(lastUpdatedBlocks ++ updatedBlocks.toSeq) new InterruptibleIterator(context, cachedValues) } finally { loading.synchronized { loading.remove(key) loading.notifyAll() } } } }

从getOrCompute的实现分析其处理逻辑如下：

1）从存储体系获取Block；

2）如果确实获取到了Block，那么将它封装为InterruptibleIterator并返回。如果还没有缓存Block，则重新计算或者从CheckPoint中获取数据，并调用putInBlockManager方法将数据写入缓存后封装为InterruptibleIterator并返回。其中RDD的computeOrReadCheckpoint方法将在6.1节说明。

putInBlockManager的实现见代码清单4-51。

代码清单4-51 putInBlockManager的实现

private def putInBlockManager[T]( key: BlockId, values: Iterator[T], level: StorageLevel, updatedBlocks: ArrayBuffer[(BlockId, BlockStatus)], effectiveStorageLevel: Option[StorageLevel] = None): Iterator[T] = { val putLevel = effectiveStorageLevel.getOrElse(level) if (!putLevel.useMemory) { updatedBlocks ++= blockManager.putIterator(key, values, level, tellMaster = true, effectiveStorageLevel) blockManager.get(key) match { case Some(v) => v.data.asInstanceOf[Iterator[T]] case None => logInfo(s"Failure to store $key") throw new BlockException(key, s"Block manager failed to return cached value for $key!") } } else { blockManager.memoryStore.unrollSafely(key, values, updatedBlocks) match { case Left(arr) => // We have successfully unrolled the entire partition, so cache it in memory updatedBlocks ++= blockManager.putArray(key, arr, level, tellMaster = true, effectiveStorageLevel) arr.iterator.asInstanceOf[Iterator[T]] case Right(it) => // There is not enough space to cache this partition in memory val returnValues = it.asInstanceOf[Iterator[T]] if (putLevel.useDisk) { logWarning(s"Persisting partition $key to disk instead.") val diskOnlyLevel = StorageLevel(useDisk = true, useMemory = false, useOffHeap = false, deserialized = false, putLevel.replication) putInBlockManager[T](key, returnValues, level, updatedBlocks, Some(diskOnlyLevel)) } else { returnValues } } } }

从putInBlockManager的实现，总结它的处理步骤如下。

1）获取实际的存储级别。

2）如果存储级别不允许使用内存，那么直接调用BlockManager的putIterator方法。在doPut方法的处理中，由于存储级别不允许使用内存，所以数据实际被直接写入了磁盘或者Tachyon。

3）如果存储级别允许使用内存，那么首先尝试展开。如果展开成功，说明有足够内存可以存储数据，因此将数据存入内存；如果展开失败，则将数据存入磁盘。

### 4.11 压缩算法

为了节省磁盘存储空间，有些情况下需要对Block进行压缩。根据配置属性spark.io.compression.codec来确定要使用的压缩算法（默认为snappy，此压缩算法在牺牲少量压缩比例的条件下，却极大地提高了压缩速度），并生成SnappyCompressionCodec的实例，见代码清单4-52。

代码清单4-52 CompressionCodec的实现

private[spark] object CompressionCodec { private val shortCompressionCodecNames = Map( "lz4" -> classOf[LZ4CompressionCodec].getName, "lzf" -> classOf[LZFCompressionCodec].getName, "snappy" -> classOf[SnappyCompressionCodec].getName) def createCodec(conf: SparkConf): CompressionCodec = { createCodec(conf, conf.get("spark.io.compression.codec", DEFAULT\_COMPRESSION\_CODEC)) } def createCodec(conf: SparkConf, codecName: String): CompressionCodec = { val codecClass = shortCompressionCodecNames.getOrElse(codecName.toLowerCase, codecName) val ctor = Class.forName(codecClass, true, Utils.getContextOrSparkClassLoader) .getConstructor(classOf[SparkConf]) ctor.newInstance(conf).asInstanceOf[CompressionCodec] } val DEFAULT\_COMPRESSION\_CODEC = "snappy" val ALL\_COMPRESSION\_CODECS = shortCompressionCodecNames.values.toSeq }

### 4.12 磁盘写入实现DiskBlockObjectWriter

DiskBlockObjectWriter被用于输出Spark任务的中间计算结果。DiskBlockObjectWriter的fileSegment方法用于创建文件分片FileSegment（FileSegment记录分片的起始、结束偏移量），代码如下。

override def fileSegment(): FileSegment = { new FileSegment(file, initialPosition, finalPosition - initialPosition) }

下面我们逐个讲解DiskBlockObjectWriter的其他方法，包括open、write、close、commit-AndClose。

1.打开一个文件输出流

DiskBlockObjectWriter的open方法，利用NIO、压缩、缓存、序列化方式打开一个文件输出流，见代码清单4-53。

代码清单4-53 DiskBlockObjectWriter的open方法

override def open(): BlockObjectWriter = { fos = new FileOutputStream(file, true) ts = new TimeTrackingOutputStream(fos) channel = fos.getChannel() bs = compressStream(new BufferedOutputStream(ts, bufferSize)) objOut = serializer.newInstance().serializeStream(bs) initialized = true this }

2.写入文件

DiskBlockObjectWriter的write方法用于将数据写入文件，并更新测量信息，见代码清单4-54。

代码清单4-54 DiskBlockObjectWriter的write方法

override def write(value: Any) { if (!initialized) { open() } objOut.writeObject(value) if (writesSinceMetricsUpdate == 32) { writesSinceMetricsUpdate = 0 updateBytesWritten() } else { writesSinceMetricsUpdate += 1 } } private def updateBytesWritten() { val pos = channel.position() writeMetrics.shuffleBytesWritten += (pos - reportedPosition) reportedPosition = pos }

3.关闭文件输出流

DiskBlockObjectWriter的close方法用于关闭文件输出流，并更新测量信息，见代码清单4-55。

代码清单4-55 关闭文件输出流

override def close() { if (initialized) { if (syncWrites) { // Force outstanding writes to disk and track how long it takes objOut.flush() def sync = fos.getFD.sync() callWithTiming(sync) } objOut.close() channel = null bs = null fos = null ts = null objOut = null initialized = false } } private def callWithTiming(f: => Unit) = { val start = System.nanoTime() f writeMetrics.shuffleWriteTime += (System.nanoTime() - start) }

4.缓存数据提交

DiskBlockObjectWriter的commitAndClose方法将缓存数据写入磁盘并关闭缓存，然后更新测量数据，见代码清单4-56。

代码清单4-56 commitAndClose提交缓存数据

override def commitAndClose(): Unit = { if (initialized) { objOut.flush() bs.flush() close() } finalPosition = file.length() // In certain compression codecs, more bytes are written after close() is called writeMetrics.shuffleBytesWritten += (finalPosition - reportedPosition) }

### 4.13 块索引shuffle管理器IndexShuffleBlockManager

IndexShuffleBlockManager通常用于获取Block索引文件，并根据索引文件读取Block文件的数据。

1.获取shuffle文件方法getBlockData

有时候我们不知道Block的BlockId，所以无法使用BlockManager的get方法获取Block。如果知道ShuffleBlockId，我们依然可以通过ShuffleBlockId记录的shuffleId和mapId获取Block。ShuffleBlockId的格式如下。

case class ShuffleBlockId(shuffleId: Int, mapId: Int, reduceId: Int) extends BlockId { def name = "shuffle\_" + shuffleId + "\_" + mapId + "\_" + reduceId }

按照此格式生成的ShuffleBlockId能够关联shuffleId、mapId和reduceId的信息，例如，shuffle\_0\_0\_0、shuffle\_0\_1\_0等。

getBlockData方法根据shuffleId和mapId（即partitionId）读取索引文件，从索引文件中获得partition计算中间结果写入文件的偏移量和中间结果的大小，根据此偏移量和大小读取文件中partition的中间计算结果，见代码清单4-57。

代码清单4-57 getBlockData的实现

override def getBlockData(blockId: ShuffleBlockId): ManagedBuffer = { val indexFile = getIndexFile(blockId.shuffleId, blockId.mapId) val in = new DataInputStream(new FileInputStream(indexFile)) try { ByteStreams.skipFully(in, blockId.reduceId \* 8) val offset = in.readLong() val nextOffset = in.readLong() new FileSegmentManagedBuffer( transportConf, getDataFile(blockId.shuffleId, blockId.mapId), offset, nextOffset - offset) } finally { in.close() } }

2.获取shuffle数据文件方法getDataFile

getDataFile的实现代码如下。

def getDataFile(shuffleId: Int, mapId: Int): File = { blockManager.diskBlockManager.getFile(ShuffleDataBlockId(shuffleId, mapId, 0)) }

getDataFile的实质是调用diskBlockManager的getFile方法，请参阅4.4.2节。

3.索引文件偏移量记录方法writeIndexFile

writeIndexFile方法用于在Block索引文件中记录各个partition的偏移量信息，便于下游Stage的任务读取，见代码清单4-58。

代码清单4-58 索引文件写入方法writeIndexFile

def writeIndexFile(shuffleId: Int, mapId: Int, lengths: Array[Long]) = { val indexFile = getIndexFile(shuffleId, mapId) val out = new DataOutputStream(new BufferedOutputStream(new FileOutput-Stream(indexFile))) try { var offset = 0L out.writeLong(offset) for (length <- lengths) { offset += length out.writeLong(offset) } } finally { out.close() } }

### 4.14 shuffle内存管理器ShuffleMemoryManager

ShuffleMemoryManager用于为执行shuffle操作的线程分配内存池。每种磁盘溢出集合（如ExternalAppendOnlyMap和ExternalSorter）都能从这个内存池获得内存。当溢出集合的数据已经输出到存储系统，获得的内存会释放。当线程执行的任务结束，整个内存池都会被Executor释放。ShuffleMemoryManager会保证每个线程都能合理地共享内存，而不会使得一些线程获得了很大的内存，导致其他线程经常不得不将溢出的数据写入磁盘。

尝试获得内存方法tryToAcquire

此方法用于当前线程尝试获得numBytes大小的内存，并返回实际获得的内存大小，见代码清单4-59。

代码清单4-59 尝试获得内存的实现

def tryToAcquire(numBytes: Long): Long = synchronized { val threadId = Thread.currentThread().getId assert(numBytes > 0, "invalid number of bytes requested: " + numBytes) if (!threadMemory.contains(threadId)) { threadMemory(threadId) = 0L notifyAll() // Will later cause waiting threads to wake up and check numThreads again } while (true) { val numActiveThreads = threadMemory.keys.size val curMem = threadMemory(threadId) val freeMemory = maxMemory - threadMemory.values.sum val maxToGrant = math.min(numBytes, math.max(0, (maxMemory / numActiveThreads) - curMem)) if (curMem < maxMemory / (2 \* numActiveThreads)) { if (freeMemory >= math.min(maxToGrant, maxMemory / (2 \* numActive-Threads) - curMem)) { val toGrant = math.min(maxToGrant, freeMemory) threadMemory(threadId) += toGrant return toGrant } else { logInfo(s"Thread $threadId waiting for at least 1/2N of shuffle memory pool to be free") wait() } } else { // Only give it as much memory as is free, which might be none if it reached 1 / numThreads val toGrant = math.min(maxToGrant, freeMemory) threadMemory(threadId) += toGrant return toGrant } } 0L // Never reached }

根据ShuffleMemoryManager的实现，它的处理逻辑：假设当前有N个线程，必须保证每个线程在溢出之前至少获得

00198.jpeg

的内存，并且每个线程最多获得

00202.jpeg

的内存。由于N是动态变化的变量，所以要持续对这些线程进行跟踪，以便无论何时在这些线程发生变化时重新按照

00208.jpeg

和

00211.jpeg

计算。

### 4.15 小结

本章一开始介绍了BlockStore的接口定义，目前虽然只有MemoryStore、DiskStore和TachyonStore三种实现，但是随着技术的发展，当有更优秀的存储中间件出现时，随时可以实现BlockStore，完成集成（Tachyon就是例子）。DiskStore对磁盘文件按照散列存储节省空间的同时提高了文件访问的效率。MemoryStore基于内存做了大量优化，其构建的内存模型值得任何大数据存储引擎借鉴。TachyonStore解决了Spark中共享磁盘文件系统性能差、计算引擎出错导致存储体系的数据丢失、内存中大量重复数据导致GC时间长等问题。

此外，FileSegment与Block索引文件共同解决了分区数据读写同一文件的问题；shuffle服务基于Netty实现的Block上传下载服务；shuffle任务通过（

00214.jpeg

，

00221.jpeg

）的区间管理获得内存的大小保证每个线程都能合理地共享内存并减少磁盘I/O操作；Executor的BlockManager与Driver的BlockManager上的BlockManagerMasterActor在ActorSystem中通信，保证Driver获取实时的Bl