

第6章

Chapter 6

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

我们生活的地球上，存在着种类繁多的生物。这些生物的生存离不开阳光、空气和水。中国传统文化认为，天空风云多变，变幻无穷，非常强健，是君子向往的；无论是花鸟鱼虫，还是山川河海，都由大地承载，亦是君子追求的。古人常说：“天高任鸟飞，海阔凭鱼跃。”但是鸟飞得久了也得回巢，鱼跃得再高还是得回到水中。世间万物，无论你是什么，依然需要大地的承载。大地是安静的，也是非常包容的。

Spark 的存储体系在整个 Spark 架构中与大地在生态圈中的作用非常类似。Spark 的存储体系从 Spark 集群的横向角度看，贯穿了集群中的每个实例。从单个节点来看，Spark 的存储体系隶属于 SparkEnv。由于存储体系的内容非常丰富，很有必要开辟单独的一章内容进行详细分析。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。

早在 5.5 节介绍 BroadcastManager 的时候，我们就知道对广播对象进行广播离不开存储体系的支持。除此之外，无论是 SparkContext 的初始化，还是任务的提交与执行，始终都离不开存储体系。Spark 为了避免 Hadoop 频繁读写磁盘造成磁盘 I/O 成为性能瓶颈，优先会将配置信息、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。

本章主要讲解的内容如下。

- 存储体系的基本概念。
- 存储体系架构。
- BlockManager 中的各个组件。

6.1 存储体系概述

本书在 5.7 节曾介绍过存储体系的创建，那时只为帮助读者了解 SparkEnv，现在是时候对 Spark 的存储体系进行详细的分析了。简单来讲，Spark 存储体系是各个 Driver 和 Executor 实例中的 BlockManager 所组成的。但是从一个整体出发，把各个节点的 BlockManager 看成存储体系的一部分，那么存储体系还有更多衍生的内容，比如块传输服务、map 任务输出跟踪器、Shuffle 管理器等。

6.1.1 存储体系架构

在正式介绍存储体系的内容之前，如果能对存储体系从整体上有个宏观的认识，无疑有利于读者对后续内容的理解。图 6-1 能够从整体上表示存储体系架构。

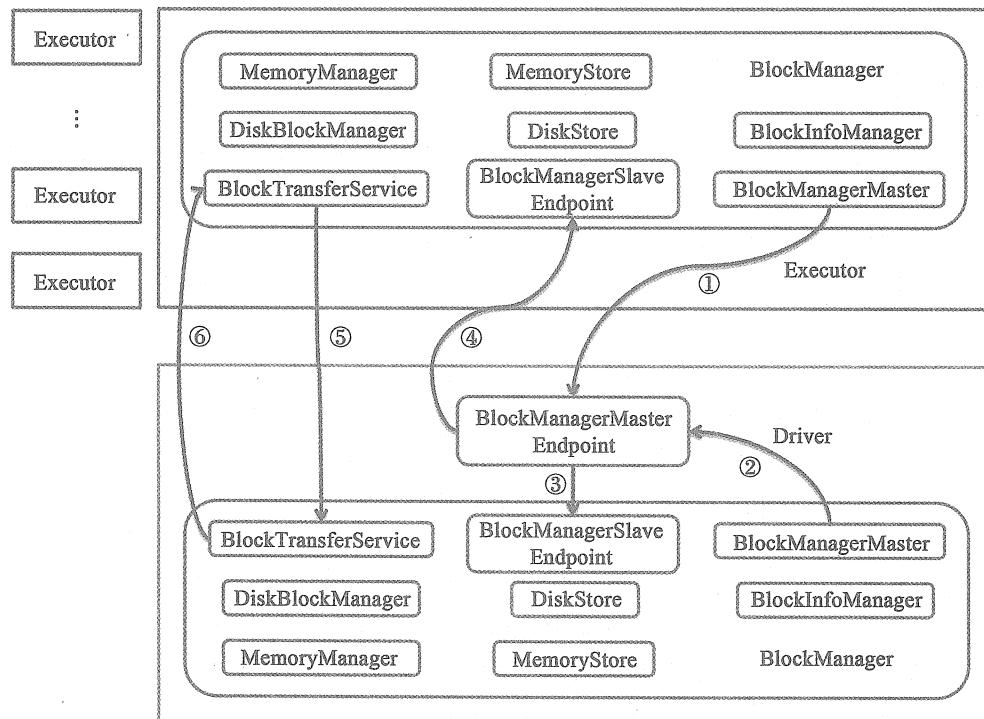


图 6-1 存储体系架构

从图 6-1 我们可以看到，BlockManager 依托于很多组件的服务，这些组件包括以下几项。

1) BlockManagerMaster：代理 BlockManager 与 Driver 上的 BlockManagerMasterEndpoint 通信。图 6-1 中的记号①表示 Executor 节点上的 BlockManager 通过 BlockManagerMaster 与 BlockManagerMasterEndpoint 进行通信，记号②表示 Driver 节点上的 BlockManager 通过 BlockManagerMaster 与 BlockManagerMasterEndpoint 进行通信。这些通信的内容有很多，例如，注册 BlockManager、更新 Block 信息、获取 Block 的位置（即 Block 所在的 BlockManager）、删除 Executor 等。BlockManagerMaster 之所以能够和 BlockManagerMasterEndpoint 通信，是因为它持有了 BlockManagerMasterEndpoint 的 RpcEndpointRef。

2) BlockManagerMasterEndpoint：由 Driver 上的 SparkEnv 负责创建和注册到 Driver 的 RpcEnv 中。BlockManagerMasterEndpoint 只存在于 Driver 的 SparkEnv 中，Driver 或 Executor 上的 BlockManagerMaster 的 driverEndpoint 属性将持有 BlockManagerMaster-Endpoint 的 RpcEndpointRef。BlockManagerMasterEndpoint 主要对各个节点上的 BlockManager、BlockManager 与 Executor 的映射关系及 Block 位置信息（即 Block 所在的 BlockManager）等进行管理。

3) BlockManagerSlaveEndpoint：每个 Executor 或 Driver 的 SparkEnv 中都有属于自己的 BlockManagerSlaveEndpoint，分别由各自的 SparkEnv 负责创建和注册到各自的 RpcEnv 中。Driver 或 Executor 都存在各自的 BlockManagerSlaveEndpoint，并由各自 BlockManager 的 slaveEndpoint 属性持有各自 BlockManagerSlaveEndpoint 的 RpcEndpointRef。BlockManager-SlaveEndpoint 将接收 BlockManagerMasterEndpoint 下发的命令。图 6-1 中的记号③表示 BlockManagerMasterEndpoint 向 Driver 节点上的 BlockManagerSlaveEndpoint 下发命令，记号④表示 BlockManagerMasterEndpoint 向 Executor 节点上的 BlockManagerSlaveEndpoint 下发命令。这些下发命令有很多，例如，删除 Block、获取 Block 状态、获取匹配的 BlockId 等。

4) SerializerManager：序列化管理器。SerializerManager 的内容已在 5.4 节详细分析过，此处不再赘述。

5) MemoryManager：内存管理器。负责对单个节点上内存的分配与回收。

6) MapOutputTracker：map 任务输出跟踪器。其实现与工作原理已在 5.6 节详细介绍过。

7) ShuffleManager：Shuffle 管理器。由于 Shuffle 与计算的关系更为紧密，所以留在第 8 章详细介绍。

8) BlockTransferService：块传输服务。此组件也与 Shuffle 相关联，主要用于不同阶段的任务之间的 Block 数据的传输与读写。例如，map 任务所在节点的 BlockTransferService 给 Shuffle 对应的 reduce 任务提供下载 map 中间输出结果的服务。

9) shuffleClient：Shuffle 的客户端。与 BlockTransferService 配合使用。图 6-1 中的记号⑤表示 Executor 上的 shuffleClient 通过 Driver 上的 BlockTransferService 提供的服务上传和下载 Block，记号⑥表示 Driver 上的 shuffleClient 通过 Executor 上的 BlockTransferService 提供的服务上传和下载 Block。此外，不同 Executor 节点上的 BlockTransferService 和 shuffleClient 之间也可以互相上传、下载 Block。

10) SecurityManager：安全管理器。具体实现已在 5.2 节介绍。

11) DiskBlockManager：磁盘块管理器。对磁盘上的文件及目录的读写操作进行管理。

12) BlockInfoManager：块信息管理器。负责对 Block 的元数据及锁资源进行管理。

13) MemoryStore：内存存储。依赖于 MemoryManager，负责对 Block 的内存存储。

14) DiskStore：磁盘存储。依赖于 DiskBlockManager，负责对 Block 的磁盘存储。

小贴士：在 Spark 1.x.x 版本中还提供了 TachyonStore，TachyonStore 负责向 Tachyon 中存储数据。Tachyon 也诞生于 UC Berkeley 的 AMP 实验室，是以内存为中心的高容错的分布式文件系统，能够为内存计算框架（比如 Spark、MapReduce 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。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。自 Spark 2.0.0 版本开始，TachyonStore 已经被移除。Tachyon 发展到 1.0.0 版本时改名为 Alluxio。更多 Alluxio 的信息，请访问 <http://www.alluxio.org>。

6.1.2 基本概念

存储体系中有一些概念，需要详细说明。正所谓“磨刀不误砍柴工”，有了对这些概念的理解，后面的内容学习起来会更加容易。

1. BlockManager 的唯一标识 BlockManagerId

根据之前的了解，我们知道在 Driver 或者 Executor 中有任务执行的环境 SparkEnv。每个 SparkEnv 中都有 BlockManager，这些 BlockManager 位于不同的节点和实例上。BlockManager 之间需要通过 RpcEnv、shuffleClient 及 BlockTransferService 相互通信，所以大家需要互相认识，正如每个人都有身份证号一样，每个 BlockManager 都有其在 Spark 集群内的唯一标识。BlockManagerId 就是 BlockManager 的身份证。

Spark 通过 BlockManagerId 中的 host、port、executorId 等信息来区分 BlockManager。BlockManagerId 中的属性包括以下几项。

- ❑ host_：主机域名或 IP。
- ❑ port_：此端口实际使用了 BlockManager 中的 BlockTransferService 对外服务的端口。
- ❑ executorId_：当前 BlockManager 所在的实例的 ID。如果实例是 Driver，那么 ID 为 driver，否则由 Master 负责给各个 Executor 分配，ID 格式为 app- 日期格式字符串 - 数字。

- topologyInfo_：拓扑信息。

理解了 BlockManagerId 中的属性，下面来看看它提供的方法。

- executorId：返回 executorId_ 的值。
- hostPort：返回 host:port 格式的字符串。
- host：返回 host_ 的值。
- port：返回 port_ 的值。
- topologyInfo：返回 topologyInfo_ 的值。
- isDriver：当前 BlockManager 所在的实例是否是 Driver。此方法实际根据 executorId_ 的值是否是 driver 来判断。
- writeExternal：将 BlockManagerId 的所有信息序列化后写到外部二进制流中。 writeExternal 的实现如代码清单 6-1 所示。

代码清单6-1 writeExternal的实现

```
override def writeExternal(out: ObjectOutput): Unit = Utils.tryOrIOException {
    out.writeUTF(executorId_)
    out.writeUTF(host_)
    out.writeInt(port_)
    out.writeBoolean(topologyInfo_.isDefined)
    // we only write topologyInfo if we have it
    topologyInfo_.foreach(out.writeUTF(_))
}
```

- readExternal：从外部二进制流中读取 BlockManagerId 的所有信息。readExternal 的实现如代码清单 6-2 所示。

代码清单6-2 readExternal的实现

```
override def readExternal(in: ObjectInput): Unit = Utils.tryOrIOException {
    executorId_ = in.readUTF()
    host_ = in.readUTF()
    port_ = in.readInt()
    val isTopologyInfoAvailable = in.readBoolean()
    topologyInfo_ = if (isTopologyInfoAvailable) Option(in.readUTF()) else None
}
```

2. 块的唯一标识 BlockId

了解操作系统存储的读者应该知道，文件系统的文件在存储到磁盘上时，都是以块为单位写入的。操作系统的块是以固定的大小读写的，例如，512 字节、1024 字节、2048 字节、4096 字节等。

在 Spark 的存储体系中，数据的读写也是以块为单位，只不过这个块并非操作系统的块，而是设计用于 Spark 存储体系的块。每个 Block 都有唯一的标识，Spark 把这个标识抽象为 BlockId。抽象类 BlockId 的定义如代码清单 6-3 所示。

代码清单6-3 BlockId的定义

```

@DeveloperApi
sealed abstract class BlockId {
    def name: String
    def asRDDId: Option[RDDBlockId] = if (isRDD) Some(asInstanceOf[RDDBlockId])
                                         else None
    def isRDD: Boolean = isInstanceOf[RDDBlockId]
    def isShuffle: Boolean = isInstanceOf[ShuffleBlockId]
    def isBroadcast: Boolean = isInstanceOf[BroadcastBlockId]
    override def toString: String = name
    override def hashCode: Int = name.hashCode
    override def equals(other: Any): Boolean = other match {
        case o: BlockId => getClass == o.getClass && name.equals(o.name)
        case _ => false
    }
}

```

根据代码清单 6-3，BlockId 中定义了以下方法。

- name: Block 全局唯一的标识名。
- isRDD: 当前 BlockId 是否是 RDDBlockId。
- asRDDId: 将当前 BlockId 转换为 RDDBlockId。如果当前 BlockId 是 RDDBlockId，则转换为 RDDBlockId，否则返回 None。
- isShuffle: 当前 BlockId 是否是 ShuffleBlockId。
- isBroadcast: 当前 BlockId 是否是 BroadcastBlockId。

BlockId 有很多子类，例如，RDDBlockId、ShuffleBlockId、BroadcastBlockId 等。BroadcastBlockId 曾在 5.5 节介绍广播管理器 BroadcastManager 时介绍过，BlockId 的子类都是用相似的方式实现的。

3. 存储级别 StorageLevel

前文提到，Spark 的存储体系包括磁盘存储与内存存储。Spark 将内存又分为堆外内存和堆内存。有些数据块本身支持序列化及反序列化，有些数据块还支持备份与复制。Spark 存储体系将以上这些数据块的不同特性抽象为存储级别（StorageLevel）。

分析 StorageLevel，也是从其成员属性开始。StorageLevel 中的成员属性如下。

- _useDisk: 能否写入磁盘。当 Block 的 StorageLevel 中的 _useDisk 为 true 时，存储体系将允许 Block 写入磁盘。
- _useMemory: 能否写入堆内存。当 Block 的 StorageLevel 中的 _useMemory 为 true 时，存储体系将允许 Block 写入堆内存。
- _useOffHeap: 能否写入堆外内存。当 Block 的 StorageLevel 中的 _useOffHeap 为 true 时，存储体系将允许 Block 写入堆外内存。
- _deserialized: 是否需要对 Block 反序列化。当 Block 本身经过了序列化后，Block 的 StorageLevel 中的 _deserialized 将被设置为 true，即可以对 Block 进行反序列化。

- ❑ `_replication`: Block 的复制份数。Block 的 StorageLevel 中的 `_replication` 默认等于 1, 可以在构造 Block 的 StorageLevel 时明确指定 `_replication` 的数量。当 `_replication` 大于 1 时, Block 除了在本地的存储体系中写入一份, 还会复制到其他不同节点的存储体系中写入, 达到复制备份的目的。

有了对 StorageLevel 中属性的了解, 现在来看看 StorageLevel 提供了哪些方法。

- ❑ `useDisk`: 能否写入磁盘。实际直接返回了 `_useDisk` 的值。
- ❑ `useMemory`: 能否写入堆内存。实际直接返回了 `_useMemory` 的值。
- ❑ `useOffHeap`: 能否写入堆外内存。实际直接返回了 `_useOffHeap` 的值。
- ❑ `deserialized`: 是否需要对 Block 反序列化。实际直接返回了 `_deserialized` 的值。
- ❑ `replication`: 复制份数。实际直接返回了 `_replication` 的值。
- ❑ `memoryMode`: 内存模式。如果 `useOffHeap` 为 true, 则返回枚举值 `MemoryMode.OFF_HEAP`, 否则返回枚举值 `MemoryMode.ON_HEAP`。
- ❑ `clone`: 对当前 StorageLevel 进行克隆, 并返回克隆的 StorageLevel。
- ❑ `isValid`: 当前的 StorageLevel 是否有效。判断的条件如下。

`(useMemory || useDisk) && (replication > 0)`

- ❑ `toInt`: 将当前 StorageLevel 转换为整型表示。`toInt` 的实现如代码清单 6-4 所示。

代码清单 6-4 `toInt` 的实现

```
def toInt: Int = {
    var ret = 0
    if (_useDisk) {
        ret |= 8
    }
    if (_useMemory) {
        ret |= 4
    }
    if (_useOffHeap) {
        ret |= 2
    }
    if (_deserialized) {
        ret |= 1
    }
    ret
}
```

根据代码清单 6-4, `toInt` 方法实际是把 StorageLevel 的 `_useDisk`、`_useMemory`、`_useOffHeap`、`_deserialized` 这四个属性设置到四位数字的各个状态位。例如, 1000 表示存储级别为允许写入磁盘; 1100 表示存储级别为允许写入磁盘和堆内存; 1111 表示存储级别为允许写入磁盘、堆内存及堆外内存, 并且需要反序列化。

- ❑ `writeExternal`: 将 StorageLevel 首先通过 `toInt` 方法将 `_useDisk`、`_useMemory`、`_useOffHeap`、`_deserialized` 四个属性设置到四位数的状态位, 然后与 `_replication` 一起被序列化写入外部二进制流。`writeExternal` 的实现如代码清单 6-5 所示。

代码清单6-5 writeExternal的实现

```
override def writeExternal(out: ObjectOutput): Unit = Utils.tryOrIOException {
    out.writeByte(toInt)
    out.writeByte(_replication)
}
```

- ❑ `readExternal`：从外部二进制流中读取 `StorageLevel` 的各个属性。`readExternal` 的实现如代码清单 6-6 所示。

代码清单6-6 readExternal的实现

```
override def readExternal(in: ObjectInput): Unit = Utils.tryOrIOException {
    val flags = in.readByte()
    _useDisk = (flags & 8) != 0
    _useMemory = (flags & 4) != 0
    _useOffHeap = (flags & 2) != 0
    _deserialized = (flags & 1) != 0
    _replication = in.readByte()
}
```

了解了 `StorageLevel` 原生类提供的这些方法，我们再来看看 `StorageLevel` 的伴生对象。由于 `StorageLevel` 的构造器是私有的，所以 `StorageLevel` 的伴生对象中已经预先定义了很多存储体系需要的 `StorageLevel`，如代码清单 6-7 所示。

代码清单6-7 内置的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)
```

代码清单 6-7 所示的代码调用 `StorageLevel` 的构造器创建了多种存储级别。`StorageLevel` 私有构造器的参数从左至右分别为 `_useDisk`、`_useMemory`、`_useOffHeap`、`_deserialized`、`_replication`。

4. 块信息 BlockInfo

`BlockInfo` 用于描述块的元数据信息，包括存储级别、`Block` 类型、大小、锁信息等。按照惯例，我们先看看 `BlockInfo` 的成员属性。

- ❑ `level`: `BlockInfo` 所描述的 `Block` 的存储级别，即 `StorageLevel`。
- ❑ `classTag`: `BlockInfo` 所描述的 `Block` 的类型。

- tellMaster: BlockInfo 所描述的 Block 是否需要告知 Master。
- _size: BlockInfo 所描述的 Block 的大小。
- _readerCount: BlockInfo 所描述的 Block 被锁定读取的次数。
- _writerTask: 任务尝试在对 Block 进行写操作前, 首先必须获得对应 BlockInfo 的写锁。_writerTask 用于保存任务尝试的 ID (每个任务在实际执行时, 会多次尝试, 每次尝试都会分配一个 ID)。

有了对 BlockInfo 的了解, 现在来看看 BlockInfo 提供的方法。

- size 与 size_: 对 _size 的读、写。
- readerCount 与 readerCount_: 对 _readerCount 的读、写。
- writerTask 与 writerTask_: 对 _writerTask 的读、写。

5. BlockResult

BlockResult 用于封装从本地的 BlockManager 中获取的 Block 数据及与 Block 相关联的度量数据。BlockResult 中有以下属性。

- data: Block 及与 Block 相关联的度量数据。
- readMethod: 读取 Block 的方法。readMethod 采用枚举类型 DataReadMethod 提供的 Memory、Disk、Hadoop、Network 四个枚举值。
- bytes: 读取的 Block 的字节长度。

6. BlockStatus

样例类 BlockStatus 用于封装 Block 的状态信息, 包括以下属性。

- storageLevel: 即 Block 的 StorageLevel。
- memSize: Block 占用的内存大小。
- diskSize: Block 占用的磁盘大小。
- isCached: 是否存储到存储体系中, 即 memSize 与 diskSize 的大小之和是否大于 0。

6.2 Block 信息管理器

按照 BlockInfoManager 的命名, 笔者将本节的标题定为“Block 信息管理器”, BlockInfoManager 的确对 BlockInfo 进行了一些简单的管理, 但是 BlockInfoManager 将主要对 Block 的锁资源进行管理, 当读者阅读了本节的内容, 将会同意笔者的观点。

6.2.1 Block 锁的基本概念

BlockInfoManager 是 BlockManager 内部的子组件之一, BlockInfoManager 对 Block 的锁管理采用了共享锁与排他锁, 其中读锁是共享锁, 写锁是排他锁。表 6-1 展示了 Block 的读锁与写锁之间的互斥性。

表 6-1 Block 的读锁与写锁之间的互斥性

	读 锁	写 锁
读锁	S	X
写锁	X	X

表 6-1 中采用了介绍数据库锁的常规表示方法：S 代表共享，X 代表排他。

要了解 BlockInfoManager 的工作原理，首先应从了解它的属性开始。BlockInfoManager 的成员属性包括以下几个。

- infos：BlockId 与 BlockInfo 之间映射关系的缓存。
- writeLocksByTask：每次任务执行尝试的标识 TaskAttemptId 与执行获取的 Block 的写锁之间的映射关系。TaskAttemptId 与写锁之间是一对多的关系，即一次任务尝试执行会获取零到多个 Block 的写锁。类型 TaskAttemptId 的本质是 Long 类型，其定义如下。

```
private type TaskAttemptId = Long
```

- readLocksByTask：每次任务尝试执行的标识 TaskAttemptId 与执行获取的 Block 的读锁之间的映射关系。TaskAttemptId 与读锁之间是一对多的关系，即一次任务尝试执行会获取零到多个 Block 的读锁，并且会记录对于同一个 Block 的读锁的占用次数。

有了对 BlockInfoManager 中属性的了解，我们可以通过图 6-2 来展示 BlockInfoManager 对 Block 的锁管理。

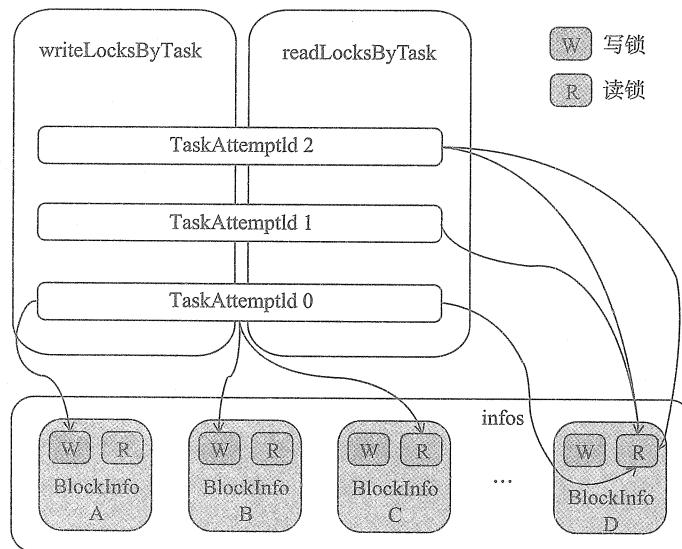


图 6-2 BlockInfoManager 对 Block 的锁管理

从图 6-2，我们能直观地知道以下内容。

- 由 TaskAttemptId 0 标记的任务尝试执行线程获取了 BlockInfo A 和 BlockInfo B 的写锁，并且获取了 BlockInfo C 和 BlockInfo D 的读锁。
- 由 TaskAttemptId 1 标记的任务尝试执行线程获取了 BlockInfo D 的读锁。
- 由 TaskAttemptId 2 标记的任务尝试执行线程多次获取了 BlockInfo D 的读锁，这说明 Block 的读锁是可以重入的。

根据图中三个任务尝试执行线程获取锁的不同展示，我们可以知道，一个任务尝试执行线程可以同时获得零到多个不同 Block 的写锁或零到多个不同 Block 的读锁，但不能同时获得同一个 Block 的读锁与写锁（这点和一些数据库的设计是不同的）。读锁是可以重入的，但是写锁不能重入。

6.2.2 Block 锁的实现

有了 BlockInfoManager 对 Block 的锁管理的了解，现在来看看 BlockInfoManager 提供的方法是如何实现 Block 的锁管理机制的。

- 1) registerTask：注册 TaskAttemptId。registerTask 的实现如代码清单 6-8 所示。

代码清单6-8 注册TaskAttemptId

```
def registerTask(taskAttemptId: TaskAttemptId): Unit = synchronized {
    require(!readLocksByTask.contains(taskAttemptId),
           s"Task attempt $taskAttemptId is already registered")
    readLocksByTask(taskAttemptId) = ConcurrentHashMultiset.create()
}
```

2) currentTaskAttemptId：获取任务上下文 TaskContext 中当前正在执行的任务尝试的 TaskAttemptId。如果任务上下文 TaskContext 中没有任务尝试的 TaskAttemptId，那么返回 BlockInfo.NON_TASK_WRITER。currentTaskAttemptId 方法的实现如代码清单 6-9 所示。

代码清单6-9 currentTaskAttemptId的实现

```
private def currentTaskAttemptId: TaskAttemptId = {
    Option(TaskContext.get()).map(_.taskAttemptId()).getOrElse(BlockInfo.NON_TASK_WRITER)
}
```

- 3) lockForReading：锁定读。lockForReading 的实现如代码清单 6-10 所示。

代码清单6-10 lockForReading的实现

```
def lockForReading(
    blockId: BlockId,
    blocking: Boolean = true): Option[BlockInfo] = synchronized {
    logTrace(s"Task $currentTaskAttemptId trying to acquire read lock for $blockId")
    do {
        infos.get(blockId) match { // 从infos中获取BlockId对应的BlockInfo
            case None => return None
            case Some(blockInfo) =>
                if (blocking) {
                    val lock = new ReadLock(blockInfo)
                    lock.lock()
                    return Some(blockInfo)
                } else
                    return Some(blockInfo)
        }
    } while (true)
}
```

```

    case Some(info) =>
      if (info.writerTask == BlockInfo.NO_WRITER) { // 由当前任务尝试线程持有读锁并
        return BlockInfo
      }
      info.readerCount += 1
      readLocksByTask(currentTaskAttemptId).add(blockId)
      logTrace(s"Task $currentTaskAttemptId acquired read lock for $blockId")
      return Some(info)
    }
  }
  if (blocking) {
    wait() // 等待占用写锁的任务尝试线程释放Block的写锁后唤醒当前线程
  }
} while (blocking)
None
}

```

根据代码清单 6-10，lockForReading 方法的执行步骤如下。

- ① 从 infos 中获取 BlockId 对应的 BlockInfo。如果缓存 infos 中没有对应的 BlockInfo，则返回 None，否则进入下一步。
- ② 如果 Block 的写锁没有被其他任务尝试线程占用，则由当前任务尝试线程持有读锁并返回 BlockInfo，否则进入下一步。
- ③ 如果允许阻塞（即 blocking 为 true），那么当前线程将等待，直到占用写锁的线程释放 Block 的写锁后唤醒当前线程。如果占有写锁的线程一直不释放写锁，那么当前线程将出现“饥饿”状况，即可能无限期等待下去。

4) lockForWriting：锁定写。lockForWriting 的实现如代码清单 6-11 所示。

代码清单6-11 lockForWriting的实现

```

def lockForWriting(
  blockId: BlockId,
  blocking: Boolean = true): Option[BlockInfo] = synchronized {
  logTrace(s"Task $currentTaskAttemptId trying to acquire write lock for
    $blockId")
  do {
    infos.get(blockId) match { // 从infos中获取BlockId对应的BlockInfo
      case None => return None
      case Some(info) =>
        if (info.writerTask == BlockInfo.NO_WRITER && info.readerCount == 0) {
          info.writerTask = currentTaskAttemptId // 由当前任务尝试线程持有写锁
          writeLocksByTask.addBinding(currentTaskAttemptId, blockId)
          logTrace(s"Task $currentTaskAttemptId acquired write lock for $blockId")
          return Some(info)
        }
    }
    if (blocking) {
      wait() // 等待占用写锁的线程释放Block的写锁后唤醒当前线程
    }
  } while (blocking)
  None
}

```

根据代码清单 6-11，lockForWriting 方法的执行步骤如下。

① 从 infos 中获取 BlockId 对应的 BlockInfo。如果缓存 infos 中没有对应的 BlockInfo，则返回 None，否则进入下一步。

② 如果 Block 的写锁没有被其他任务尝试线程占用，且没有线程正在读取此 Block，则由当前任务尝试线程持有写锁并返回 BlockInfo，否则进入下一步。写锁没有被占用并且没有线程正在读取此 Block 的条件也说明了任务尝试执行线程不能同时获得同一个 Block 的读锁与写锁。

③ 如果允许阻塞（即 blocking 为 true），那么当前线程将等待，直到占用写锁的线程释放 Block 的写锁后唤醒当前线程。如果占有写锁的线程一直不释放写锁，那么当前线程将出现“饥饿”状况，即可能无限期等待下去。

小贴士：lockForReading 和 lockForWriting 这两个方法共同实现了写锁与写锁、写锁与读锁之间的互斥性，同时也实现了读锁与读锁之间的共享。此外，这两个方法都提供了阻塞的方式。这种方式在读锁或写锁的争用较少或锁的持有时间都非常短暂，能够带来一定的性能提升。如果获取锁的线程发现锁被占用，就立即失败，然而这个锁很快又被释放了，结果是获取锁的线程无法正常执行。如果获取锁的线程可以等待的话，很快它就发现自己能重新获得锁了，然后推进当前线程继续执行。

- get：获取 BlockId 对应的 BlockInfo。get 方法的实现如代码清单 6-12 所示。

代码清单6-12 获取BlockId对应的BlockInfo

```
private[storage] def get(blockId: BlockId): Option[BlockInfo] = synchronized {
    infos.get(blockId)
}
```

- unlock：释放 BlockId 对应的 Block 上的锁。

代码清单6-13 释放BlockId对应的Block上的锁

```
def unlock(blockId: BlockId): Unit = synchronized {
    logTrace(s"Task $currentTaskAttemptId releasing lock for $blockId")
    val info = get(blockId).getOrElse { // 获取BlockId对应的BlockInfo
        throw new IllegalStateException(s"Block $blockId not found")
    }
    if (info.writerTask != BlockInfo.NO_WRITER) {
        info.writerTask = BlockInfo.NO_WRITER // 释放当前Block的写锁
        writeLocksByTask.removeBinding(currentTaskAttemptId, blockId)
    } else { // 释放当前Block的读锁
        assert(info.readerCount > 0, s"Block $blockId is not locked for reading")
        info.readerCount -= 1
        val countsForTask = readLocksByTask(currentTaskAttemptId)
        val newPinCountForTask: Int = countsForTask.remove(blockId, 1) - 1
        assert(newPinCountForTask >= 0,
            s"Task $currentTaskAttemptId release lock on block $blockId more times than
            it acquired it")
    }
}
```

```

    }
    notifyAll()
}

```

根据代码清单 6-13，unlock 方法的执行步骤如下。

- ① 获取 BlockId 对应的 BlockInfo。
- ② 如果当前任务尝试线程已经获得了 Block 的写锁，则释放当前 Block 的写锁。
- ③ 如果当前任务尝试线程没有获得 Block 的写锁，则释放当前 Block 的读锁。释放读锁实际是减少当前任务尝试线程已经获取的 Block 的读锁次数。

小贴士：上述代码中的 newPinCountForTask 是当前尝试执行线程占有 BlockId 对应 Block 的读锁的次数与 1 的差值，如果大于等于 0，则说明当前尝试执行线程多次获得了这个读锁。

□ downgradeLock：锁降级。downgradeLock 的实现如代码清单 6-14 所示。

代码清单6-14 downgradeLock的实现

```

def downgradeLock(blockId: BlockId): Unit = synchronized {
  logTrace(s"Task $currentTaskAttemptId downgrading write lock for $blockId")
  val info = get(blockId).get // 获取BlockId对应的BlockInfo
  require(info.writerTask == currentTaskAttemptId,
    s"Task $currentTaskAttemptId tried to downgrade a write lock that it does not hold on " +
    s" block $blockId")
  unlock(blockId) // 释放Block的写锁
  val lockOutcome = lockForReading(blockId, blocking = false) // 获取Block的读锁
  assert(lockOutcome.isDefined)
}

```

根据代码清单 6-14，锁降级的过程如下。

- ① 获取 BlockId 对应的 BlockInfo。
- ② 调用 unlock 方法释放当前任务尝试线程从 BlockId 对应 Block 获取的写锁。
- ③ 由于已经释放了 BlockId 对应 Block 的写锁，所以用非阻塞方式获取 BlockId 对应 Block 的读锁。

7) lockNewBlockForWriting：写新 Block 时获得写锁。lockNewBlockForWriting 的实现如代码清单 6-15 所示。

代码清单6-15 写新Block时获得写锁

```

def lockNewBlockForWriting(
  blockId: BlockId,
  newBlockInfo: BlockInfo): Boolean = synchronized {
  logTrace(s"Task $currentTaskAttemptId trying to put $blockId")
  lockForReading(blockId) match { // 获取Block的读锁
    case Some(info) =>
      false
    case None =>

```

```

infos(blockId) = newBlockInfo
lockForWriting(blockId) // 获取Block的写锁
true
}
}

```

根据代码清单 6-15，lockNewBlockForWriting 的执行步骤如下。

① 获取 BlockId 对应的 Block 的读锁。

② 如果上一步能够获取到 Block 的读锁，则说明 BlockId 对应的 Block 已经存在。这种情况发生在多个线程在写同一个 Block 时产生竞争，已经有线程率先一步，当前线程将没有必要再获得写锁，只需要返回 false。

③ 如果第①步没有获取到 Block 的读锁，则说明 BlockId 对应的 Block 还不存在。这种情况下，当前线程首先将 BlockId 与新的 BlockInfo 的映射关系放入 infos，然后获取 BlockId 对应的 Block 的写锁，最后返回 true。

8) releaseAllLocksForTask：释放给定的任务尝试线程所占用的所有 Block 的锁，并通知所有等待获取锁的线程。由于此方法不常使用且实现简单，留给感兴趣的读者自行查阅。

9) size：返回 infos 的大小，即所有 Block 的数量。

10) entries：以迭代器形式返回 infos。entries 的实现如代码清单 6-16 所示。

代码清单6-16 以迭代器形式返回infos

```

def entries: Iterator[(BlockId, BlockInfo)] = synchronized {
    infos.toArray.toIterator
}

```

11) removeBlock：移除 BlockId 对应的 BlockInfo。removeBlock 方法的实现如代码清单 6-17 所示。

代码清单6-17 移除BlockId对应的BlockInfo

```

def removeBlock(blockId: BlockId): Unit = synchronized {
    logTrace(s"Task $currentTaskAttemptId trying to remove block $blockId")
    infos.get(blockId) match { // 获取BlockId对应的BlockInfo
        case Some(blockInfo) =>
            if (blockInfo.writerTask != currentTaskAttemptId) {
                throw new IllegalStateException(
                    s"Task $currentTaskAttemptId called remove() on block $blockId without
                     a write lock")
            } else { // 移除BlockInfo
                infos.remove(blockId)
                blockInfo.readerCount = 0
                blockInfo.writerTask = BlockInfo.NO_WRITER
                writeLocksByTask.removeBinding(currentTaskAttemptId, blockId)
            }
        case None =>
            throw new IllegalArgumentException(
                s"Task $currentTaskAttemptId called remove() on non-existent block
                 $blockId")
    }
}

```

```

    }
    notifyAll() // 通知所有在BlockId对应的Block的锁上等待的线程
}

```

根据代码清单 6-17，removeBlock 的执行步骤如下。

- ① 获取 BlockId 对应的 BlockInfo。
- ② 如果对 BlockInfo 正在写入的任务尝试线程是当前线程的话，当前线程才有权利去移除 BlockInfo。移除 BlockInfo 的操作如下。

- 将 BlockInfo 从 infos 中移除。
 - 将 BlockInfo 的读线程数清零。
 - 将 BlockInfo 的 writerTask 置为 BlockInfo.NO_WRITER。
 - 将任务尝试线程与 BlockId 的关系清除。
- ③ 通知所有在 BlockId 对应的 Block 的锁上等待的线程。

12) clear：清除 BlockInfoManager 中的所有信息，并通知所有在 BlockInfoManager 管理的 Block 的锁上等待的线程。

6.3 磁盘 Block 管理器

DiskBlockManager 是存储体系的成员之一，它负责为逻辑的 Block 与数据写入磁盘的位置之间建立逻辑的映射关系。对于不了解的事物，我们应该先看看它定义了哪些属性。DiskBlockManager 的属性如下。

- conf：即 SparkConf。
- deleteFilesOnStop：停止 DiskBlockManager 的时候是否删除本地目录的布尔类型标记。当不指定外部的 ShuffleClient（即 spark.shuffle.service.enabled 属性为 false）或者当前实例是 Driver 时，此属性为 true。
- localDirs：本地目录的数组。

6.3.1 本地目录结构

localDirs 是 DiskBlockManager 管理的本地目录数组。localDirs 是通过调用 createLocalDirs 方法（见代码清单 6-18）创建的本地目录数组，其实质是调用了 Utils 工具类的 getConfiguredLocalDirs 方法获取本地路径（Utils 的详细介绍请阅读附录 A。getConfiguredLocalDirs 方法默认获取 spark.local.dir 属性或者系统属性 java.io.tmpdir 指定的目录，目录可能有多个），并在每个路径下创建以 blockmgr- 为前缀，UUID 为后缀的随机字符串的子目录，例如，blockmgr-4949e19c-490c-48fc-ad6a-d80f4dbe73df。

代码清单6-18 创建DiskBlockManager的本地目录

```
private def createLocalDirs(conf: SparkConf): Array[File] = {
```

```
Utils.getConfiguredLocalDirs(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
  }
}
```

- ❑ subDirsPerLocalDir：磁盘存储 DiskStore 的本地子目录的数量。可以通过 spark.diskStore.subDirectories 属性配置，默认为 64。
 - ❑ subDirs：DiskStore 的本地子目录的二维数组，即 File[localDirs.length][subDirsPerLocalDir]。
 - ❑ shutdownHook：此属性的作用是在初始化 DiskBlockManager 时，调用 addShutdownHook 方法（见代码清单 6-19），为 DiskBlockManager 设置好关闭钩子。

代码清单6-19 为DiskBlockManager设置关闭钩子

```
private def addShutdownHook(): AnyRef = {
    logDebug("Adding shutdown hook") // force eager creation of logger
    ShutdownHookManager.addShutdownHook(ShutdownHookManager.TEMP_DIR_SHUTDOWN_
        PRIORITY + 1) { () =>
        logInfo("Shutdown hook called")
        DiskBlockManager.this.doStop()
    }
}
```

有了对 localDirs 的了解，让我们通过图 6-3 对 DiskBlockManager 管理的文件目录有一个整体的认识。

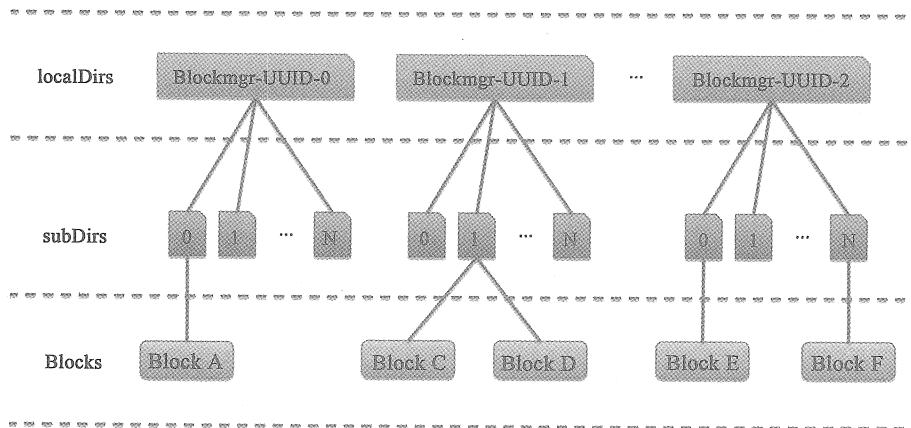


图 6-3 DiskBlockManager 管理的文件目录结构

图 6-3 中显示了一级目录，即 localDirs 数组。图中的一级目录名称都采用了简单的示意表示，例如，Blockmgr-UUID-0、Blockmgr-UUID-1、Blockmgr-UUID-2，代表每个文件夹名称由 Blockmgr- 和 UUID 生成的随机串组成，且此随机串不相同。subDirs 表示 DiskBlockManager 管理的二级目录，每个一级目录下都有 N 个二级目录，这里使用 N 代表 subDirsPerLocalDir 属性的大小。每个二级目录下有若干个 Block 的文件，有些二级目录下可能暂时还没有 Block 文件。有了对 DiskBlockManager 管理的文件目录结构的整体认识，下面来看看 DiskBlockManager 提供了哪些方法来构成和管理这些目录。

6.3.2 DiskBlockManager 提供的方法

1. getFile(filename: String)

此方法根据指定的文件名获取文件。此方法的实现如代码清单 6-20 所示。

代码清单 6-20 根据指定的文件名获取文件

```
def getFile(filename: String): File = {
    val hash = Utils.nonNegativeHash(filename) // 获取文件名的非负哈希值
    val dirId = hash % localDirs.length           // 按照取余方式获得选中的一级目录
    // 获得的余数作为选中的二级目录
    val subDirId = (hash / localDirs.length) % subDirsPerLocalDir
    val subDir = subDirs(dirId).synchronized { // 获取二级目录
        val old = subDirs(dirId)(subDirId)
        if (old != null) {
            old
        } else {
            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.")
            }
            subDirs(dirId)(subDirId) = newDir
            newDir
        }
    }
}

new File(subDir, filename) // 返回二级目录下的文件
}
```

根据代码清单 6-20，getFile 的执行步骤如下。

- 1) 调用 Utils 工具类的 nonNegativeHash 方法获取文件名的非负哈希值。
- 2) 从 localDirs 数组中按照取余方式获得选中的一级目录。
- 3) 哈希值除以一级目录的大小获得商，然后用商数与 subDirsPerLocalDir 取余获得的余数作为选中的二级目录。
- 4) 获取二级目录。如果二级目录不存在，则需要创建二级目录。
- 5) 返回二级目录下的文件。

2. getFile(blockId: BlockId)

此方法根据 BlockId 获取文件。此方法的实现如代码清单 6-21 所示。

代码清单6-21 根据BlockId获取文件

```
def getFile(blockId: BlockId): File = getFile(blockId.name)
```

根据代码清单 6-21, getFile(blockId: BlockId) 实际是以 BlockId 的 name 为参数, 通过调用 getFile(filename: String) 方法实现的。

3. containsBlock

此方法用于检查本地 localDirs 目录中是否包含 BlockId 对应的文件。containsBlock 的实现如代码清单 6-22 所示。

代码清单6-22 containsBlock的实现

```
def containsBlock(blockId: BlockId): Boolean = {
    getFile(blockId.name).exists()
}
```

4. getAllFiles

此方法用于获取本地 localDirs 目录中的所有文件。getAllFiles 的实现如代码清单 6-23 所示。

代码清单6-23 getAllFiles的实现

```
def getAllFiles(): Seq[File] = {
    subDirs.flatMap { dir =>
        dir.synchronized {
            dir.clone()
        }
    }.filter(_ != null).flatMap { dir =>
        val files = dir.listFiles()
        if (files != null) files else Seq.empty
    }
}
```

根据代码清单 6-23 可以知道, getAllFiles 为了保证线程安全, 采用了同步 + 克隆的方式。

5. getAllBlocks

此方法用于获取本地 localDirs 目录中所有 Block 的 BlockId。getAllBlocks 方法的实现如代码清单 6-24 所示。

代码清单6-24 getAllBlocks的实现

```
def getAllBlocks(): Seq[BlockId] = {
    getAllFiles().map(f => BlockId(f.getName()))
}
```

6. createTempLocalBlock

此方法用于为中间结果创建唯一的 BlockId 和文件，此文件将用于保存本地 Block 的数据。createTempLocalBlock 的实现如代码清单 6-25 所示。

代码清单6-25 createTempLocalBlock的实现

```
def createTempLocalBlock(): (TempLocalBlockId, File) = {
    var blockId = new TempLocalBlockId(UUID.randomUUID())
    while (getFile(blockId).exists()) {
        blockId = new TempLocalBlockId(UUID.randomUUID())
    }
    (blockId, getFile(blockId))
}
```

7. createTempShuffleBlock

此方法创建唯一的 BlockId 和文件，用来存储 Shuffle 中间结果（即 map 任务的输出）。createTempShuffleBlock 的实现如代码清单 6-26 所示。

代码清单6-26 createTempShuffleBlock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))
}
```

8. stop

此方法用于正常停止 DiskBlockManager。stop 的实现如代码清单 6-27 所示。

代码清单6-27 停止DiskBlockManager

```
private[spark] def stop() {
    try {
        ShutdownHookManager.removeShutdownHook(shutdownHook)
    } catch {
        case e: Exception =>
            logError(s"Exception while removing shutdown hook.", e)
    }
    doStop()
}
```

从代码清单 6-27 看到，实际停止 DiskBlockManager 的方法为 doStop，其实现如代码清单 6-28 所示。

代码清单6-28 doStop的实现

```
private def doStop(): Unit = {
    if (deleteFilesOnStop) {
```

```

localDirs.foreach { localDir =>
  if (localDir.isDirectory() && localDir.exists()) {
    try {
      if (!ShutdownHookManager.hasRootAsShutdownDeleteDir(localDir)) {
        Utils.deleteRecursively(localDir)
      }
    } catch {
      case e: Exception =>
        logError(s"Exception while deleting local spark dir: $localDir", e)
    }
  }
}
}

```

根据代码清单 6-28，doStop 的主要逻辑是遍历 localDirs 数组中的一级目录，并调用工具类 Utils 的 deleteRecursively 方法（具体实现请参阅附录 A），递归删除一级目录及其子目录或子文件。

6.4 磁盘存储 DiskStore

DiskStore 负责将 Block 存储到磁盘。由于 DiskStore 依赖于 DiskBlockManager 的服务，所以我们在上一节先介绍了 DiskBlockManager。在 Spark 1.x.x 版本中，BlockStore 提供了对磁盘存储 DiskStore、内存存储 MemoryStore 及 Tachyon（即 Alluxio）存储 TachyonStore 的统一规范，DiskStore、MemoryStore 和 TachyonStore 都是具体的实现。但从 Spark 2.0.0 版本开始，取消了 TachyonStore，而且也不再使用 BlockStore 提供统一的接口规范，DiskStore 和 MemoryStore 都是分别实现的。本节将介绍 DiskStore 的实现。

我们首先从了解 DiskStore 的属性开始，逐步理解 DiskStore 发挥的作用，以及 DiskStore 与 DiskBlockManager 之间的关系。DiskStore 的属性有如下几项。

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

小贴士：什么是 FileChannel 的内存镜像映射方法？在 Java NIO 中，FileChannel 的 map 方法所提供的快速读写技术，其实质上是将通道所连接的数据节点中的全部或部分数据直接映射到内存的一个 Buffer 中，而这个内存 Buffer 块就是节点数据的映像，你直接对这个 Buffer 进行修改，会影响到节点数据。这个 Buffer 叫做 MappedBuffer，即镜像 Buffer。由于是内存镜像，因此处理速度快。

有了对 DiskStore 中属性的了解，现在我们来看看 DiskStore 提供的方法。

1. getSize

此方法用于获取给定 BlockId 所对应 Block 的大小。getSize 的实现如代码清单 6-29 所示。

代码清单6-29 获取Block文件的大小

```
def getSize(blockId: BlockId): Long = {
    diskManager.getFile(blockId.name).length
}
```

根据代码清单 6-29，可以看到 DiskStore 的 getSize 方法实质是对 DiskBlockManager 的 getFile 方法的调用，在获取了 File 之后，取得 File 的大小。

2. contains

此方法用于判断本地磁盘存储路径下是否包含给定 BlockId 所对应的 Block 文件。contains 的实现如代码清单 6-30 所示。

代码清单6-30 Block文件是否存在

```
def contains(blockId: BlockId): Boolean = {
    val file = diskManager.getFile(blockId.name)
    file.exists()
}
```

3. remove

此方法用于删除给定 BlockId 所对应的 Block 文件。remove 的实现如代码清单 6-31 所示。

代码清单6-31 删除Block文件

```
def remove(blockId: BlockId): Boolean = {
    val file = diskManager.getFile(blockId.name)
    if (file.exists()) {
        val ret = file.delete()
        if (!ret) {
            logWarning(s"Error deleting ${file.getPath()}")
        }
        ret
    } else {
        false
    }
}
```

4. put

此方法用于将 BlockId 所对应的 Block 写入磁盘。put 的实现如代码清单 6-32 所示。

代码清单6-32 写入Block文件

```
def put(blockId: BlockId)(writeFunc: FileOutputStream => Unit): Unit = {
```

```

if (contains(blockId)) { // 判断给定BlockId所对应的Block文件是否存在
    throw new IllegalStateException(s"Block $blockId is already present in the
        disk store")
}
logDebug(s"Attempting to put block $blockId")
val startTime = System.currentTimeMillis
val file = diskManager.getFile(blockId) // 获取BlockId所对应的Block文件
val fileOutputStream = new FileOutputStream(file)
var threwException: Boolean = true
try {
    writeFunc(fileOutputStream) // 对Block文件写入
    threwException = false
} finally {
    try {
        Closeables.close(fileOutputStream, threwException)
    } finally {
        if (threwException) {
            remove(blockId)
        }
    }
}
val finishTime = System.currentTimeMillis
logDebug("Block %s stored as %s file on disk in %d ms".format(
    file.getName,
    Utils.bytesToString(file.length()),
    finishTime - startTime))
}

```

根据代码清单 6-32，put 的执行步骤如下。

- 1) 调用 contains 方法判断给定 BlockId 所对应的 Block 文件是否存在，当存在时进入下一步。
- 2) 调用 DiskBlockManager 的 getFile 方法获取 BlockId 所对应的 Block 文件，并打开文件输出流。
- 3) 调用回调函数 writeFunc，对 Block 文件写入。当写入失败时，还需要调用 remove 方法删除 BlockId 所对应的 Block 文件。

5. putBytes

此方法用于将 BlockId 所对应的 Block 写入磁盘，Block 的内容已经封装为 ChunkedByteBuffer。putBytes 的实现如代码清单 6-33 所示。

代码清单6-33 putBytes的实现

```

def putBytes(blockId: BlockId, bytes: ChunkedByteBuffer): Unit = {
    put(blockId) { fileOutputStream =>
        val channel = fileOutputStream.getChannel
        Utils.tryWithSafeFinally {
            bytes.writeFully(channel)
        } {
    }
}

```

```
        channel.close()  
    }  
}
```

根据代码清单 6-33，在 putBytes 方法中设置了回调函数，并调用 put 方法。根据 put 方法的实现，我们知道最后会回调外部的回调函数。此回调函数中使用了工具类 Utils 的 tryWithSafeFinally 方法，其详细介绍请阅读附录 A。

6. getBytes

此方法用于读取给定 BlockId 所对应的 Block，并封装为 ChunkedByteBuffer 返回。getBytes 的实现如代码清单 6-34 所示。

代码清单6-34 读取给定BlockId所对应的Block

```

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

```

6.3 节和 6.4 节介绍的都是磁盘存储的内容。下面将介绍内存存储。

6.5 内存管理器

Spark 与 Hadoop 的重要区别之一就在于对内存的使用。Hadoop 只将内存作为计算资源，Spark 除将内存作为计算资源外，还将内存的一部分纳入到存储体系中。Spark 使用 MemoryManager 对存储体系和内存计算所使用的内存进行管理。由于本章主讲存储体系的内容，因此本节将只介绍和存储体系相关的部分，涉及计算引擎的内容，将放在第 8 章介绍。

6.5.1 内存池模型

内存池好比游泳馆的游泳池，只不过游泳池装的是水，内存池装的是内存。游泳馆往往不止一个游泳池，Spark 的存储体系的每个存储节点上也不止一个内存池。内存池实质上是对物理内存的逻辑规划，协助 Spark 任务在运行时合理地使用内存资源。Spark 将内存从逻辑上区分为堆内存和堆外内存，称为内存模式（MemoryMode）。枚举类型 MemoryMode 中定义了堆内存和堆外内存，代码如下。

```
@Private
public enum MemoryMode {
    ON_HEAP,
    OFF_HEAP
}
```

这里的堆内存并不能与 JVM 中的 Java 堆直接画等号，它只是 JVM 堆内存的一部分。堆外内存则是 Spark 使用 sun.misc.Unsafe 的 API 直接在工作节点的系统内存中开辟的空间。无论是哪种内存，都需要一个内存池对内存进行资源管理，抽象类 MemoryPool（见代码清单 6-35）定义了内存池的规范。

代码清单 6-35 内存池的规范

```
private[memory] abstract class MemoryPool(lock: Object) {
    @GuardedBy("lock")
    private[this] var _poolSize: Long = 0
    final def poolSize: Long = lock.synchronized {
        _poolSize
    }
    final def memoryFree: Long = lock.synchronized {
        _poolSize - memoryUsed
    }
    final def incrementPoolSize(delta: Long): Unit = lock.synchronized {
        require(delta >= 0)
        _poolSize += delta
    }
    final def decrementPoolSize(delta: Long): Unit = lock.synchronized {
        require(delta >= 0)
        require(delta <= _poolSize)
        require(_poolSize - delta >= memoryUsed)
        _poolSize -= delta
    }
    def memoryUsed: Long
}
```

根据代码清单 6-35，内存池的基本属性如下。

□ lock：对内存池提供线程安全保证的锁对象。

□ _poolSize：内存池的大小（单位为字节）。

内存池的基本方法如下。

□ poolSize：返回内存池的大小（即 _poolSize，单位为字节）的方法。

- `memoryUsed`：获取已经使用的内存大小（单位为字节）。此方法需要 `MemoryPool` 的子类实现。
- `memoryFree`：获取内存池的空闲空间（即 `_poolSize` 减去 `memoryUsed` 的大小，单位为字节）。
- `incrementPoolSize`：给内存池扩展 `delta` 给定的大小（单位为字节）。`delta` 必须为正整数。
- `decrementPoolSize`：将内存池缩小 `delta` 给定的大小（单位为字节）。`delta` 必须为正整数且 `_poolSize` 与 `delta` 的差要大于等于 `memoryUsed`（即已经使用的内存不能从内存池中移除）。

根据 `MemoryPool` 所定义的内存池规范，我们可以用图 6-4 来表示 `MemoryPool` 的内存模型。

图 6-4 所示的内存模型完全是逻辑上的划分，并不是物理上的内存划分。

Spark 中一共有两种 `MemoryPool` 的具体实现，其继承体系如图 6-5 所示。

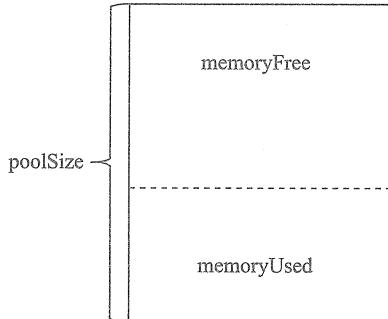


图 6-4 `MemoryPool` 的内存模型

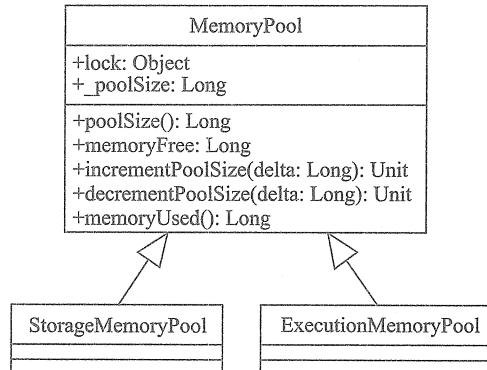


图 6-5 `MemoryPool` 的继承体系

图 6-5 中的 `StorageMemoryPool` 是存储体系用到的内存池，而 `ExecutionMemoryPool` 则是计算引擎用到的内存池。

6.5.2 StorageMemoryPool 详解

Spark 既将内存作为存储体系的一部分，又作为计算引擎所需要的计算资源，因此 `MemoryPool` 既可用于存储体系的实现类 `StorageMemoryPool`，又有用于计算的 `Execution MemoryPool`。由于本章内容着重介绍存储体系，所以本节只介绍 `StorageMemoryPool`，而 `ExecutionMemoryPool` 则留在第 8 章介绍。

`StorageMemoryPool` 是对用于存储的物理内存的逻辑抽象，通过对存储内存的逻辑管理，提高 Spark 存储体系对内存的使用效率。要理解 `StorageMemoryPool`，先从其属性开始。`StorageMemoryPool` 继承了 `MemoryPool` 的 `lock` 和 `_poolSize` 两个属性，还增加了一些

特有的属性。

- ❑ memoryMode：内存模式（MemoryMode）。由此可以断定用于存储的内存池包括堆内存的内存池和堆外内存的内存池。StorageMemoryPool 的堆外内存和堆内存都是逻辑上的区分，前者是使用 sun.misc.Unsafe 的 API 分配的系统内存，后者是系统分配给 JVM 堆内存的一部分。
- ❑ poolName：内存池的名称。如果 memoryMode 是 MemoryMode.ON_HEAP，则内存池名称为 on-heap storage。如果 memoryMode 是 MemoryMode.OFF_HEAP，则内存池名称为 off-heap storage。
- ❑ _memoryUsed：已经使用的内存大小（单位为字节）。
- ❑ _memoryStore：当前 StorageMemoryPool 所关联的 MemoryStore。MemoryStore 将在 6.6 节详细介绍。

了解了 StorageMemoryPool 的属性，现在来看看 StorageMemoryPool 中的方法。

- ❑ memoryUsed：此方法实现了由 MemoryPool 定义的 memoryUsed 接口，实际返回了 _memoryUsed 属性的值。
- ❑ memoryStore：获取当前 StorageMemoryPool 所关联的 MemoryStore，实际返回了 _memoryStore 属性引用的 MemoryStore。
- ❑ setMemoryStore：设置当前 StorageMemoryPool 所关联的 MemoryStore，实际设置了 _memoryStore 属性。
- ❑ releaseAllMemory：释放当前内存池的所有内存，即将 _memoryUsed 设置为 0。

以上方法的实现都很简单，下面介绍一些需要详细分析的方法。

1. acquireMemory

acquireMemory 方法（见代码清单 6-36）用于给 BlockId 对应的 Block 获取 numBytes 指定大小的内存。

代码清单6-36 获取存储Block所需内存

```
def acquireMemory(blockId: BlockId, numBytes: Long): Boolean = lock.synchronized {
    val numBytesToFree = math.max(0, numBytes - memoryFree)
    acquireMemory(blockId, numBytes, numBytesToFree)
}
```

根据代码清单 6-36，此方法首先计算要申请的内存大小 numBytes 与空闲空间 memoryFree 的差值，如果 numBytesToFree 大于 0，则说明需要腾出部分 Block 所占用的内存空间。然后调用重载的 acquireMemory 方法申请获得内存。重载的 acquireMemory 方法（见代码清单 6-37）用于给 BlockId 对应的 Block 获取 Block 所需大小（即 numBytes-ToAcquire）的内存。当 StorageMemoryPool 内存不足时，还需要腾出其他 Block 占用的内存给当前 Block，腾出的大小由 numBytesToFree 属性指定。

代码清单6-37 获取内存

```

def acquireMemory(
    blockId: BlockId,
    numBytesToAcquire: Long,
    numBytesToFree: Long): Boolean = lock.synchronized {
    assert(numBytesToAcquire >= 0)
    assert(numBytesToFree >= 0)
    assert(memoryUsed <= poolSize)
    if (numBytesToFree > 0) { // 腾出numBytesToFree属性指定大小的空间
        memoryStore.evictBlocksToFreeSpace(Some(blockId), numBytesToFree, memoryMode)
    }
    val enoughMemory = numBytesToAcquire <= memoryFree // 判断内存是否充足
    if (enoughMemory) {
        _memoryUsed += numBytesToAcquire // 增加已经使用的内存大小
    }
    enoughMemory // 返回是否成功获得了用于存储Block的内存空间
}

```

根据代码清单 6-37，acquireMemory 方法的执行步骤如下。

- 1) 调用 MemoryStore 的 evictBlocksToFreeSpace 方法（见代码清单 6-57），腾出 numBytesToFree 属性指定大小的空间。
- 2) 判断内存是否充足（即 numBytesToAcquire 是否小于等于 memoryFree）。
- 3) 如果内存充足，则将 numBytesToAcquire 增加到 _memoryUsed（即逻辑上获得了用于存储 Block 的内存空间）。
- 4) 返回布尔值 enoughMemory，即是否成功获得了用于存储 Block 的内存空间。

2. releaseMemory

releaseMemory 方法用于释放内存，其实现如代码清单 6-38 所示。

代码清单6-38 释放内存

```

def releaseMemory(size: Long): Unit = lock.synchronized {
    if (size > _memoryUsed) {
        logWarning(s"Attempted to release $size bytes of storage " +
            s"memory when we only have ${_memoryUsed} bytes")
        _memoryUsed = 0
    } else {
        _memoryUsed -= size
    }
}

```

3. freeSpaceToShrinkPool

freeSpaceToShrinkPool 方法（见代码清单 6-39）用于释放指定大小的空间，缩小内存池的大小。

代码清单6-39 freeSpaceToShrinkPool的实现

```

def freeSpaceToShrinkPool(spaceToFree: Long): Long = lock.synchronized {

```

```

val spaceFreedByReleasingUnusedMemory = math.min(spaceToFree, memoryFree)
val remainingSpaceToFree = spaceToFree - spaceFreedByReleasingUnusedMemory
if (remainingSpaceToFree > 0) {
    val spaceFreedByEviction =
        memoryStore.evictBlocksToFreeSpace(None, remainingSpaceToFree, memoryMode)
    spaceFreedByReleasingUnusedMemory + spaceFreedByEviction
} else {
    spaceFreedByReleasingUnusedMemory
}
}

```

根据代码清单 6-39，freeSpaceToShrinkPool 方法的逻辑为缩小内存池指定的大小（即 spaceToFree）。如果空闲内存（即 memoryFree）大于等于 spaceToFree，那么返回 spaceToFree 即可；否则需要腾出占用内存的部分 Block，以补齐 spaceToFree 的大小，并返回腾出的空间与 memoryFree 的大小之和。由于腾出了占用内存的部分 Block，所以应该减少 _memoryUsed 的大小，但是并未看到代码中如此实现，这是因为调用 evictBlocksToFreeSpace 腾出 Block 时，evictBlocksToFreeSpace 会调用 blockEviction-Handler（即 BlockManager）的 dropFromMemory，而 BlockManager 的 dropFromMemory 方法将会调用 StorageMemoryPool 的 releaseMemory 方法（见代码清单 6-38），因此此处不再需要减少 _memoryUsed 的大小。

6.5.3 MemoryManager 模型

有了 MemoryPool 模型和 StorageMemoryPool 的基础，我们可以来看看抽象类 MemoryManager 定义的内存管理器的接口规范了。了解 MemoryManager 也应当从其属性开始。MemoryManager 的属性中既有和存储体系相关的，也有和任务执行相关的，这些属性如下。

- ❑ conf：即 SparkConf。
- ❑ numCores：CPU 内核数。
- ❑ onHeapStorageMemory：用于存储的堆内存大小。
- ❑ onHeapExecutionMemory：用于执行计算的堆内存大小。
- ❑ onHeapStorageMemoryPool：用于堆内存的存储内存池（StorageMemoryPool），大小由 onHeapStorageMemory 属性指定。
- ❑ offHeapStorageMemoryPool：堆外内存的存储内存池（StorageMemoryPool）。
- ❑ onHeapExecutionMemoryPool：堆内存的执行计算内存池（ExecutionMemoryPool），大小由 onHeapExecutionMemory 属性指定。
- ❑ offHeapExecutionMemoryPool：堆外内存的执行计算内存池（ExecutionMemoryPool）。
- ❑ maxOffHeapMemory：堆外内存的最大值。可以通过 spark.memory.offHeap.size 属性指定，默认为 0。
- ❑ offHeapStorageMemory：用于存储的堆外内存大小。可以通过 spark.memory.storage-

Fraction 属性（默认为 0.5）修改存储占用堆外内存的分数大小来影响 offHeapStorageMemory 的大小。offHeapStorageMemory 的计算公式为：

```
offHeapStorageMemory = maxOffHeapMemory * spark.memory.storageFraction
```

根据我们对 MemoryManager 属性的理解，可以用图 6-6 来表示 MemoryManager 管理的 4 块内存池。

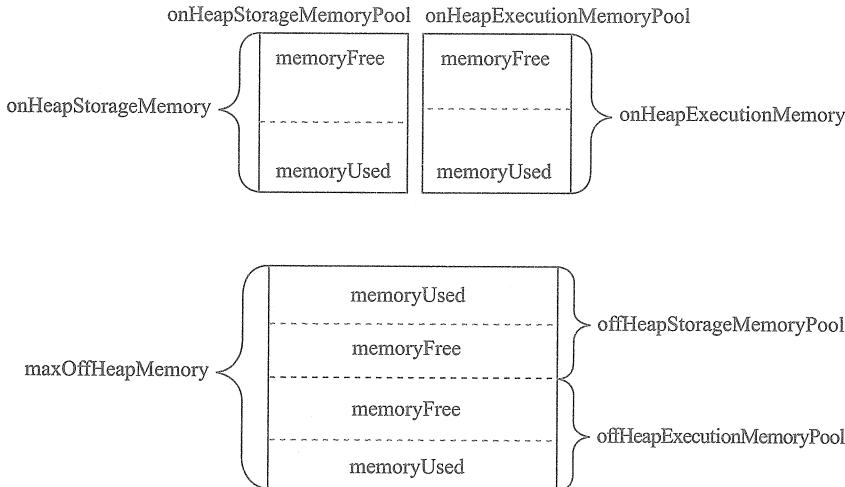


图 6-6 MemoryManager 管理的 4 块内存池

有了对 MemoryManager 管理的内存从整体上的认识，现在来看看 MemoryManager 提供了哪些方法。由于 MemoryManager 中提供的方法既有用于存储体系的，也有用于任务计算的，所以本节只介绍 MemoryManager 中提供的与存储体系有关的方法，这些方法包括如下几个。

- `maxOnHeapStorageMemory`：返回用于存储的最大堆内存。此方法需要子类实现。
- `maxOffHeapStorageMemory`：返回用于存储的最大堆外内存。此方法需要子类实现。
- `setMemoryStore`：给 `onHeapStorageMemoryPool` 和 `offHeapStorageMemoryPool` 设置 `MemoryStore`。`setMemoryStore` 方法（见代码清单 6-40）实际调用了 `StorageMemoryPool` 的 `setMemoryStore` 方法。

代码清单 6-40 setMemoryStore 的实现

```
final def setMemoryStore(store: MemoryStore): Unit = synchronized {
    onHeapStorageMemoryPool.setMemoryStore(store)
    offHeapStorageMemoryPool.setMemoryStore(store)
}
```

- `acquireStorageMemory`：为存储 `BlockId` 对应的 `Block`，从堆内存或堆外内存获取所需大小（即 `numBytes`）的内存。此方法的接口定义如下。

```
def acquireStorageMemory(blockId: BlockId, numBytes: Long, memoryMode: MemoryMode): Boolean
```

- ❑ `acquireUnrollMemory`：为展开 BlockId 对应的 Block，从堆内存或堆外内存获取所需大小（即 numBytes）的内存。此方法的接口定义如下。

```
def acquireUnrollMemory(blockId: BlockId, numBytes: Long, memoryMode: MemoryMode): Boolean
```

- ❑ `releaseStorageMemory`：从堆内存或堆外内存释放指定大小（即 numBytes）的内存。此方法的实现如代码清单 6-41 所示。

代码清单6-41 `releaseStorageMemory`的实现

```
def releaseStorageMemory(numBytes: Long, memoryMode: MemoryMode): Unit = synchronized {
    memoryMode match {
        case MemoryMode.ON_HEAP => onHeapStorageMemoryPool.releaseMemory(numBytes)
        case MemoryMode.OFF_HEAP => offHeapStorageMemoryPool.releaseMemory(numBytes)
    }
}
```

- ❑ `releaseAllStorageMemory`：从堆内存及堆外内存释放所有内存。此方法的实现如代码清单 6-42 所示。

代码清单6-42 `releaseAllStorageMemory`的实现

```
final def releaseAllStorageMemory(): Unit = synchronized {
    onHeapStorageMemoryPool.releaseAllMemory()
    offHeapStorageMemoryPool.releaseAllMemory()
}
```

- ❑ `releaseUnrollMemory`：释放指定大小（即 numBytes）的展开内存。此方法的实现如代码清单 6-43 所示。

代码清单6-43 `releaseUnrollMemory`的实现

```
final def releaseUnrollMemory(numBytes: Long, memoryMode: MemoryMode): Unit =
    synchronized {
        releaseStorageMemory(numBytes, memoryMode)
    }
```

- ❑ `storageMemoryUsed`：`onHeapStorageMemoryPool` 与 `offHeapStorageMemoryPool` 中一共占用的存储内存。此方法的实现如代码清单 6-44 所示。

代码清单6-44 `storageMemoryUsed`的实现

```
final def storageMemoryUsed: Long = synchronized {
    onHeapStorageMemoryPool.memoryUsed + offHeapStorageMemoryPool.memoryUsed
}
```

`MemoryManager` 有两个子类，分别是 `StaticMemoryManager` 和 `UnifiedMemoryManager`。`StaticMemoryManager` 保留了早期 Spark 版本遗留下来的静态内存管理机制，也不存在堆外内存的模型。在静态内存管理机制下，Spark 应用程序在运行期的存储内存和执行内存的大小均为固定的。从 Spark 1.6.0 版本开始，以 `UnifiedMemoryManager` 作为默认的内存管理

器。UnifiedMemoryManager 提供了统一的内存管理机制，即 Spark 应用程序在运行期的存储内存和执行内存将共享统一的内存空间，可以动态调节两块内存的空间大小，所以下一节将着重介绍 UnifiedMemoryManager。

6.5.4 UnifiedMemoryManager 详解

UnifiedMemoryManager 在 MemoryManager 的内存模型之上，将计算内存和存储内存之间的边界修改为“软”边界，即任何一方可以向另一方借用空闲的内存。为了对 UnifiedMemoryManager 的实现原理有更深入的理解，还是先掌握 UnifiedMemoryManager 的成员属性。

- 1) conf: 即 SparkConf。此构造器属性将用于父类 MemoryManager 的构造器属性——conf。
- 2) maxHeapMemory: 最大堆内存。大小为系统可用内存与 spark.memory.fraction 属性值（默认为 0.6）的乘积。
- 3) onHeapStorageRegionSize: 用于存储的堆内存大小。此构造器属性将用于父类 MemoryManager 的构造器属性——onHeapStorageMemory。由于 UnifiedMemoryManager 的构造器属性中没有 onHeapExecutionMemory，所以 maxHeapMemory 与 onHeapStorageRegionSize 的差值就作为父类 MemoryManager 的构造器属性——onHeapExecutionMemory。
- 4) numCores: CPU 内核数。此构造器属性将用于父类 MemoryManager 的构造器属性——numCores。

有了对 UnifiedMemoryManager 属性的了解，我们针对 UnifiedMemoryManager 对图 6-6 中的四块内存池进行调整，如图 6-7 所示。

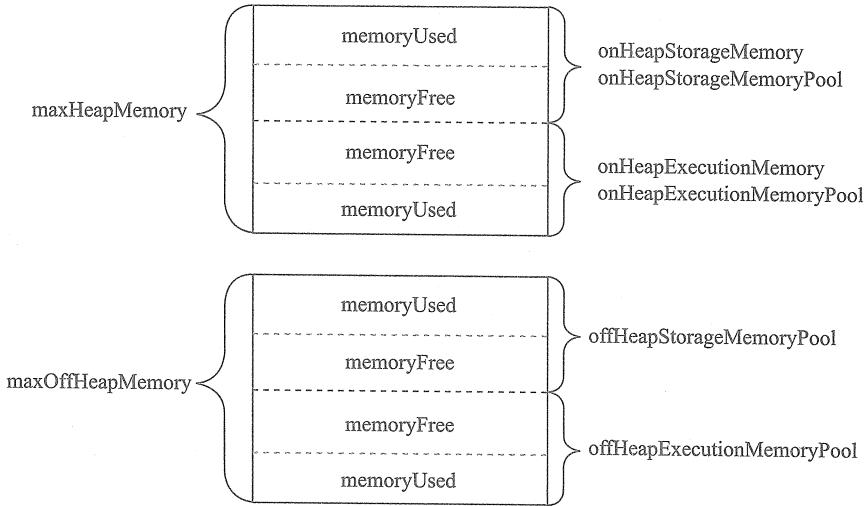


图 6-7 UnifiedMemoryManager 管理的四块内存

图 6-7 将图 6-6 中毫不相干的 onHeapStorageMemoryPool 和 onHeapExecutionMemoryPool 合在了一起，将堆内存作为一个整体看待。而且 onHeapStorageMemoryPool 与 onHeapExecutionMemoryPool 之间，offHeapStorageMemoryPool 与 offHeapExecutionMemoryPool 之间的实线也调整为虚线，表示它们之间都是“软”边界。存储方或计算方的空闲空间（即 memoryFree 表示的区域）都可以借给另一方使用。

有了前面的铺垫，现在来看看 UnifiedMemoryManager 实现的继承自父类 MemoryManager 的方法。由于 UnifiedMemoryManager 中既有存储相关的方法，也有执行内存相关的方法，所以本节只选择存储相关的方法进行介绍，执行内存相关的方法留到第 8 章进行介绍。

1) maxOnHeapStorageMemory：返回用于存储的最大堆内存。此方法的实现如代码清单 6-45 所示。

代码清单 6-45 maxOnHeapStorageMemory 的实现

```
override def maxOnHeapStorageMemory: Long = synchronized {
    maxHeapMemory - onHeapExecutionMemoryPool.memoryUsed
}
```

根据代码清单 6-45，你可能对 maxOnHeapStorageMemory 的实现提出质疑：“maxOnHeapStorageMemory 不是应该等于 onHeapStorageMemory 吗？”这说明你的确理解了 MemoryManager 的内存模型。在本节开头曾经说过，计算内存和存储内存之间的“软”边界和互借内存的概念，所以凡是计算内存没有使用的或空闲的内存，都有可能被借用，作为当前内存管理器用于存储的部分。

2) maxOffHeapStorageMemory：返回用于存储的最大堆外内存。此方法的实现如代码清单 6-46。

代码清单 6-46 maxOffHeapStorageMemory 的实现

```
override def maxOffHeapStorageMemory: Long = synchronized {
    maxOffHeapMemory - offHeapExecutionMemoryPool.memoryUsed
}
```

根据代码清单 6-46，我们知道存储方和计算方不光是堆内存可以互借，堆外内存的空间也可以互借。

3) acquireStorageMemory：为存储 BlockId 对应的 Block，从堆内存或堆外内存获取所需大小（即 numBytes）的内存。此方法的实现如代码清单 6-47 所示。

代码清单 6-47 acquireStorageMemory 的实现

```
override def acquireStorageMemory(
    blockId: BlockId,
    numBytes: Long,
    memoryMode: MemoryMode): Boolean = synchronized {
    assertInvariants()
```

```

assert(numBytes >= 0)
// 获取内存模式对应的计算内存池、存储内存池和可以存储的最大空间
val (executionPool, storagePool, maxMemory) = memoryMode match {
  case MemoryMode.ON_HEAP => (
    onHeapExecutionMemoryPool,
    onHeapStorageMemoryPool,
    maxOnHeapStorageMemory)
  case MemoryMode.OFF_HEAP => (
    offHeapExecutionMemoryPool,
    offHeapStorageMemoryPool,
    maxOffHeapMemory)
}
if (numBytes > maxMemory) { // numBytes不能大于可以存储的最大空间
  logInfo(s"Will not store $blockId as the required space ($numBytes bytes)
  exceeds our " +
  s"memory limit ($maxMemory bytes)")
  return false
}
if (numBytes > storagePool.memoryFree) { // 存储内存池空间不足，到计算内存池中去借用
  space
  val memoryBorrowedFromExecution = Math.min(executionPool.memoryFree, numBytes)
  executionPool.decrementPoolSize(memoryBorrowedFromExecution)
  storagePool.incrementPoolSize(memoryBorrowedFromExecution)
}
storagePool.acquireMemory(blockId, numBytes) //从存储内存池获得存储BlockId对应的
block所需的空间
}

```

根据代码清单 6-47，acquireStorageMemory 方法的执行步骤如下。

- ① 根据内存模式，获取此内存模式的计算内存池、存储内存池和可以存储的最大空间。
 - ② 对要获得的存储大小进行校验，即 numBytes 不能大于可以存储的最大空间。
 - ③ 如果要获得的存储大小比存储内存池的空闲空间要大，那么就到计算内存池中去借用空间。借用的空间取 numBytes 和计算内存池的空闲空间的最小值。
 - ④ 从存储内存池获得存储 BlockId 对应的 Block 所需的空间。
- 4) acquireUnrollMemory：为展开 BlockId 对应的 Block，从堆内存或堆外内存获取所需大小（即 numBytes）的内存。此方法实际代理调用了 acquireStorageMemory，没有任何额外的逻辑，故省略其代码实现。

6.6 内存存储 MemoryStore

MemoryStore 负责将 Block 存储到内存。Spark 通过将广播数据、RDD、Shuffle 数据存储到内存，减少了对磁盘 I/O 的依赖，提高了程序的读写效率。由于 MemoryStore 依赖于 MemoryManager 的服务，所以我们在上一节先介绍了 MemoryManager。下面我们将首先分析 MemoryStore 的内存模型，然后介绍 MemoryStore 提供的方法。

6.6.1 MemoryStore 的内存模型

Block 在内存中以什么形式存在呢？是将文件直接缓存到内存？Spark 将内存中的 Block 抽象为特质 MemoryEntry，其定义如下。

```
private sealed trait MemoryEntry[T] {
    def size: Long
    def memoryMode: MemoryMode
    def classTag: ClassTag[T]
}
```

根据上面的代码可以看到，MemoryEntry 提供了三个接口方法。

- size：当前 Block 的大小。
- memoryMode：Block 存入内存的内存模式。
- classTag：Block 的类型标记。

MemoryEntry 有两个实现类，它们的实现如下。

```
private case class DeserializedMemoryEntry[T] (
    value: Array[T],
    size: Long,
    classTag: ClassTag[T]) extends MemoryEntry[T] {
    val memoryMode: MemoryMode = MemoryMode.ON_HEAP
}
private case class SerializedMemoryEntry[T] (
    buffer: ChunkedByteBuffer,
    memoryMode: MemoryMode,
    classTag: ClassTag[T]) extends MemoryEntry[T] {
    def size: Long = buffer.size
}
```

DeserializedMemoryEntry 表示反序列化后的 MemoryEntry，而 SerializedMemoryEntry 表示序列化后的 MemoryEntry。

为了了解 MemoryStore，我们先来看看 MemoryStore 的属性信息。

- conf：即 SparkConf。
- blockInfoManager：即 Block 信息管理器 BlockInfoManager。
- serializerManager：即序列化管理器 SerializerManager。
- memoryManager：即内存管理器 MemoryManager。MemoryStore 存储 Block，使用的就是 MemoryManager 内的 maxOnHeapStorageMemory 和 maxOffHeapStorageMemory 两块内存池。
- blockEvictionHandler：Block 驱逐处理器。blockEvictionHandler 用于将 Block 从内存中驱逐出去。blockEvictionHandler 的类型是 BlockEvictionHandler，BlockEvictionHandler 定义了将对象从内存中移除的接口，BlockEvictionHandler 的定义如下。

```
private[storage] trait BlockEvictionHandler {
    private[storage] def dropFromMemory[T: ClassTag] (
        blockId: BlockId,
```

```

    data: () => Either[Array[T], ChunkedByteBuffer]): StorageLevel
}

```

BlockManager 实现了特质 BlockEvictionHandler，并重写了 dropFromMemory 方法，BlockManager 在构造 MemoryStore 时，将自身的引用作为 blockEvictionHandler 参数传递给 MemoryStore 的构造器，因而 BlockEvictionHandler 就是 BlockManager。BlockManager 及 BlockManager 重写的 dropFromMemory 方法的实现都将在 6.7 节详细介绍。

- entries：内存中的 BlockId 与 MemoryEntry（Block 的内存形式）之间映射关系的缓存。
- onHeapUnrollMemoryMap：任务尝试线程的标识 TaskAttemptId 与任务尝试线程在堆内存展开的所有 Block 占用的内存大小之和之间的映射关系。
- offHeapUnrollMemoryMap：任务尝试线程的标识 TaskAttemptId 与任务尝试线程在堆外内存展开的所有 Block 占用的内存大小之和之间的映射关系。
- unrollMemoryThreshold：用来展开任何 Block 之前，初始请求的内存大小，可以修改属性 spark.storage.unrollMemoryThreshold（默认为 1MB）改变大小。

MemoryStore 除了以上属性外，还有一些方法对 MemoryStore 的模型提供了概念上的描述。

- maxMemory：MemoryStore 用于存储 Block 的最大内存，其实质为 MemoryManager 的 maxOnHeapStorageMemory 和 maxOffHeapStorageMemory 之和。如果 Memory Manager 为 StaticMemoryManager，那么 maxMemory 的大小是固定的。如果 Memory Manager 为 UnifiedMemoryManager，那么 maxMemory 的大小是动态变化的。
- memoryUsed：MemoryStore 中已经使用的内存大小。其实质为 MemoryManager 中 onHeapStorageMemoryPool 已经使用的大小和 offHeapStorageMemoryPool 已经使用的大小之和。
- currentUnrollMemory：MemoryStore 用于展开 Block 使用的内存大小。其实质为 onHeapUnrollMemoryMap 中的所有用于展开 Block 所占用的内存大小与 offHeapUnrollMemoryMap 中的所有用于展开 Block 所占用的内存大小之和。
- blocksMemoryUsed：MemoryStore 用于存储 Block（即 MemoryEntry）使用的内存大小，即 memoryUsed 与 currentUnrollMemory 的差值。
- currentUnrollMemoryForThisTask：当前的任务尝试线程用于展开 Block 所占用的内存。即 onHeapUnrollMemoryMap 中缓存的当前任务尝试线程对应的占用大小与 offHeapUnrollMemoryMap 中缓存的当前的任务尝试线程对应的占用大小之和。
- numTasksUnrolling：当前使用 MemoryStore 展开 Block 的任务的数量。其实质为 onHeapUnrollMemoryMap 的键集合与 offHeapUnrollMemoryMap 的键集合的并集。

有了对这些 MemoryStore 中成员的了解，我们可以来看看 MemoryStore 的内存模型。

MemoryStore 相比于 MemoryManager，提供了一种宏观的内存模型，MemoryManager 模型的堆内存和堆外内存 在 MemoryStore 的内存模型中是透明的，UnifiedMemoryManager 中存储内存与计算内存的“软”边界在 MemoryStore 的内存模型中也是透明的，读者应该时刻记着这些不同点。MemoryStore 的内存模型如图 6-8 所示。

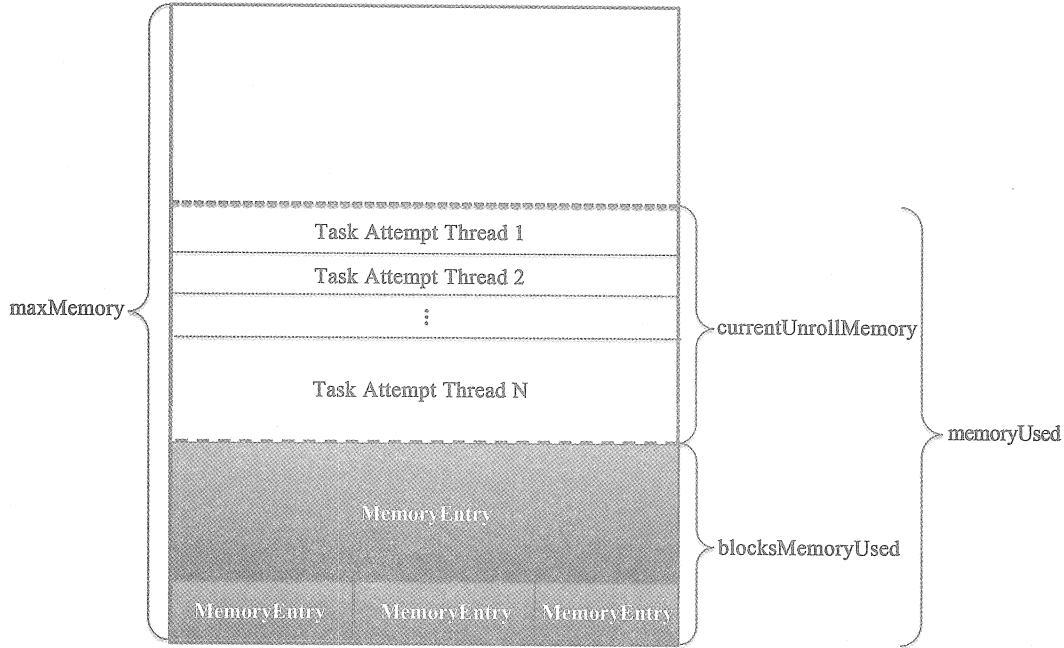


图 6-8 MemoryStore 的内存模型

从图中看出，整个 MemoryStore 的存储分为三块：一块是 MemoryStore 的 entries 属性持有的很多 MemoryEntry 所占据的内存 blocksMemoryUsed；一块是 onHeapUnrollMemoryMap 或 offHeapUnrollMemoryMap 中使用展开方式占用的内存 currentUnrollMemory。展开 Block 的行为类似于人们生活中的“占座”，一间教室里有些座位有人，有些则空着。在座位上放一本书表示有人正在使用，那么别人就不会坐这些座位。这样可以防止在你需要座位的时候，却发现已经没有了位置。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。blocksMemoryUsed 和 currentUnrollMemory 的空间之和是已经使用的空间，用 memoryUsed 表示。还有一块内存没有任何标记，表示未使用。

6.6.2 MemoryStore 提供的方法

MemoryStore 提供了很多方法，便于对 Block 数据的存储和读取。MemoryStore 提供的方法如下。

1. getSize

此方法用于获取 BlockId 对应 MemoryEntry（即 Block 的内存形式）所占用的大小。此

方法的实现如代码清单 6-48 所示。

代码清单 6-48 getSize 的实现

```
def getSize(blockId: BlockId): Long = {
    entries.synchronized {
        entries.get(blockId).size
    }
}
```

2. putBytes

此方法将 BlockId 对应的 Block（已经封装为 ChunkedByteBuffer）写入内存。此方法的实现如代码清单 6-49 所示。

代码清单 6-49 putBytes 的实现

```
def putBytes[T: ClassTag](
    blockId: BlockId,
    size: Long,
    memoryMode: MemoryMode,
    _bytes: () => ChunkedByteBuffer): Boolean = {
    require(!contains(blockId), s"Block $blockId is already present in the MemoryStore")
    if (memoryManager.acquireStorageMemory(blockId, size, memoryMode)) { // 获取逻辑内存
        val bytes = _bytes() // 获取Block的数据
        assert(bytes.size == size)
        val entry = new SerializedMemoryEntry[T](bytes, memoryMode, implicitly[ClassTag[T]])
        entries.synchronized {
            entries.put(blockId, entry) // 将Block数据写入内存
        }
        logInfo("Block %s stored as bytes in memory (estimated size %s, free %s)".format(
            blockId, Utils.bytesToString(size), Utils.bytesToString(maxMemory - blocksMemoryUsed)))
        true
    } else {
        false
    }
}
```

根据代码清单 6-49，putBytes 的执行步骤如下。

- 1) 从 MemoryManager 中获取用于存储 BlockId 对应的 Block 的逻辑内存。如果获取失败则返回 false，否则进入下一步。
- 2) 调用 _bytes 函数，获取 Block 的数据，即 ChunkedByteBuffer。
- 3) 创建 Block 对应的 SerializedMemoryEntry。
- 4) 将 SerializedMemoryEntry 放入 entries 缓存。
- 5) 返回 true。

3. reserveUnrollMemoryForThisTask

此方法用于为展开尝试执行任务给定的 Block，保留指定内存模式上指定大小的内存。reserveUnrollMemoryForThisTask 的实现如代码清单 6-50 所示。

代码清单6-50 reserveUnrollMemoryForThisTask的实现

```

def reserveUnrollMemoryForThisTask(
    blockId: BlockId,
    memory: Long,
    memoryMode: MemoryMode): Boolean = {
    memoryManager.synchronized {
        val success = memoryManager.acquireUnrollMemory(blockId, memory, memoryMode)
        // 获取展开内存
        if (success) {
            val taskAttemptId = currentTaskAttemptId() // 更新taskAttemptId与展开内存大小
                之间的映射关系
            val unrollMemoryMap = memoryMode match {
                case MemoryMode.ON_HEAP => onHeapUnrollMemoryMap
                case MemoryMode.OFF_HEAP => offHeapUnrollMemoryMap
            }
            unrollMemoryMap(taskAttemptId) = unrollMemoryMap.getOrElse(taskAttemptId,
                0L) + memory
        }
        success // 返回获取成功或失败的状态
    }
}

```

根据代码清单 6-50，reserveUnrollMemoryForThisTask 的执行步骤如下。

- 1) 调用 MemoryManager 的 acquireUnrollMemory 方法获取展开内存。
- 2) 如果获取内存成功，则更新 taskAttemptId 与任务尝试线程在堆内存或堆外内存展开的所有 Block 占用的内存大小之和之间的映射关系。
- 3) 返回获取成功或失败的状态。

4. releaseUnrollMemoryForThisTask

此方法用于释放任务尝试线程占用的内存，releaseUnrollMemoryForThisTask 的实现如代码清单 6-51 所示。

代码清单6-51 releaseUnrollMemoryForThisTask的实现

```

def releaseUnrollMemoryForThisTask(memoryMode: MemoryMode, memory: Long = Long.
    MaxValue): Unit = {
    val taskAttemptId = currentTaskAttemptId()
    memoryManager.synchronized {
        val unrollMemoryMap = memoryMode match {
            case MemoryMode.ON_HEAP => onHeapUnrollMemoryMap
            case MemoryMode.OFF_HEAP => offHeapUnrollMemoryMap
        }
        if (unrollMemoryMap.contains(taskAttemptId)) {
            val memoryToRelease = math.min(memory, unrollMemoryMap(taskAttemptId)) // 计
                算要释放的内存
            if (memoryToRelease > 0) { // 释放展开内存
                unrollMemoryMap(taskAttemptId) -= memoryToRelease
                memoryManager.releaseUnrollMemory(memoryToRelease, memoryMode)
            }
            if (unrollMemoryMap(taskAttemptId) == 0) {
                unrollMemoryMap.remove(taskAttemptId) // 清除taskAttemptId与展开内存大小之间
            }
        }
    }
}

```

的映射关系

根据代码清单 6-51，releaseUnrollMemoryForThisTask 的执行步骤如下。

1) 计算实际要释放的内存大小，此大小为指定要释放的大小和任务尝试线程在堆内存或堆外内存实际占有的内存大小之和之间的最小值。

2) 更新 taskAttemptId 与任务尝试线程在堆内存或堆外内存展开的所有 Block 占用的内存大小之和之间的映射关系。

3) 调用 MemoryManager 的 releaseUnrollMemory 方法释放内存。

4) 如果任务尝试线程在堆内存或堆外内存展开的所有 Block 占用的内存大小之和为 0, 则清除此 taskAttemptId 与任务尝试线程在堆内存或堆外内存展开的所有 Block 占用的内存大小之和之间的映射关系。

5. putIteratorAsValues

此方法将 BlockId 对应的 Block (已经转换为 Iterator) 写入内存。有时候放入内存的 Block 很大，所以一次性将此对象写入内存可能将引发 OOM (即 java.lang.OutOfMemoryError) 异常。为了避免这种情况的发生，首先需要将 Block 转换为 Iterator，然后渐进式地展开此 Iterator，并且周期性地检查是否有足够的展开内存。此方法涉及很多变量，为了便于理解，这里先解释这些变量的含义，然后再分析方法实现。

`putIteratorAsValues` 中的变量包括以下几个。

- ❑ elementsUnrolled: 已经展开的元素数量。
 - ❑ keepUnrolling: MemoryStore 是否仍然有足够的内存, 以便于继续展开 Block (即 Iterator)。
 - ❑ initialMemoryThreshold: 即 unrollMemoryThreshold。用来展开任何 Block 之前, 初始请求的内存大小, 可以修改属性 spark.storage.unrollMemoryThreshold (默认为 1MB) 改变大小。
 - ❑ memoryCheckPeriod: 检查内存是否有足够的阀值, 此值固定为 16。字面上有周期的含义, 但是此周期并非指时间, 而是已经展开的元素的数量 elementsUnrolled。
 - ❑ memoryThreshold: 当前任务用于展开 Block 所保留的内存。
 - ❑ memoryGrowthFactor: 展开内存不充足时, 请求增长的因子。此值固定为 1.5。
 - ❑ unrollMemoryUsedByThisBlock: Block 已经使用的展开内存大小, 初始大小为 initialMemoryThreshold。
 - ❑ vector: 用于追踪 Block (即 Iterator) 每次迭代的数据。类型为 SizeTrackingVector。有了对 putIteratorAsValues 方法中各个变量的掌握, 理解 putIteratorAsValues 的实现将容易。putIteratorAsValues 的实现如代码清单 6-52 所示。

代码清单6-52 putIteratorAsValues的实现

```

private[storage] def putIteratorAsValues[T] (
    blockId: BlockId,
    values: Iterator[T],
    classTag: ClassTag[T]): Either[PartiallyUnrolledIterator[T], Long] = {
    // 忽略声明变量的代码
    // 不断迭代读取Iterator中的数据，将数据放入追踪器vector中
    while (values.hasNext && keepUnrolling) {
        vector += values.next()
        if (elementsUnrolled % memoryCheckPeriod == 0) { // 周期性地检查
            val currentSize = vector.estimateSize()
            if (currentSize >= memoryThreshold) {
                val amountToRequest = (currentSize * memoryGrowthFactor - memory-
                    Threshold).toLong
                keepUnrolling =
                    reserveUnrollMemoryForThisTask(blockId, amountToRequest, MemoryMode.ON_HEAP)
                if (keepUnrolling) {
                    unrollMemoryUsedByThisBlock += amountToRequest
                }
                memoryThreshold += amountToRequest
            }
        }
        elementsUnrolled += 1
    }

    if (keepUnrolling) { // 申请到足够多的展开内存，将数据写入内存
        val arrayValues = vector.toArray
        vector = null
        val entry =
            new DeserializedMemoryEntry[T](arrayValues, SizeEstimator.estimate(arrayValues),
                classTag)
        val size = entry.size
        def transferUnrollToStorage(amount: Long): Unit = { // 将展开Block的内存转换为存
            // 储Block的内存
            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")
            }
        }
        val enoughStorageMemory = {
            if (unrollMemoryUsedByThisBlock <= size) {
                val acquiredExtra =
                    memoryManager.acquireStorageMemory(
                        blockId, size - unrollMemoryUsedByThisBlock, MemoryMode.ON_HEAP)
                if (acquiredExtra) {
                    transferUnrollToStorage(unrollMemoryUsedByThisBlock)
                }
                acquiredExtra
            } else { // 当unrollMemoryUsedByThisBlock > size, 归还多余的展开内存空间
                val excessUnrollMemory = unrollMemoryUsedByThisBlock - size
                releaseUnrollMemoryForThisTask(MemoryMode.ON_HEAP, excessUnrollMemory)
                transferUnrollToStorage(size)
                true
            }
        }
    }
}

```

```

        }
    }

    if (enoughStorageMemory) {
        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 {
        assert(currentUnrollMemoryForThisTask >= unrollMemoryUsedByThisBlock,
            "released too much unroll memory")
        Left(new PartiallyUnrolledIterator(
            this,
            MemoryMode.ON_HEAP,
            unrollMemoryUsedByThisBlock,
            unrolled = arrayValues.toIterator,
            rest = Iterator.empty))
    }
} else {
    logUnrollFailureMessage(blockId, vector.estimateSize())
    Left(new PartiallyUnrolledIterator(
        this,
        MemoryMode.ON_HEAP,
        unrollMemoryUsedByThisBlock,
        unrolled = vector.iterator,
        rest = values))
}

```

根据代码清单 6-52，`putIteratorAsValues` 方法的执行步骤如下。

1) 不断迭代读取 Iterator 中的数据，将数据放入追踪器 vector 中，并周期性地检查 vector 中所有数据的估算大小 currentSize 是否已经超过了 memoryThreshold。当发现 currentSize 超过 memoryThreshold，则为当前任务请求新的保留内存（内存大小的计算公式为： $\text{currentSize} * \text{memoryGrowthFactor} - \text{memoryThreshold}$ ）。在堆上成功申请到足够的内存后，需要更新 unrollMemoryUsedByThisBlock 和 memoryThreshold 的大小。

2) 如果展开 Iterator 中所有的数据后, keepUnrolling 为 true, 则说明已经为 Block 申请到足够多的保留内存。接下来的处理步骤如下。

- ① 将 vector 中的数据封装为 DeserializedMemoryEntry，并重新估算 vector 的大小 size。
 - ② 如果 unrollMemoryUsedByThisBlock 大于 size，说明用于展开的内存过多，需要 MemoryManager 归还多余的空间。归还的内存大小为 unrollMemoryUsedByThisBlock - size。之后调用 transferUnrollToStorage 方法将展开 Block 占用的内存转换为用于存储 Block 的内存，此转换过程是原子的。
 - ③ 如果有足够的内存存储 Block，则将 BlockId 与 DeserializedMemoryEntry 的映射关系插入 entries 并返回 Right (size)。

④ 如果没有足够的内存存储 Block，则创建 PartiallyUnrolledIterator 并返回 Left。

3) 如果展开 Iterator 中所有的数据后，keepUnrolling 为 false，说明没有为 Block 申请到足够多的保留内存，此时将创建 PartiallyUnrolledIterator 并返回 Left。

6. putIteratorAsBytes

此方法以序列化后的字节数组方式，将 BlockId 对应的 Block（已经转换为 Iterator）写入内存。由于此方法的实现与 putIteratorAsValues 类似，所以留给感兴趣的读者自行阅读理解。

7. getBytes

此方法从内存中读取 BlockId 对应的 Block（已经封装为 ChunkedByteBuffer）。此方法的实现如代码清单 6-53 所示。

代码清单6-53 getBytes的实现

```
def getBytes(blockId: BlockId): Option[ChunkedByteBuffer] = {
    val entry = entries.synchronized { entries.get(blockId) }
    entry match {
        case null => None
        case e: DeserializedMemoryEntry[_] =>
            throw new IllegalArgumentException("should only call getBytes on serialized blocks")
        case SerializedMemoryEntry(bytes, _, _) => Some(bytes)
    }
}
```

根据代码清单 6-53，我们知道 getBytes 只能获取序列化的 Block。

8. getValues

此方法用于从内存中读取 BlockId 对应的 Block（已经封装为 Iterator）。此方法的实现如代码清单 6-54 所示。

代码清单6-54 getValues的实现

```
def getValues(blockId: BlockId): Option[Iterator[_]] = {
    val entry = entries.synchronized { entries.get(blockId) }
    entry match {
        case null => None
        case e: SerializedMemoryEntry[_] =>
            throw new IllegalArgumentException("should only call getValues on
deserialized blocks")
        case DeserializedMemoryEntry(values, _, _) =>
            val x = Some(values)
            x.map(_.iterator)
    }
}
```

根据代码清单 6-54，我们知道 getValues 只能获取没有序列化的 Block。

9. remove

此方法用于从内存中移除 BlockId 对应的 Block，remove 的实现如代码清单 6-55 所示。

代码清单6-55 移除Block

```

def remove(blockId: BlockId): Boolean = memoryManager.synchronized {
    val entry = entries.synchronized {
        entries.remove(blockId) // 将Block从内存移除
    }
    if (entry != null) {
        entry match {
            case SerializedMemoryEntry(buffer, _, _) => buffer.dispose()
            case _ =>
        }
        // 释放逻辑的存储内存
        memoryManager.releaseStorageMemory(entry.size, entry.memoryMode)
        logDebug(s"Block $blockId of size ${entry.size} dropped " +
            s"from memory (free ${maxMemory - blocksMemoryUsed})")
        true // 移除成功
    } else {
        false // 移除失败
    }
}

```

根据代码清单 6-55，remove 的执行步骤如下。

- 1) 将 BlockId 对应的 MemoryEntry 从 entries 中移除。如果 entries 中存在 BlockId 对应的 MemoryEntry，则进入第 2) 步，否则返回 false。
- 2) 如果 MemoryEntry 是 SerializedMemoryEntry，则还要将对应的 ChunkedByteBuffer 清理。
- 3) 调用 MemoryManager 的 releaseStorageMemory 方法，释放使用的存储内存。
- 4) 返回 true。

10. clear

此方法用于清空 MemoryStore，实现如代码清单 6-56 所示。

代码清单6-56 清空MemoryStore

```

def clear(): Unit = memoryManager.synchronized {
    entries.synchronized {
        entries.clear()
    }
    onHeapUnrollMemoryMap.clear()
    offHeapUnrollMemoryMap.clear()
    memoryManager.releaseAllStorageMemory()
    logInfo("MemoryStore cleared")
}

```

11. evictBlocksToFreeSpace

此方法用于驱逐 Block，以便释放一些空间来存储新的 Block，具体实现如代码清单 6-57 所示。

代码清单6-57 驱逐Block

```

private[spark] def evictBlocksToFreeSpace(
    blockId: Option[BlockId],
    space: Long,
    memoryMode: MemoryMode): Long = {
  assert(space > 0)
  memoryManager.synchronized {
    var freedMemory = 0L
    val rddToAdd = blockId.flatMap(getRddId)
    val selectedBlocks = new ArrayBuffer[BlockId]
    def blockIsEvictable(blockId: BlockId, entry: MemoryEntry[_]): Boolean = {
      entry.memoryMode == memoryMode && (rddToAdd.isEmpty || rddToAdd != getRddId(blockId))
    }
    entries.synchronized {
      val iterator = entries.entrySet().iterator()
      while (freedMemory < space && iterator.hasNext) { // 选择符合驱逐条件的Block
        val pair = iterator.next()
        val blockId = pair.getKey
        val entry = pair.getValue
        if (blockIsEvictable(blockId, entry)) {
          if (blockInfoManager.lockForWriting(blockId, blocking = false).isDefined) {
            selectedBlocks += blockId
            freedMemory += pair.getValue.size
          }
        }
      }
    }
  }

  def dropBlock[T](blockId: BlockId, entry: MemoryEntry[T]): Unit = {
    val data = entry match {
      case DeserializedMemoryEntry(values, _, _) => Left(values)
      case SerializedMemoryEntry(buffer, _, _) => Right(buffer)
    }
    val newEffectiveStorageLevel =
      blockEvictionHandler.dropFromMemory(blockId, () => data)(entry.classTag)
    if (newEffectiveStorageLevel.isValid) {
      blockInfoManager.unlock(blockId)
    } else {
      blockInfoManager.removeBlock(blockId)
    }
  }
}

if (freedMemory >= space) { // 通过驱逐可以为存储Block提供足够的空间，则进行驱逐
  logInfo(s"${selectedBlocks.size} blocks selected for dropping " +
    s"(${Utils.bytesToString(freedMemory)} bytes)")
  for (blockId <- selectedBlocks) {
    val entry = entries.synchronized { entries.get(blockId) }
    if (entry != null) {
      dropBlock(blockId, entry)
    }
  }
  logInfo(s"After dropping ${selectedBlocks.size} blocks, " +
    s"free memory is ${Utils.bytesToString(maxMemory - blocksMemoryUsed)}")
}

```

```
    freedMemory
} else { // 通过驱逐不能为存储Block提供足够的空间，则释放原本准备要驱逐的各个Block的写锁
    blockId.foreach { id =>
        logInfo(s"Will not store $id")
    }
    selectedBlocks.foreach { id =>
        blockInfoManager.unlock(id)
    }
    OL
}
}
```

根据代码清单 6-57，`evictBlocksToFreeSpace` 中定义了一些局部变量。

- ❑ blockId：要存储的 Block 的 BlockId。
 - ❑ space：需要驱逐 Block 所腾出的内存大小。
 - ❑ memoryMode：存储 Block 所需的内存模式。
 - ❑ freedMemory：已经释放的内存大小。
 - ❑ rddToAdd：将要添加的 RDD 的 RDDBlockId 标记。rddToAdd 实际是通过对 BlockId 应用 getRddId 方法得到的。getRddId 的实现如下。

```
private def getRddId(blockId: BlockId): Option[Int] = {
  blockId.asRDDId.map(_.rddId)
}
```

上述代码说明首先调用了 BlockId 的 asRDDId 方法（见代码清单 6-3），将 BlockId 转换为 RDDBlockId，然后获取 RDDBlockId 的 rddId 属性。

- selectedBlocks: 已经选择的用于驱逐的 Block 的 BlockId 的数组。

有了对以上变量的理解，现在来看看 evictBlocksToFreeSpace 的执行步骤。

1) 当 freedMemory 小于 space 时, 不断迭代遍历 iterator。对于每个 entries 中的 BlockId 和 MemoryEntry, 首先找出其中符合条件的 Block (只有符合条件的 Block 才会被驱逐), 然后获取 Block 的写锁, 最后将此 Block 的 BlockId 放入 selectedBlocks 并且将 freedMemory 增加 Block 的大小。哪些 Block 将符合条件呢? 同时满足以下两个条件的。

- ① MemoryEntry 的内存模式与所需的内存模式一致。

- ② BlockId 对应的 Block 不是 RDD，或者 BlockId 与 blockId 不是同一个 RDD

2) 经过了第 1) 步的处理, 如果 freedMemory 大于等于 space, 这说明通过驱逐一定数量的 Block, 已经为存储 blockId 对应的 Block 腾出了足够的内存空间, 此时需要遍历 selectedBlocks 中的每个 BlockId, 并移除每个 BlockId 对应的 Block。如果 Block 从内存中迁移到其他存储(如 DiskStore)中, 那么需要调用 BlockInfoManager 的 unlock 释放当前任务尝试线程获取的被迁移 Block 的写锁。如果 Block 从存储体系中彻底移除了, 那么需要调用 BlockInfoManager 的 removeBlock 方法删除被迁移 Block 的信息。

- 3) 经过了第 1) 步的处理, 如果 freedMemory 小于 space。这说明即使驱逐内存中所

有符合条件的 Block，腾出的空间也不足以存储 blockId 对应的 Block，此时需要当前任务尝试线程释放 selectedBlocks 中每个 BlockId 对应的 Block 的写锁。



注意 有些读者可能会发现对于每个符合条件的 Block，都通过调用 BlockInfoManager 的 lockForWriting 方法获取 Block 的写锁。这是因为不应该对有线程正在读取或写入的 Block 进行驱逐，只能驱逐那些没有被其他线程读或写的 Block。

12. contains

此方法用于判断本地 MemoryStore 中是否包含给定 BlockId 所对应的 Block 文件。contains 的实现如代码清单 6-58 所示。

代码清单6-58 Block文件是否存在

```
def contains(blockId: BlockId): Boolean = {
    entries.synchronized { entries.containsKey(blockId) }
}
```

6.7 块管理器 BlockManager

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

6.7.1 BlockManager 的初始化

根据 5.7 节的介绍，每个 Driver 或 Executor 在创建自身的 SparkEnv 时都会创建 BlockManager。BlockManager 只有在其 initialize 方法被调用后才能发挥作用，其实现如代码清单 6-59 所示。

代码清单6-59 初始化BlockManager

```
def initialize(appId: String): Unit = {
    blockTransferService.init(this)
    shuffleClient.init(appId)
    blockReplicationPolicy = { // 设置Block的复制策略
        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
    }
}
```

```

    }

    val id =
      BlockManagerId(executorId, blockTransferService.hostName, blockTransfer-
      Service.port, None)

    val idFromMaster = master.registerBlockManager(
      id,
      maxMemory,
      slaveEndpoint)
    // 生成当前BlockManager的BlockManagerId
    blockManagerId = if (idFromMaster != null) idFromMaster else id
    // 生成shuffleServerId
    shuffleServerId = if (externalShuffleServiceEnabled) {
      logInfo(s"external shuffle service port = $externalShuffleServicePort")
      BlockManagerId(executorId, blockTransferService.hostName, externalShuffle
        ServicePort)
    } else {
      blockManagerId
    }

    if (externalShuffleServiceEnabled && !blockManagerId.isDriver) {
      registerWithExternalShuffleServer() // 注册外部的Shuffle服务
    }

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

```

根据代码清单 6-59，BlockManager 初始化的步骤如下。

- 1) 初始化 BlockTransferService，即 5.7 节介绍的 NettyBlockTransferService。
- 2) 初始化 Shuffle 客户端。有关 shuffleClient 的内容将在 6.9 节详细介绍。
- 3) 设置 Block 的复制策略。可以通过 spark.storage.replication.policy 属性来设置 Block 的复制策略，默认为 RandomBlockReplicationPolicy。这里使用了工具类 Utils 的 classForName 方法加载 Class，其具体实现可以参阅附录 A。
- 4) 生成当前 BlockManager 的 BlockManagerId。BlockManager 在本地创建的 BlockManagerId 实际只是在向 BlockManagerMaster 注册 BlockManager 时，给 BlockManagerMaster 提供参考，BlockManagerMaster 将会创建一个包含了拓扑信息的新 BlockManagerId 作为正式分配给 BlockManager 的身份标识。
- 5) 生成 shuffleServerId。当启用了外部 Shuffle 服务时将新建一个 BlockManagerId 作为 shuffleServerId，否则是 BlockManager 自身的 BlockManagerId。
- 6) 当启用了外部 Shuffle 服务，并且当前 BlockManager 所在节点不是 Driver 时，需要注册外部的 Shuffle 服务。

6.7.2 BlockManager 提供的方法

BlockManager 提供了很多方法，这些方法将为 Spark 的存储体系提供支持，所以有必

要将它们介绍一番。读者在初次阅读本书时不必了解代码实现，只需要知道 BlockManager 提供了哪些方法即可，在介绍具体的调用时，可以回来查阅其实现，帮助我们理解。

1. reregister

此方法用于向 BlockManagerMaster 重新注册 BlockManager，并向 BlockManagerMaster 报告所有的 Block 信息。reregister 的实现如代码清单 6-60 所示。

代码清单6-60 reregister的实现

```
def reregister(): Unit = {
    master.registerBlockManager(blockManagerId, maxMemory, slaveEndpoint)
    reportAllBlocks()
}
```

根据代码清单 6-60，reregister 的步骤如下。

- 1) 调用 BlockManagerMaster 的 registerBlockManager 方法（见代码清单 6-90）向 BlockManagerMaster 注册 BlockManager。

- 2) 调用 reportAllBlocks 方法报告所有的 Block 信息。

reportAllBlocks 的实现如下。

```
private def reportAllBlocks(): Unit = {
    logInfo(s"Reporting ${blockInfoManager.size} blocks to the master.")
    for ((blockId, info) <- blockInfoManager.entries) {
        val status = getCurrentBlockStatus(blockId, info)
        if (info.tellMaster && !tryToReportBlockStatus(blockId, status)) {
            logError(s"Failed to report $blockId to master; giving up.")
            return
        }
    }
}
```

可以看到，reportAllBlocks 方法遍历 BlockInfoManager 管理的所有 BlockId 与 BlockInfo 的映射关系，并进行如下操作。

- 1) 调用 getCurrentBlockStatus 方法（见代码清单 6-61），获取 Block 的状态信息 Block Status。

- 2) 如果需要将 Block 的 BlockStatus 汇报给 BlockManagerMaster，则调用 tryToReportBlockStatus 方法（见代码清单 6-62），向 BlockManagerMaster 汇报此 Block 的状态信息。

代码清单6-61 获取Block的状态信息

```
private def getCurrentBlockStatus(blockId: BlockId, info: BlockInfo): BlockStatus = {
    info.synchronized {
        info.level match {
            case null =>
                BlockStatus.empty
            case level =>
                val inMem = level.useMemory && memoryStore.contains(blockId)
                val onDisk = level.useDisk && diskStore.contains(blockId)
                val deserialized = if (inMem) level.deserialized else false
                BlockStatus(BlockStatus.State(BlockStatus.StateType.ON_DISK, inMem), onDisk, deserialized)
        }
    }
}
```

```
    val replication = if (inMem || onDisk) level.replication else 1
    val storageLevel = StorageLevel(
        useDisk = onDisk,
        useMemory = inMem,
        useOffHeap = level.useOffHeap,
        deserialized = deserialized,
        replication = replication)
    val memSize = if (inMem) memoryStore.getSize(blockId) else 0L
    val diskSize = if (onDisk) diskStore.getSize(blockId) else 0L
    BlockStatus(storageLevel, memSize, diskSize)
}
}
```

代码清单6-62 向BlockManagerMaster汇报Block的状态信息

```
private def tryToReportBlockStatus(
    blockId: BlockId,
    status: BlockStatus,
    droppedMemorySize: Long = 0L): Boolean = {
  val storageLevel = status.storageLevel
  val inMemSize = Math.max(status.memSize, droppedMemorySize)
  val onDiskSize = status.diskSize
  master.updateBlockInfo(blockManagerId, blockId, storageLevel, inMemSize, onDiskSize)
}
```

根据代码清单 6-62 可以看到，向 BlockManagerMaster 汇报 Block 的状态信息是通过调用 BlockManagerMaster 的 updateBlockInfo 方法完成的。BlockManagerMaster 的 updateBlockInfo 方法将向 BlockManagerMasterEndpoint 发送 UpdateBlockInfo 消息。BlockManagerMasterEndpoint 对 UpdateBlockInfo 消息的处理如代码清单 6-91 所示。

2. getLocalBytes

此方法用于从存储体系获取 BlockId 所对应 Block 的数据，并封装为 ChunkedByteBuffer 后返回，其实现如代码清单 6-63 所示。

代码清单6-63 getLocalBytes的实现

```
def getLocalBytes(blockId: BlockId): Option[ChunkedByteBuffer] = {
    logDebug(s"Getting local block $blockId as bytes")
    if (blockId.isShuffle) {
        val shuffleBlockResolver = shuffleManager.shuffleBlockResolver
        Option(new ChunkedByteBuffer(
            shuffleBlockResolver.getBlockData(blockId.asInstanceOf[ShuffleBlockId]),
            nioByteBuffer())))
    } else {
        blockInfoManager.lockForReading(blockId).map { info => doGetLocalBytes(
            blockId, info) }
    }
}
```

根据代码清单 6-63，`getLocalBytes` 的执行步骤如下。

1) 如果当前 Block 是 ShuffleBlock，那么调用 ShuffleManager 的 ShuffleBlockResolver 组件的 getBlockData 方法（ShuffleBlockResolver 将在 8.6 节详细介绍）获取 Block 数据，并封装为 ChunkedByteBuffer 返回。

2) 如果当前 Block 不是 ShuffleBlock，那么首先获取 Block 的读锁，然后调用 doGetLocalBytes 方法获取 Block 数据。

doGetLocalBytes 的实现如代码清单 6-64 所示。

代码清单 6-64 doGetLocalBytes 的实现

```

private def doGetLocalBytes(blockId: BlockId, info: BlockInfo): ChunkedByteBuffer = {
    val level = info.level // 获取Block的存储级别
    logDebug(s"Level for block $blockId is $level")
    if (level.deserialized) { // Block没有被序列化，按照DiskStore、MemoryStore的顺序获取Block数据
        if (level.useDisk && diskStore.contains(blockId)) {
            diskStore.getBytes(blockId)
        } else if (level.useMemory && memoryStore.contains(blockId)) {
            serializerManager.dataSerializeWithExplicitClassTag(
                blockId, memoryStore.getValues(blockId).get, info.classTag)
        } else {
            handleLocalReadFailure(blockId)
        }
    } else { // Block被序列化了，那么按照MemoryStore、DiskStore的顺序获取Block数据
        if (level.useMemory && memoryStore.contains(blockId)) {
            memoryStore.getBytes(blockId).get
        } else if (level.useDisk && diskStore.contains(blockId)) {
            val diskBytes = diskStore.getBytes(blockId)
            maybeCacheDiskBytesInMemory(info, blockId, level, diskBytes).getOrElse
                (diskBytes)
        } else {
            handleLocalReadFailure(blockId)
        }
    }
}
}

```

根据代码清单 6-64，doGetLocalBytes 的执行步骤如下。

- 1) 获取 Block 的存储级别。
- 2) 如果 Block 的存储级别说明 Block 没有被序列化，那么按照 DiskStore、Memory Store 的顺序，获取 Block 数据。
- 3) 如果 Block 存储级别说明 Block 被序列化了，那么按照 MemoryStore、DiskStore 的顺序，获取 Block 数据。

3. getBlockData

此方法用于获取本地 Block 的数据，其实现如代码清单 6-65 所示。

代码清单 6-65 获取本地Block的数据

```

override def getBlockData(blockId: BlockId): ManagedBuffer = {
    if (blockId.isShuffle) { // 当前Block是ShuffleBlock

```

```

        shuffleManager.shuffleBlockResolver.getBlockData(blockId.asInstanceOf[  

          ShuffleBlockId])  

    } else { // 当前Block不是ShuffleBlock  

      getLocalBytes(blockId) match {  

        case Some(buffer) => new BlockManagerManagedBuffer(blockInfoManager,  

          blockId, buffer)  

        case None =>  

          reportBlockStatus(blockId, BlockStatus.empty)  

          throw new BlockNotFoundException(blockId.toString)  

      }  

    }  

}

```

根据代码清单 6-65，getBlockData 的执行步骤如下。

- 1) 如果当前 Block 是 ShuffleBlock，那么调用 ShuffleManager 的 ShuffleBlockResolver 组件的 getBlockData 方法获取 Block 数据。
- 2) 如果当前 Block 不是 ShuffleBlock，那么调用 getLocalBytes 获取 Block 数据。如果调用 getLocalBytes 能够获取到 Block 数据，则封装为 BlockManagerManagedBuffer，否则调用 reportBlockStatus 方法（见代码清单 6-66）告诉 BlockManagerMaster，此 Block 不存在。

代码清单6-66 reportBlockStatus的实现

```

private def reportBlockStatus(  

  blockId: BlockId,  

  status: BlockStatus,  

  droppedMemorySize: Long = 0L): Unit = {  

  val needReregister = !tryToReportBlockStatus(blockId, status, droppedMemorySize)  

  if (needReregister) {  

    logInfo(s"Got told to re-register updating block $blockId")  

    asyncReregister()  

  }  

  logDebug(s"Told master about block $blockId")
}

```

根据代码清单 6-66，reportBlockStatus 的执行步骤如下。

- 1) 调用 tryToReportBlockStatus 方法向 BlockManagerMaster 汇报 BlockStatus。
 - 2) 如果第 1) 步返回 true，则说明需要重新向 BlockManagerMaster 注册当前 BlockManager，因而调用 asyncReregister 方法向 BlockManagerMaster 异步注册 BlockManager。
- asyncReregister 方法实际另起线程调用 reregister，来实现异步注册 BlockManager。asyncReregister 的实现如下。

```

private def asyncReregister(): Unit = {  

  asyncReregisterLock.synchronized {  

    if (asyncReregisterTask == null) {  

      asyncReregisterTask = Future[Unit] {  

        reregister()  

        asyncReregisterLock.synchronized {  

          asyncReregisterTask = null
      }
    }
  }
}

```

```
        }
    } (futureExecutionContext)
}
}
```

4. putBytes

要介绍 putBytes，需要首先介绍 doPut。doPut 用于执行 Block 的写入，其实现如代码清单 6-67 所示。

代码清单6-67 doPut的实现

```
private def doPut[T] (blockId: BlockId,
                     level: StorageLevel,
                     classTag: ClassTag[_],
                     tellMaster: Boolean,
                     keepReadLock: Boolean) (putBody: BlockInfo => Option[T]): Option[T] = {
    require(blockId != null, "BlockId is null")
    require(level != null && level.isValid, "StorageLevel is null or invalid")

    val putBlockInfo = {
        val newInfo = new BlockInfo(level, classTag, tellMaster)
        if (blockInfoManager.lockNewBlockForWriting(blockId, newInfo)) { // 获取Block的写锁
            newInfo
        } else {
            logWarning(s"Block $blockId already exists on this machine; not re-adding it")
            if (!keepReadLock) {
                // lockNewBlockForWriting returned a read lock on the existing block, so we must free it:
                releaseLock(blockId)
            }
            return None
        }
    }
    val startTimeMs = System.currentTimeMillis
    var exceptionWasThrown: Boolean = true
    val result: Option[T] = try {
        val res = putBody(putBlockInfo) // 执行Block写入
        exceptionWasThrown = false
        if (res.isEmpty) {
            // Block成功存储, 执行锁降级或释放锁
            if (keepReadLock) {
                blockInfoManager.downgradeLock(blockId)
            } else {
                blockInfoManager.unlock(blockId)
            }
        } else { // Block存储失败, 移除此Block
            removeBlockInternal(blockId, tellMaster = false)
            logWarning(s"Putting block $blockId failed")
        }
        res
    } finally {
```

```

    if (exceptionWasThrown) {
      logWarning(s"Putting block $blockId failed due to an exception")
      removeBlockInternal(blockId, tellMaster = tellMaster)
      addUpdatedBlockStatusToTaskMetrics(blockId, BlockStatus.empty)
    }
  }
  if (level.replication > 1) {
    logDebug("Putting block %s with replication took %s"
      .format(blockId, Utils.getUsedTimeMs(startTimeMs)))
  } else {
    logDebug("Putting block %s without replication took %s"
      .format(blockId, Utils.getUsedTimeMs(startTimeMs)))
  }
  result
}

```

根据代码清单 6-67，我们看到 doPut 有一个函数参数 putBody，putBody 将执行真正的 Block 数据写入。doPut 的执行步骤如下。

- 1) 获取 Block 的写锁。如果 Block 已经存在且不需要持有读锁，则需要当前线程释放持有的读锁。
- 2) 调用 putBody，执行写入。
- 3) 如果写入成功，则在需要保持读锁的情况下将写锁降级为读锁，在不需要保持读锁的情况下，释放所有锁。
- 4) 如果写入失败，则调用 removeBlockInternal 方法移除此 Block。
- 5) 如果写入时发生异常，也需要调用 removeBlockInternal 方法移除此 Block。此外，还需要调用 addUpdatedBlockStatusToTaskMetrics 方法更新任务度量信息。

在 doPut 方法中调用了 removeBlockInternal 方法来移除 Block，其实现如代码清单 6-68 所示。

代码清单 6-68 removeBlockInternal 的实现

```

private def removeBlockInternal(blockId: BlockId, tellMaster: Boolean): Unit = {
  val removedFromMemory = memoryStore.remove(blockId)
  val removedFromDisk = diskStore.remove(blockId)
  if (!removedFromMemory && !removedFromDisk) {
    logWarning(s"Block $blockId could not be removed as it was not found on disk
      or in memory")
  }
  blockInfoManager.removeBlock(blockId)
  if (tellMaster) {
    reportBlockStatus(blockId, BlockStatus.empty)
  }
}

```

根据代码清单 6-68，removeBlockInternal 的执行步骤如下。

- 1) 从 MemoryStore 中移除 Block。
- 2) 从 DiskStore 中移除 Block。

- 3) 从 BlockInfoManager 中移除 Block 对应的 BlockInfo。
- 4) 如果需要向 BlockManagerMaster 汇报 Block 状态，则调用 reportBlockStatus 方法（见代码清单 6-66）。

了解了 doPut，现在来看看 putBytes 的实现。putBytes 用于写入 Block 数据，其实现如代码清单 6-69 所示。

代码清单 6-69 putBytes 的实现

```

def putBytes[T: ClassTag](
    blockId: BlockId,
    bytes: ChunkedByteBuffer,
    level: StorageLevel,
    tellMaster: Boolean = true,
    encrypt: Boolean = false): Boolean = {
  require(bytes != null, "Bytes is null")

  val bytesToStore =
    if (encrypt && securityManager.ioEncryptionKey.isDefined) {
      try {
        val data = bytes.toByteBuffer
        val in = new ByteBufferInputStream(data, true)
        val byteBufOut = new ByteBufferOutputStream(data.remaining())
        val out = CryptoStreamUtils.createCryptoOutputStream(byteBufOut, conf,
          securityManager.ioEncryptionKey.get)
        try {
          ByteStreams.copy(in, out)
        } finally {
          in.close()
          out.close()
        }
        new ChunkedByteBuffer(byteBufOut.toByteBuffer)
      } finally {
        bytes.dispose()
      }
    } else {
      bytes
    }
  doPutBytes(blockId, bytesToStore, level, implicitly[ClassTag[T]], tellMaster)
}

```

根据代码清单 6-69，putBytes 首先对 Block 数据进行加密处理，然后调用 doPutBytes 方法写入 Block 数据。doPutBytes 的实现如下。

```

private def doPutBytes[T] (
    blockId: BlockId,
    bytes: ChunkedByteBuffer,
    level: StorageLevel,
    classTag: ClassTag[T],
    tellMaster: Boolean = true,
    keepReadLock: Boolean = false): Boolean = {
  doPut(blockId, level, classTag, tellMaster = tellMaster, keepReadLock =
    keepReadLock) { info =>

```

```

val startTimeMs = System.currentTimeMillis
val replicationFuture = if (level.replication > 1) {
    Future { // 创建异步线程通过调用replicate方法复制Block数据到其他节点的存储体系中
        replicate(blockId, bytes, level, classTag)
    }(futureExecutionContext)
} else {
    null
}
val size = bytes.size
if (level.useMemory) { // 优先写入内存
    val putSucceeded = if (level.deserialized) {
        val values =
            serializerManager.dataDeserializeStream(blockId, bytes.toInputStream())
            (classTag)
        memoryStore.putIteratorAsValues(blockId, values, classTag) match {
            case Right(_) => true
            case Left(iter) =>
                iter.close()
                false
        }
    } else {
        memoryStore.putBytes(blockId, size, level.memoryMode, () => bytes)
    }
    if (!putSucceeded && level.useDisk) { // 内存不足，写入磁盘
        logWarning(s"Persisting block $blockId to disk instead.")
        diskStore.putBytes(blockId, bytes)
    }
} else if (level.useDisk) { // 不能使用内存时，写入磁盘
    diskStore.putBytes(blockId, bytes)
}
val putBlockStatus = getCurrentBlockStatus(blockId, info)
val blockWasSuccessfullyStored = putBlockStatus.storageLevel.isValid
if (blockWasSuccessfullyStored) {
    info.size = size
    if (tellMaster && info.tellMaster) {
        reportBlockStatus(blockId, putBlockStatus) // 向BlockManagerMaster报告
        Block的状态
    }
    addUpdatedBlockStatusToTaskMetrics(blockId, putBlockStatus)
}
logDebug("Put block %s locally took %s".format(blockId, Utils.getUsedTimeMs
    (startTimeMs)))
if (level.replication > 1) {
    // 等待异步的复制线程完成
    try {
        Await.ready(replicationFuture, Duration.Inf)
    } catch {
        case NonFatal(t) =>
            throw new Exception("Error occurred while waiting for replication to
                finish", t)
    }
}
if (blockWasSuccessfullyStored) {
    None
} else {
    Some(bytes)
}

```

```

    }
}.isEmpty
}

```

根据 doPutBytes 的实现，其首先定义了偏函数，这个偏函数将作为 doPut 的 putBody 参数，然后调用 doPut 方法，doPut 方法将调用此偏函数，偏函数写入数据的步骤如下。

- 1) 如果 Block 的 StorageLevel 的复制数量大于 1，则创建异步线程通过调用 replicate 方法复制 Block 数据到其他节点的存储体系中。

- 2) 如果 Block 的 StorageLevel 允许数据写入内存，首先写入内存。如果内存不足且 Block 的 StorageLevel 允许数据写入磁盘，则写入磁盘。

- 3) 如果 Block 的 StorageLevel 允许数据写入磁盘，则写入磁盘。

- 4) 调用 getCurrentBlockStatus 方法（见代码清单 6-61）获取当前 Block 的状态。如果此状态说明 Block 数据成功存储到存储体系，那么调用 reportBlockStatus 方法（见代码清单 6-66）向 BlockManagerMaster 报告 Block 的状态，还调用 addUpdatedBlockStatusToTaskMetrics 方法更新任务度量信息。

5. putBlockData

此方法用于将 Block 数据写入本地，其实现如代码清单 6-70 所示。

代码清单6-70 将Block数据写入本地

```

override def putBlockData(
  blockId: BlockId,
  data: ManagedBuffer,
  level: StorageLevel,
  classTag: ClassTag[_]): Boolean = {
  putBytes(blockId, new ChunkedByteBuffer(data.nioByteBuffer()), level)(classTag)
}

```

根据代码清单 6-70，putBlockData 实际调用了 putBytes。

6. getStatus

此方法用于获取 Block 的状态，其实现如代码清单 6-71 所示。

代码清单6-71 获取Block的状态

```

def getStatus(blockId: BlockId): Option[BlockStatus] = {
  blockInfoManager.get(blockId).map { info =>
    val memSize = if (memoryStore.contains(blockId)) memoryStore.getSize(blockId) else 0L
    val diskSize = if (diskStore.contains(blockId)) diskStore.getSize(blockId) else 0L
    BlockStatus(info.level, memSize = memSize, diskSize = diskSize)
  }
}

```

7. getMatchingBlockIds

此方法用于获取匹配过滤器条件的 BlockId 的序列，其实现如代码清单 6-72 所示。

代码清单6-72 getMatchingBlockIds的实现

```
def getMatchingBlockIds(filter: BlockId => Boolean): Seq[BlockId] = {
    (blockInfoManager.entries.map(_.value) ++ diskBlockManager.getAllBlocks())
        .filter(filter)
        .toArray
        .toSeq
}
```

根据代码清单 6-72，我们看到除了从 BlockInfoManager 的 entries 缓存中获取 BlockId 外，还需要从 DiskBlockManager 中获取。这是因为 DiskBlockManager 中可能存在 BlockInfoManager 不知道的 Block。

8. getLocalValues

此方法用于从本地的 BlockManager 中获取 Block 数据，其实现如代码清单 6-73 所示。

代码清单6-73 从本地的BlockManager中获取Block数据

```
def getLocalValues(blockId: BlockId): Option[BlockResult] = {
    logDebug(s"Getting local block $blockId")
    blockInfoManager.lockForReading(blockId) match { // 获取BlockId所对应的读锁
        case None =>
            logDebug(s"Block $blockId was not found")
            None
        case Some(info) =>
            val level = info.level
            logDebug(s"Level for block $blockId is $level")
            if (level.useMemory && memoryStore.contains(blockId)) {
                //优先从MemoryStore中读取Block数据
                val iter: Iterator[Any] = if (level.deserialized) {
                    memoryStore.getValues(blockId).get
                } else {
                    serializerManager.dataDeserializeStream(
                        blockId, memoryStore.getBytes(blockId).get.toInputStream())(info.classTag)
                }
                val ci = CompletionIterator[Any, Iterator[Any]](iter, releaseLock(blockId))
                Some(new BlockResult(ci, DataReadMethod.Memory, info.size))
            } else if (level.useDisk && diskStore.contains(blockId)) {
                //从DiskStore中读取Block数据
                val iterToReturn: Iterator[Any] = {
                    val diskBytes = diskStore.getBytes(blockId)
                    if (level.deserialized) {
                        val diskValues = serializerManager.dataDeserializeStream(
                            blockId,
                            diskBytes.toInputStream(dispose = true))(info.classTag)
                        maybeCacheDiskValuesInMemory(info, blockId, level, diskValues)
                    } else {
                        val stream = maybeCacheDiskBytesInMemory(info, blockId, level,
                            diskBytes)
                        .map(_.toInputStream(dispose = false))
                        .getOrElse { diskBytes.toInputStream(dispose = true) }
                        serializerManager.dataDeserializeStream(blockId, stream)(info.classTag)
                    }
                }
            }
    }
}
```

```
    val ci = CompletionIterator[Any, Iterator[Any]](iterToReturn, releaseLock(blockId))
    Some(new BlockResult(ci, DataReadMethod.Disk, info.size))
} else {
    handleLocalReadFailure(blockId)
}
}
```

根据代码清单 6-73，getLocalValues 的执行步骤如下。

- 1) 获取 BlockId 所对应的读锁。
 - 2) 优先从 MemoryStore 中读取 Block 数据。
 - 3) 从 DiskStore 中读取 Block 数据。

9. getRemoteBytes

`getRemoteBytes` 方法的作用为从远端的 `BlockManager` 以序列化的字节形式获取 `Block` 数据。但在此之前，首先介绍获取 `Block` 位置信息的方法 `getLocations`，其实现如代码清单 6-74 所示。

代码清单6-74 getLocations的实现

```
private def getLocations(blockId: BlockId): Seq[BlockManagerId] = {
    val locs = Random.shuffle(master.getLocations(blockId))
    val (preferredLocs, otherLocs) = locs.partition { loc => blockManagerId.host ===
        loc.host }
    preferredLocs ++ otherLocs
}
```

根据代码清单 6-74，getLocations 方法的执行步骤如下。

- 1) 调用 BlockManagerMaster 的 getLocations 方法获取所需 Block 所有的所有位置信息(即 BlockManagerId)序列，并随机打乱。
 - 2) 将 BlockManagerId 序列划分为 preferredLocs 与 otherLocs。preferredLocs 中的 Block ManagerId 所标识的 BlockManager 与当前 BlockManager 位于同一机器上，而 otherLocs 中的 BlockManagerId 所标识的 BlockManager 与当前 BlockManager 位于不同机器上。
 - 3) 将 preferredLocs 中的 BlockManagerId 放置在 otherLocs 中的 BlockManagerId 前面，构成一个新的序列返回。这一步骤涉及 Block 的本地性选择。

有了对 `getLocations` 方法的了解，现在来看看 `getRemoteBytes` 的实现，如代码清单 6-75 所示。

代码清单6-75 getRemoteBytes的实现

```
def getRemoteBytes(blockId: BlockId): Option[ChunkedByteBuffer] = {
    logDebug(s"Getting remote block $blockId")
    require(blockId != null, "BlockId is null")
    var runningFailureCount = 0
    var totalFailureCount = 0
    val locations = getLocations(blockId) // 获取Block所在的所有位置信息
```

```

val maxFetchFailures = locations.size
var locationIterator = locations.iterator
while (locationIterator.hasNext) {
    val loc = locationIterator.next()
    logDebug(s"Getting remote block $blockId from $loc")
    val data = try { // 以同步方式从远端下载Block
        blockTransferService.fetchBlockSync(
            loc.host, loc.port, loc.executorId, blockId.toString).nioByteBuffer()
    } catch {
        case NonFatal(e) =>
            runningFailureCount += 1
            totalFailureCount += 1
            if (totalFailureCount >= maxFetchFailures) { // 已经做了最大努力，但还是没能
                下载成功
            logWarning(s"Failed to fetch block after $totalFailureCount fetch failures. " +
                s"Most recent failure cause: ", e)
            return None
    }
    // 刷新Block所在的所有位置信息
    if (runningFailureCount >= maxFailuresBeforeLocationRefresh) {
        locationIterator = getLocations(blockId).iterator
        logDebug(s"Refreshed locations from the driver " +
            s"after ${runningFailureCount} fetch failures.")
        runningFailureCount = 0
    }
    null
}
if (data != null) {
    return Some(new ChunkedByteBuffer(data))
}
logDebug(s"The value of block $blockId is null")
}
logDebug(s"Block $blockId not found")
None
}

```

根据代码清单 6-75，getRemoteBytes 的执行步骤如下。

- 1) 调用 getLocations 方法获取 Block 所有的所有位置信息序列 locations。
- 2) 设置 maxFetchFailures 等于 locations 的大小（即最大获取失败次数）。
- 3) 从 locations 序列中按顺序取出下一个 BlockManagerId，并调用 BlockTransferService 的 fetchBlockSync 方法（见代码清单 6-109），以同步方式从远端下载 Block。
- 4) 如果调用 fetchBlockSync 方法时发生了异常，则增加下载失败次数（runningFailureCount）和下载失败总数（totalFailureCount）。当 totalFailureCount 大于等于 maxFetchFailures 时，说明已经作了最大努力。当 runningFailureCount 大于等于 maxFailuresBeforeLocationRefresh 时，则会重新调用 getLocations 方法刷新 Block 所有的所有位置信息，并将 runningFailureCount 清零。
- 5) 如果第 3) 步获取到数据，那么将得到的数据封装为 ChunkedByteBuffer 并返回，否则回到第 3) 步继续执行。

6) 如果没有获取到数据，则返回 None。

10. get

此方法用于优先从本地获取 Block 数据，当本地获取不到所需的 Block 数据，再从远端获取 Block 数据，其实现如代码清单 6-76 所示。

代码清单6-76 获取Block数据

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

11. downgradeLock

此方法用于将当前线程持有的 Block 的写锁降级为读锁，其实现如代码清单 6-77 所示。

代码清单6-77 锁降级

```
def downgradeLock(blockId: BlockId): Unit = {
    blockInfoManager.downgradeLock(blockId)
}
```

downgradeLock 实际代理了 BlockInfoManager 的 downgradeLock 方法。

12. releaseLock

此方法用于当前线程对持有的 Block 的锁进行释放，其实现如代码清单 6-78 所示。

代码清单6-78 释放锁

```
def releaseLock(blockId: BlockId): Unit = {
    blockInfoManager.unlock(blockId)
}
```

releaseLock 实际调用了 BlockInfoManager 的 unlock 方法。

13. registerTask

此方法用于将任务尝试线程注册到 BlockInfoManager，其实现如代码清单 6-79 所示。

代码清单6-79 注册任务

```
def registerTask(taskAttemptId: Long): Unit = {
    blockInfoManager.registerTask(taskAttemptId)
}
```

registerTask 实际代理了 BlockInfoManager 的 registerTask 方法。

14. releaseAllLocksForTask

此方法用于任务尝试线程对持有的所有 Block 的锁进行释放，其实现如代码清单 6-80 所示。

代码清单6-80 releaseAllLocksForTask的实现

```
def releaseAllLocksForTask(taskAttemptId: Long): Seq[BlockId] = {
    blockInfoManager.releaseAllLocksForTask(taskAttemptId)
}
```

releaseAllLocksForTask 实际代理了 BlockInfoManager 的 releaseAllLocksForTask 方法。

15. getOrElseUpdate

getOrElseUpdate 方法用于获取 Block。如果 Block 存在，则获取此 Block 并返回 BlockResult，否则调用 makeIterator 方法计算 Block，并持久化后返回 BlockResult 或 Iterator。getOrElseUpdate 的实现如代码清单 6-81 所示。

代码清单6-81 getOrElseUpdate的实现

```
def getOrElseUpdate[T] (
    blockId: BlockId,
    level: StorageLevel,
    classTag: ClassTag[T],
    makeIterator: () => Iterator[T]): Either[BlockResult, Iterator[T]] = {
    get[T](blockId)(classTag) match { // 从本地或远端的BlockManager获取Block
        case Some(block) =>
            return Left(block)
        case _ =>
            // Need to compute the block.
    }
    doPutIterator(blockId, makeIterator, level, classTag, keepReadLock = true)
    match {
        case None => // Block已经成功存储到内存
            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(iterator) => // Block存储到内存时发生了错误
            Right(iterator)
    }
}
```

根据代码清单 6-81，getOrElseUpdate 的执行步骤如下。

- 1) 从本地或远端的 BlockManager 获取 Block。如果能够获取到 Block，则返回 Left。
- 2) 调用 doPutIterator 方法计算、持久化 Block。doPutIterator 方法的实现与 doPutBytes

十分相似，都定义了计算、持久化 Block 的偏函数，并以此偏函数作为 putBody 参数调用 doPut，所以不再赘述。

3) doPutIterator 方法的返回结果为 None，说明计算得到的 Block 已经成功存储到内存，因此再次读取此 Block。

4) doPutIterator 方法的返回结果匹配 Some，说明计算得到的 Block 存储到内存时发生了错误。

16. putIterator

此方法用于将 Block 数据写入存储体系，其实现如代码清单 6-82 所示。

代码清单6-82 putIterator的实现

```
def putIterator[T: ClassTag] (
    blockId: BlockId,
    values: Iterator[T],
    level: StorageLevel,
    tellMaster: Boolean = true): Boolean = {
  require(values != null, "Values is null")
  doPutIterator(blockId, () -> values, level, implicitly[ClassTag[T]], tellMaster)
  match {
    case None =>
      true
    case Some(iter) =>
      iter.close()
      false
  }
}
```

根据代码清单 6-82，putIterator 内部实际也调用了 doPutIterator 方法。当 doPutIterator 返回 None，说明计算得到的 Block 已经成功存储到内存，因此再次读取此 Block。doPutIterator 方法的返回结果匹配 Some，则说明计算得到的 Block 存储到内存时发生了错误。

17. getDiskWriter

此方法用于创建并获取 DiskBlockObjectWriter，通过 DiskBlockObjectWriter 可以跳过对 DiskStore 的使用，直接将数据写入磁盘。getDiskWriter 的实现如代码清单 6-83 所示。

代码清单6-83 getDiskWriter的实现

```
def getDiskWriter(
    blockId: BlockId,
    file: File,
    serializerInstance: SerializerInstance,
    bufferSize: Int,
    writeMetrics: ShuffleWriteMetrics): DiskBlockObjectWriter = {
  val syncWrites = conf.getBoolean("spark.shuffle.sync", false)
  new DiskBlockObjectWriter(file, serializerManager, serializerInstance, bufferSize,
    syncWrites, writeMetrics, blockId)
}
```

根据代码清单 6-83 我们知道，属性 spark.shuffle.sync 将决定 DiskBlockObjectWriter 把数据写入磁盘时是采用同步方式还是异步方式，默认是异步方式。

18. getSingle

此方法的作用为获取由单个对象组成的 Block，其实现如代码清单 6-84 所示。

代码清单6-84 getSingle的实现

```
def getSingle[T: ClassTag](blockId: BlockId): Option[T] = {
    get[T](blockId).map(_.data.next().asInstanceOf[T])
}
```

根据代码清单 6-84，我们知道 getSingle 其实调用了 get 方法。

19. putSingle

此方法用于将由单个对象组成的 Block 写入存储体系，其实现如代码清单 6-85 所示。

代码清单6-85 putSingle的实现

```
def putSingle[T: ClassTag](
    blockId: BlockId,
    value: T,
    level: StorageLevel,
    tellMaster: Boolean = true): Boolean = {
    putIterator(blockId, Iterator(value), level, tellMaster)
}
```

根据代码清单 6-85，putSingle 实际调用了 putIterator。

20. dropFromMemory

此方法用于从内存中删除 Block，当 Block 的存储级别允许写入磁盘，Block 将被写入磁盘。此方法主要在内存不足，需要从内存腾出空间时使用。dropFromMemory 的实现如代码清单 6-86 所示。

代码清单6-86 dropFromMemory的实现

```
private[storage] override def dropFromMemory[T: ClassTag](
    blockId: BlockId,
    data: () => Either[Array[T], ChunkedByteBuffer]): StorageLevel = {
    logInfo(s"Dropping block $blockId from memory")
    // 确认当前任务尝试线程是否已经持有BlockId对应的写锁
    val info = blockInfoManager.assertBlockIsLockedForWriting(blockId)
    var blockIsUpdated = false
    val level = info.level
    // 将Block写入磁盘
    if (level.useDisk && !diskStore.contains(blockId)) {
        logInfo(s"Writing block $blockId to disk")
        data() match {
            case Left(elements) =>
                diskStore.put(blockId) { fileOutputStream =>
                    serializerManager.dataSerializeStream(
                        blockId,
```

```

        fileOutputStream,
        elements.toIterator) (info.classTag.asInstanceOf[ClassTag[T]])
    }
    case Right(bytes) =>
      diskStore.putBytes(blockId, bytes)
}
blockIsUpdated = true
}
// 将内存中的Block删除
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, status, droppedMemorySize) // 向BlockManagerMaster
  报告Block状态
}
if (blockIsUpdated) {
  addUpdatedBlockStatusToTaskMetrics(blockId, status) // 更新任务度量信息
}
status.storageLevel // 返回Block的存储级别
}

```

根据代码清单 6-86，dropFromMemory 的执行步骤如下。

- 1) 确认当前任务尝试线程是否已经持有 BlockId 对应的写锁。
- 2) 如果 Block 对应的存储级别允许 Block 使用磁盘，并且 Block 尚未写入磁盘，则调用 DiskStore 的 put 方法或 putBytes 方法将 Block 写入磁盘。
- 3) 如果 MemoryStore 中存在 Block，则调用 MemoryStore 的 getSize 方法获取将要从内存中删除的 Block 的大小 droppedMemorySize。
- 4) 调用 MemoryStore 的 remove 方法将内存中的 Block 删除。
- 5) 调用 getCurrentBlockStatus 方法（见代码清单 6-61）获取 Block 的当前状态。
- 6) 如果 BlockInfo 的 tellMaster 属性为 true，则调用 reportBlockStatus 方法（见代码清单 6-66）向 BlockManagerMaster 报告 Block 状态。
- 7) 当 Block 写入了磁盘或 Block 从内存中删除，则调用 addUpdatedBlockStatusToTask Metrics 方法更新任务度量信息。
- 8) 返回 Block 的存储级别。

21. removeBlock

此方法的作用为删除指定的 Block，其实现如代码清单 6-87 所示。

代码清单6-87 removeBlock的实现

```
def removeBlock(blockId: BlockId, tellMaster: Boolean = true): Unit = {
    logDebug(s"Removing block $blockId")
    blockInfoManager.lockForWriting(blockId) match { // 获取指定Block的写锁
        case None =>
            // The block has already been removed; do nothing.
            logWarning(s"Asked to remove block $blockId, which does not exist")
        case Some(info) => // 从存储体系中删除Block
            removeBlockInternal(blockId, tellMaster = tellMaster && info.tellMaster)
            addUpdatedBlockStatusToTaskMetrics(blockId, BlockStatus.empty)
    }
}
```

根据代码清单 6-87，removeBlock 方法的执行步骤如下。

- 1) 获取指定 Block 的写锁。
- 2) 如果获取到写锁，则调用 removeBlockInternal 方法（见代码清单 6-68）从存储体系中删除 Block。

22. removeRdd

此方法的作用为移除属于指定 RDD 的所有 Block，其实现如代码清单 6-88 所示。

代码清单6-88 removeRdd的实现

```
def removeRdd(rddId: Int): Int = {
    logInfo(s"Removing RDD $rddId")
    val blocksToRemove = blockInfoManager.entries.flatMap(_.asRDDId).filter(_.
        rddId == rddId)
    blocksToRemove.foreach { blockId => removeBlock(blockId, tellMaster = false) }
    blocksToRemove.size
}
```

根据代码清单 6-88，removeRdd 的执行步骤如下。

- 1) 从 BlockInfoManager 的 entries 中找出所有的 RDDBlockId，并过滤出其 rddId 属性等于指定 rddId 的所有 RDDBlockId。
- 2) 调用 removeBlock 方法删除过滤出来的所有 RDDBlockId。

23. removeBroadcast

此方法的作用为移除属于指定 Broadcast 的所有 Block，其实现如代码清单 6-89 所示。

代码清单6-89 removeBroadcast的实现

```
def removeBroadcast(broadcastId: Long, tellMaster: Boolean): Int = {
    logDebug(s"Removing broadcast $broadcastId")
    val blocksToRemove = blockInfoManager.entries.map(_.asBroadcastId).collect {
        case bid @ BroadcastBlockId(`broadcastId`, _) => bid
    }
    blocksToRemove.foreach { blockId => removeBlock(blockId, tellMaster) }
    blocksToRemove.size
}
```

根据代码清单 6-89，可以看出 removeBroadcast 与 removeRdd 的实现相似。

6.8 BlockManagerMaster 对 BlockManager 的管理

BlockManagerMaster 的作用是对存在于 Executor 或 Driver 上的 BlockManager 进行统一管理。Executor 与 Driver 关于 BlockManager 的交互都依赖于 BlockManagerMaster，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？

根据 5.7 节的内容，我们知道 Driver 上的 BlockManagerMaster 会实例化并且注册 BlockManagerMasterEndpoint。无论是 Driver 还是 Executor，它们的 BlockManagerMaster 的 driverEndpoint 属性都将持有 BlockManagerMasterEndpoint 的 RpcEndpointRef。无论是 Driver 还是 Executor，每个 BlockManager 都拥有自己的 BlockManagerSlaveEndpoint，且 BlockManager 的 slaveEndpoint 属性保存着各自 BlockManagerSlaveEndpoint 的 RpcEndpointRef。BlockManagerMaster 负责发送消息，BlockManagerMasterEndpoint 负责消息的接收与处理，BlockManagerSlaveEndpoint 则接收 BlockManagerMasterEndpoint 下发的命令。图 6-1 中也展现了这几者之间的关系。

6.8.1 BlockManagerMaster 的职责

BlockManagerMaster 负责发送各种与存储体系相关的信息，这些消息的类型如下。

- ❑ RemoveExecutor (移除 Executor)。
- ❑ RegisterBlockManager (注册 BlockManager)。
- ❑ UpdateBlockInfo (更新 Block 信息)。
- ❑ GetLocations (获取 Block 的位置)。
- ❑ GetLocationsMultipleBlockIds (获取多个 Block 的位置)。
- ❑ GetPeers (获取其他 BlockManager 的 BlockManagerId)。
- ❑ GetExecutorEndpointRef (获取 Executor 的 EndpointRef 引用)。
- ❑ RemoveBlock (移除 Block)。
- ❑ RemoveRdd (移除 Rdd Block)。
- ❑ RemoveShuffle (移除 Shuffle Block)。
- ❑ RemoveBroadcast (移除 Broadcast Block)。
- ❑ GetMemoryStatus (获取指定的 BlockManager 的内存状态)。
- ❑ GetStorageStatus (获取存储状态)。
- ❑ GetBlockStatus (获取 Block 的状态)。
- ❑ GetMatchingBlockIds (获取匹配过滤条件的 Block)。

- HasCachedBlocks（指定的 Executor 上是否有缓存的 Block）。
- StopBlockManagerMaster（停止 BlockManagerMaster）。

可以看到，BlockManagerMaster 能够发送的消息类型多种多样，为了更容易理解，本节以 RegisterBlockManager 为例，来介绍 BlockManagerMaster 发挥的作用。BlockManagerMaster 提供的 registerBlockManager 方法负责发送 RegisterBlockManager 消息，其实现如代码清单 6-90 所示。

代码清单 6-90 注册 BlockManager

```
def registerBlockManager(
    blockManagerId: BlockManagerId,
    maxMemSize: Long,
    slaveEndpoint: RpcEndpointRef): BlockManagerId = {
  logInfo(s"Registering BlockManager $blockManagerId")
  val updatedId = driverEndpoint.askWithRetry[BlockManagerId] (
    RegisterBlockManager(blockManagerId, maxMemSize, slaveEndpoint))
  logInfo(s"Registered BlockManager $updatedId")
  updatedId
}
```

根据代码清单 6-90，注册 BlockManager 的实质是向 BlockManagerMasterEndpoint 发送 RegisterBlockManager 消息，RegisterBlockManager 将携带要注册的 BlockManager 的 blockManagerId、最大内存大小及 slaveEndpoint（即 BlockManagerSlaveEndpoint）。

BlockManagerMaster 对于其他类型的消息也提供了对应的方法实现，实现过程都与 registerBlockManager 方法类似，留给感兴趣的读者自行阅读。

6.8.2 BlockManagerMasterEndpoint 详解

BlockManagerMasterEndpoint 接收 Driver 或 Executor 上 BlockManagerMaster 发送的消息，对所有的 BlockManager 统一管理。BlockManagerMasterEndpoint 定义了一些管理 BlockManager 的属性，这些属性的作用如下。

- blockManagerInfo：BlockManagerId 与 BlockManagerInfo 之间映射关系的缓存。
- blockManagerIdByExecutor：Executor ID 与 BlockManagerId 之间映射关系的缓存。
- blockLocations：BlockId 与存储了此 BlockId 对应 Block 的 BlockManager 的 BlockManagerId 之间的一对多关系缓存。
- topologyMapper：对集群所有节点的拓扑结构的映射。

BlockManagerMasterEndpoint 重写了特质 RpcEndpoint 的 receiveAndReply 方法，用于接收 BlockManager 相关的消息，其实现如代码清单 6-91 所示。

代码清单 6-91 receiveAndReply 的实现

```
override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
  case RegisterBlockManager(blockManagerId, maxMemSize, slaveEndpoint) =>
    context.reply(register(blockManagerId, maxMemSize, slaveEndpoint))
```

```

case _updateBlockInfo @
  UpdateBlockInfo(blockManagerId, blockId, storageLevel, deserializedSize,
    size) =>
  context.reply(updateBlockInfo(blockManagerId, blockId, storageLevel,
    deserializedSize, size))
  listenerBus.post(SparkListenerBlockUpdated(BlockUpdatedInfo(_updateBlockInfo)))
case GetLocations(blockId) =>
  context.reply(getLocations(blockId))
case GetLocationsMultipleBlockIds(blockIds) =>
  context.reply(getLocationsMultipleBlockIds(blockIds))
case GetPeers(blockManagerId) =>
  context.reply(getPeers(blockManagerId))
case GetExecutorEndpointRef(executorId) =>
  context.reply(getExecutorEndpointRef(executorId))
case GetMemoryStatus =>
  context.reply(memoryStatus)
case GetStorageStatus =>
  context.reply(storageStatus)
case GetBlockStatus(blockId, askSlaves) =>
  context.reply(blockStatus(blockId, askSlaves))
case GetMatchingBlockIds(filter, askSlaves) =>
  context.reply(getMatchingBlockIds(filter, askSlaves))
case RemoveRdd(rddId) =>
  context.reply(removeRdd(rddId))
case RemoveShuffle(shuffleId) =>
  context.reply(removeShuffle(shuffleId))
case RemoveBroadcast(broadcastId, removeFromDriver) =>
  context.reply(removeBroadcast(broadcastId, removeFromDriver))
case RemoveBlock(blockId) =>
  removeBlockFromWorkers(blockId)
  context.reply(true)
case RemoveExecutor(execId) =>
  removeExecutor(execId)
  context.reply(true)
case StopBlockManagerMaster =>
  context.reply(true)
  stop()
case BlockManagerHeartbeat(blockManagerId) =>
  context.reply(heartbeatReceived(blockManagerId))
case HasCachedBlocks(executorId) =>
  blockManagerIdByExecutor.get(executorId) match {
    case Some(bm) =>
      if (blockManagerInfo.contains(bm)) {
        val bmInfo = blockManagerInfo(bm)
        context.reply(bmInfo.cachedBlocks.nonEmpty)
      } else {
        context.reply(false)
      }
    case None => context.reply(false)
  }
}

```

根据代码清单 6-91，BlockManagerMasterEndpoint 接收的消息类型正好与 BlockManagerMaster 所发送的消息一一对应。本节依然选择 RegisterBlockManager 消息，来介绍

BlockManagerMasterEndpoint 是如何接收和处理 RegisterBlockManager 消息的。根据代码清单 6-91，BlockManagerMasterEndpoint 在接收到 RegisterBlockManager 消息后，将调用 register 方法。register 的实现如代码清单 6-92 所示。

代码清单6-92 register的实现

```

private def register(
    idWithoutTopologyInfo: BlockManagerId,
    maxMemSize: Long,
    slaveEndpoint: RpcEndpointRef): BlockManagerId = {
  val id = BlockManagerId( // 生成BlockManagerId
    idWithoutTopologyInfo.executorId,
    idWithoutTopologyInfo.host,
    idWithoutTopologyInfo.port,
    topologyMapper.getTopologyForHost(idWithoutTopologyInfo.host))

  val time = System.currentTimeMillis()
  if (!blockManagerInfo.contains(id)) {
    blockManagerIdByExecutor.get(id.executorId) match {
      case Some(oldId) =>
        logError("Got two different block manager registrations on same executor - "
          + s" will replace old one $oldId with new one $id")
        removeExecutor(id.executorId) //移除blockManagerIdByExecutor中的缓存信息
      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, slaveEndpoint)
  }
  listenerBus.post(SparkListenerBlockManagerAdded(time, id, maxMemSize))
  id
}

```

根据代码清单 6-92，register 方法的执行步骤如下。

- 1) 生成 BlockManagerId。BlockManagerId 由 executorId (Executor 的标识)、host (块传输服务的 host)、port (块传输服务的 port) 及拓扑信息组成。
- 2) 如果 blockManagerInfo 缓存中不存在此 BlockManagerId，则进入下一步，否则进入第 6) 步。
- 3) 移除 blockManagerIdByExecutor、blockManagerInfo、blockLocations 等缓存中与此 BlockManagerId 有关的所有缓存信息。removeExecutor 方法的实现如代码清单 6-93 所示。其中调用的 removeBlockManager 方法的实现很简单，感兴趣的读者可以自己阅读其实现。

代码清单6-93 removeExecutor的实现

```

private def removeExecutor(execId: String) {
  logInfo("Trying to remove executor " + execId + " from BlockManagerMaster.")
}

```

```

    blockManagerIdByExecutor.get(execId).foreach(removeBlockManager)
}

```

4) 将 executorId 与新创建的 BlockManagerId 的对应关系添加到 blockManagerIdBy-Executor 中。

5) 将 BlockManagerId 与 BlockManagerInfo 的对应关系添加到缓存 blockManagerInfo。

6) 向 listenerBus 投递 SparkListenerBlockManagerAdded 类型的事件。根据 3.3 节的内容, listenerThread 线程最终将触发对所有 SparkListener 的 onBlockManagerAdded 方法的调用, 进而达到监控的目的。

本节以注册 BlockManager 为例, 介绍了 BlockManagerMasterEndpoint 的作用。其他类型的消息处理都是类似的, 故此不再赘述。

6.8.3 BlockManagerSlaveEndpoint 详解

BlockManagerSlaveEndpoint 用于接收 BlockManagerMasterEndpoint 的命令并执行相应的操作。BlockManagerSlaveEndpoint 也重写了 RpcEndpoint 的 receiveAndReply 方法, 其实现如代码清单 6-94 所示。

代码清单 6-94 BlockManagerSlaveEndpoint 的 receiveAndReply 实现

```

override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
  case RemoveBlock(blockId) =>
    doAsync[Boolean]("removing block " + blockId, context) {
      blockManager.removeBlock(blockId)
      true
    }
  case RemoveRdd(rddId) =>
    doAsync[Int]("removing RDD " + rddId, context) {
      blockManager.removeRdd(rddId)
    }
  case RemoveShuffle(shuffleId) =>
    doAsync[Boolean]("removing shuffle " + shuffleId, context) {
      if (mapOutputTracker != null) {
        mapOutputTracker.unregisterShuffle(shuffleId)
      }
      SparkEnv.get.shuffleManager.unregisterShuffle(shuffleId)
    }
  case RemoveBroadcast(broadcastId, _) =>
    doAsync[Int]("removing broadcast " + broadcastId, context) {
      blockManager.removeBroadcast(broadcastId, tellMaster = true)
    }
  case GetBlockStatus(blockId, _) =>
    context.reply(blockManager.getStatus(blockId))
  case GetMatchingBlockIds(filter, _) =>
    context.reply(blockManager.getMatchingBlockIds(filter))
  case TriggerThreadDump =>
    context.reply(Utils.getThreadDump())
}

```

本节以 Driver 删除 Block 为例，来介绍 BlockManagerSlaveEndpoint 的作用。Driver 节点删除 Block 依赖于 BlockManagerMaster 提供的 removeBlock 方法，其实现如代码清单 6-95 所示。

代码清单 6-95 removeBlock 的实现

```
def removeBlock(blockId: BlockId) {
    driverEndpoint.askWithRetry[Boolean](RemoveBlock(blockId))
}
```

根据代码清单 6-95，BlockManagerMaster 提供的 removeBlock 方法实际向 BlockManagerMasterEndpoint 发送 RemoveBlock 类型的消息。根据代码清单 6-91 中 BlockManagerMasterEndpoint 的 receiveAndReply 方法的实现，BlockManagerMasterEndpoint 首先调用 removeBlockFromWorkers，然后向客户端回复 true。removeBlockFromWorkers 方法的实现如代码清单 6-96 所示。

代码清单 6-96 removeBlockFromWorkers 的实现

```
private def removeBlockFromWorkers(blockId: BlockId) {
    val locations = blockLocations.get(blockId)
    if (locations != null) {
        locations.foreach { blockManagerId: BlockManagerId =>
            val blockManager = blockManagerInfo.get(blockManagerId)
            if (blockManager.isDefined) {
                blockManager.get.slaveEndpoint.ask[Boolean](RemoveBlock(blockId))
            }
        }
    }
}
```

根据代码清单 6-96，removeBlockFromWorkers 的执行步骤如下。

- 1) 从缓存 blockLocations 中获取 BlockId 所对应 Block 实际存储的位置集合 locations。
- 2) 遍历 locations，向每个节点的 BlockManagerSlaveEndpoint 发送 RemoveBlock 消息。

根据代码清单 6-94，BlockManagerSlaveEndpoint 接收到 RemoveBlock 消息后，将调用 BlockManager 的 removeBlock 删除 Block。

6.9 Block 传输服务

根据之前的介绍，我们知道 BlockTransferService 是 BlockManager 的子组件之一。抽象类 BlockTransferService 有两个实现：用于测试的 MockBlockTransferService 及 5.7 节曾经介绍的 NettyBlockTransferService。根据 5.7 节的介绍，我们知道 BlockManager 实际采用了 NettyBlockTransferService 提供的 Block 传输服务。

读者可能奇怪为什么要把由 Netty 实现的网络服务组件也放到存储体系里面。由于 Spark 是分布式部署的，每个 Task（准确说是任务尝试）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。NettyBlockTransferService 提供了可以被其他节点的客户端访问的 Shuffle 服务。

有了 Shuffle 的服务端，那么也需要相应的 Shuffle 客户端，以便当前节点将 Block 上传到其他节点或者从其他节点下载 Block 到本地。BlockManager 中创建 Shuffle 客户端的代码如下。

```
private[spark] val shuffleClient = if (externalShuffleServiceEnabled) {
    val transConf = SparkTransportConf.fromSparkConf(conf, "shuffle", numUsableCores)
    new ExternalShuffleClient(transConf, securityManager,
        securityManager.isAuthenticationEnabled(),
        securityManager.isSaslEncryptionEnabled())
} else {
    blockTransferService
}
```

从上面的代码可以知道，如果部署了外部的 Shuffle 服务，则需要配置 spark.shuffle.service.enabled 属性为 true（此属性将决定 externalShuffleServiceEnabled 的值，默认是 false），此时将创建 ExternalShuffleClient。但在默认情况下，NettyBlockTransferService 也会作为 Shuffle 的客户端。

小贴士：熟悉 Hadoop YARN 的用户可能已经发现 Spark 与 Hadoop 一样，都采用 Netty 来实现 Shuffle Server。

6.9.1 初始化 NettyBlockTransferService

NettyBlockTransferService 只有在其 init 方法被调用，即被初始化后才提供服务。根据 6.7.1 节的内容，BlockManager 在初始化的时候，将调用 NettyBlockTransferService 的 init 方法。NettyBlockTransferService 的 init 方法的实现如代码清单 6-97 所示。

代码清单 6-97 初始化 NettyBlockTransferService

```
override def init(blockDataManager: BlockDataManager): Unit = {
    val rpcHandler = new NettyBlockRpcServer(conf.getAppId, serializer,
        blockDataManager)
    var serverBootstrap: Option[TransportServerBootstrap] = None
    var clientBootstrap: Option[TransportClientBootstrap] = None
    if (authEnabled) {
        serverBootstrap = Some(new SaslServerBootstrap(transportConf, securityManager))
        clientBootstrap = Some(new SaslClientBootstrap(transportConf, conf.getAppId,
            securityManager,
            securityManager.isSaslEncryptionEnabled()))
    }
    transportContext = new TransportContext(transportConf, rpcHandler)
    clientFactory = transportContext.createClientFactory(clientBootstrap.toSeq.asJava)
    server = createServer(serverBootstrap.toList)
    appId = conf.getAppId
    logInfo(s"Server created on ${hostName}:${server.getPort}")
}
```

根据代码清单 6-97，NettyBlockTransferService 的初始化步骤如下。

1) 创建 NettyBlockRpcServer。NettyBlockRpcServer 继承了我们熟悉的 RpcHandler，根据 3.2 节的内容，我们知道服务端对客户端的 Block 读写请求的处理都交给了 RpcHandler 的实现类，因此 NettyBlockRpcServer 将处理 Block 块的 RPC 请求。

2) 准备客户端引导程序 TransportClientBootstrap 和服务端引导程序 TransportServerBootstrap。

3) 创建 TransportContext。

4) 创建传输客户端工厂 TransportClientFactory。

5) 创建 TransportServer。

6) 获取当前应用的 ID。

由于 NettyBlockTransferService 初始化的过程，与 5.3 节 RPC 环境的创建过程非常相似，所以不再赘述。

6.9.2 NettyBlockRpcServer 详解

由于 RPC 框架的内容已在 3.2 节有详细介绍，所以我们重点来看看 NettyBlockTransferService 与 NettyRpcEnv 的最大区别——使用的 RpcHandler 的实现类不同，NettyRpcEnv 采用了 NettyRpcHandler，而 NettyBlockTransferService 则采用了 NettyBlockRpcServer。

1. OneForOneStreamManager 的实现

NettyBlockRpcServer 中使用了 OneForOneStreamManager 来提供一对一的流服务。OneForOneStreamManager 实现了 StreamManager 的 registerChannel、getChunk、connectionTerminated、checkAuthorization、registerStream 五个方法，所以根据 3.2.6 节对 TransportRequestHandler 的介绍，我们知道 TransportRequestHandler 的 processFetchRequest 方法用到了 StreamManager 的 checkAuthorization、registerChannel 和 getChunk 三个方法，OneForOneStreamManager 将处理 ChunkFetchRequest 类型的消息。

OneForOneStreamManager 中使用 StreamState 来维护流的状态，所以我们需要先认识 StreamState，StreamState 的实现如代码清单 6-98 所示。

代码清单6-98 StreamState的实现

```
private static class StreamState {
    final String appId;
    final Iterator<ManagedBuffer> buffers;

    Channel associatedChannel = null;
    int curChunk = 0;
    StreamState(String appId, Iterator<ManagedBuffer> buffers) {
        this.appId = appId;
        this.buffers = Preconditions.checkNotNull(buffers);
    }
}
```

根据代码清单 6-98，StreamState 包含以下属性。

- 1) appId：请求流所属的应用程序 ID。此属性只有在 ExternalShuffleClient 启用后才会用到。
- 2) buffers：ManagedBuffer 的缓冲。
- 3) associatedChannel：与当前流相关联的 Channel。
- 4) curChunk：为了保证客户端按顺序每次请求一个块，所以用此属性跟踪客户端当前接收到的 ManagedBuffer 的索引。

有了对 StreamState 的了解，我们来看看 OneForOneStreamManager 的成员属性。

- 5) nextStreamId：用于生成数据流的标识，类型为 AtomicLong。
 - 6) streams：维护 streamId 与 StreamState 之间映射关系的缓存。
- 认识了以上介绍的属性，现在来看看 OneForOneStreamManager 中实现的方法。
- 7) checkAuthorization 方法（见代码清单 6-99）用以校验客户端是否有权限从给定的流中读取。

代码清单 6-99 checkAuthorization 的实现

```

@Override
public void checkAuthorization(TransportClient client, long streamId) {
    if (client.getClientId() != null) {
        StreamState state = streams.get(streamId);
        Preconditions.checkNotNull(state, "Unknown stream ID.");
        if (!client.getClientId().equals(state.appId)) {
            throw new SecurityException(String.format(
                "Client %s not authorized to read stream %d (app %s).",
                client.getClientId(),
                streamId,
                state.appId));
        }
    }
}

```

如果没有配置对管道进行 SASL 认证（参见 3.2.6 节代码清单 3-35 服务端的 SaslServerBootstrap 对管道进行 SASL 认证的内容），TransportClient 的 clientId 为 null，因而实际上并不走权限检查。当启用了 SASL 认证，客户端需要给 TransportClient 的 clientId 赋值，因此才会走此检查。checkAuthorization 方法检查的办法很简单，即将 TransportClient 的 clientId 属性值与 streamId 对应的 StreamState 的 appId 的值进行相等比较。

8) registerChannel 方法（见代码清单 6-100）用于注册管道，其实际的作用是将一个流和一条（只能是一条）客户端的 TCP 连接关联起来，这可以保证对于单个的流只会有一个客户端读取。流关闭之后就永远不能够重用了。

代码清单 6-100 registerChannel 的实现

```

@Override
public void registerChannel(Channel channel, long streamId) {

```

```

        if (streams.containsKey(streamId)) {
            streams.get(streamId).associatedChannel = channel;
        }
    }
}

```

9) getChunk 方法（见代码清单 6-101）用于获取单个的块（块被封装为 ManagedBuffer）。

代码清单6-101 getChunk的实现

```

@Override
public ManagedBuffer getChunk(long streamId, int chunkIndex) {
    StreamState state = streams.get(streamId); // 从streams中获取StreamState
    if (chunkIndex != state.curChunk) {
        throw new IllegalStateException(String.format(
            "Received out-of-order chunk index %s (expected %s)", chunkIndex, state.
            curChunk));
    } else if (!state.buffers.hasNext()) {
        throw new IllegalStateException(String.format(
            "Requested chunk index beyond end %s", chunkIndex));
    }
    state.curChunk += 1; // 将StreamState的curChunk加1，为下次接收请求做好准备
    ManagedBuffer nextChunk = state.buffers.next(); // 从buffers缓冲中获取ManagedBuffer
    if (!state.buffers.hasNext()) { // buffers缓冲中的ManagedBuffer，已经全部被客户端获取
        logger.trace("Removing stream id {}", streamId);
        streams.remove(streamId);
    }
    return nextChunk;
}

```

根据代码清单 6-101，getChunk 的执行步骤如下。

① 从 streams 中获取 StreamState。如果要获取的块的索引与 StreamState 的 curChunk 属性不相等，则说明顺序有问题。如果要获取的块的索引超出了 buffers 缓冲的大小，这说明请求了一个超出范围的块。

② 将 StreamState 的 curChunk 加 1，为下次接收请求做好准备。

③ 从 buffers 缓冲中获取 ManagedBuffer。如果 buffers 缓冲已经迭代到了末端，那么说明当前流的块已经全部被客户端获取了，需要将 streamId 与对应的 StreamState 从 streams 中移除。

④ 返回获取的 ManagedBuffer。

10) registerStream 方法（见代码清单 6-102）用于向 OneForOneStreamManager 的 streams 缓存中注册流。

代码清单6-102 registerStream的实现

```

public long registerStream(String appId, Iterator<ManagedBuffer> buffers) {
    long myStreamId = nextStreamId.getAndIncrement();
    streams.put(myStreamId, new StreamState(appId, buffers));
    return myStreamId;
}

```

根据代码清单 6-102, registerStream 方法首先生成一个新的 streamId, 然后创建 StreamState 对象, 最后将 streamId 与 StreamState 对象之间的映射关系放入 streams 中。

2. NettyBlockRpcServer 的实现

介绍了 NettyBlockRpcServer 的内部组件 OneForOneStreamManager, 下面来看看 NettyBlockRpcServer 的实现, 如代码清单 6-103 所示。

代码清单6-103 NettyBlockRpcServer的实现

```

class NettyBlockRpcServer(
    appId: String,
    serializer: Serializer,
    blockManager: BlockDataManager)
  extends RpcHandler with Logging {

  private val streamManager = new OneForOneStreamManager()

  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 => // 打开(读取)Block
        val blocks: Seq[ManagedBuffer] =
          openBlocks.blockIds.map(BlockId.apply).map(blockManager.getBlockData)
        val streamId = streamManager.registerStream(appId, blocks.iterator.asJava)
        logTrace(s"Registered streamId $streamId with ${blocks.size} buffers")
        responseContext.onSuccess(new StreamHandle(streamId, blocks.size).toByteBuffer)

      case uploadBlock: UploadBlock => // 上传Block
        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))
    }
}

override def getStreamManager(): StreamManager = streamManager
}

```

根据代码清单 6-103, NettyBlockRpcServer 实现了需要回复客户端的 receive 和 getStreamManager 两个方法, 其中 getStreamManager 方法将返回 OneForOneStreamManager。NettyBlockRpcServer 的 receive 方法将分别接收以下两种消息。

1) OpenBlocks：打开（读取）Block。NettyBlockRpcServer 对这种消息的处理步骤如下。

- ① 取出 OpenBlocks 消息携带的 BlockId 数组。
- ② 调用 BlockManager 的 getBlockData 方法，获取数组中每一个 BlockId 对应的 Block（返回值为 ManagedBuffer 的序列）。
- ③ 调用 OneForOneStreamManager 的 registerStream 方法，将 ManagedBuffer 序列注册到 OneForOneStreamManager 的 streams 缓存。
- ④ 创建 StreamHandle 消息（包含 streamId 和 ManagedBuffer 序列的大小），并通过响应上下文回复客户端。

2) UploadBlock：上传 Block。NettyBlockRpcServer 对这种消息的处理步骤如下。

- ① 对 UploadBlock 消息携带的元数据 metadata 进行反序列化，得到存储级别（StorageLevel）和类型标记（上传 Block 的类型）。
- ② 将 UploadBlock 消息携带的 Block 数据（即 blockData），封装为 NioManagedBuffer。
- ③ 获取 UploadBlock 消息携带的 BlockId。
- ④ 调用 BlockManager 的 putBlockData 方法，将 Block 存入本地存储体系。
- ⑤ 通过响应上下文回复客户端。

6.9.3 Shuffle 客户端

根据前面的介绍，如果没有部署外部的 Shuffle 服务，即 spark.shuffle.service.enabled 属性为 false 时，NettyBlockTransferService 不但通过 OneForOneStreamManager 与 NettyBlockRpcServer 对外提供 Block 上传与下载的服务，也将作为默认的 Shuffle 客户端。NettyBlockTransferService 作为 Shuffle 客户端，具有发起上传和下载请求并接收服务端响应的能力。NettyBlockTransferService 的两个方法——fetchBlocks 和 uploadBlock 将帮我们达到目的。

1. 发送下载远端 Block 的请求

NettyBlockTransferService 的 fetchBlocks 方法的实现如代码清单 6-104 所示。

代码清单 6-104 fetchBlocks 的实现

```
override def fetchBlocks(
    host: String,
    port: Int,
    execId: String,
    blockIds: Array[String],
    listener: BlockFetchingListener): Unit = {
  logTrace(s"Fetch blocks from $host:$port (executor id $execId)")
  try {
    val blockFetchStarter = new RetryingBlockFetcher.BlockFetchStarter {
      override def createAndStart(blockIds: Array[String], listener:
        BlockFetchingListener) {
        val client = clientFactory.createClient(host, port)
        ...
      }
    }
    blockFetchStarter.start(blockIds, listener)
  } catch {
    case e: Exception =>
      logError(s"Failed to fetch blocks from $host:$port due to ${e.getMessage}")
      listener.onFailure(e)
  }
}
```

```

        new OneForOneBlockFetcher(client, appId, execId, blockIds.toArray, listener).start()
    }
}

val maxRetries = transportConf.maxIORetries()
if (maxRetries > 0) { // 创建Block的重试线程
    new RetryingBlockFetcher(transportConf, blockFetchStarter, blockIds, listener).start()
} else {
    blockFetchStarter.createAndStart(blockIds, listener)
}
} catch {
    case e: Exception =>
    logError("Exception while beginning fetchBlocks", e)
    blockIds.foreach(listener.onBlockFetchFailure(_, e))
}
}

```

根据代码清单 6-104, fetchBlocks 的执行步骤如下。

- 1) 创建 RetryingBlockFetcher.BlockFetchStarter 的匿名实现类的实例 blockFetchStarter, 此匿名类实现了 BlockFetchStarter 接口的 createAndStart 方法。
- 2) 获取 spark.\$module.io.maxRetries 属性 (NettyBlockTransferService 的 module 为 shuffle) 的值作为下载请求的最大重试次数 maxRetries。
- 3) 只有配置的 spark.shuffle.io.maxRetries 属性大于 0, maxRetries 才有效, 此时将创建 RetryingBlockFetcher 并调用 RetryingBlockFetcher 的 start 方法, 否则直接调用 blockFetchStarter 的 createAndStart 方法。

RetryingBlockFetcher 的 start 方法只是调用了 fetchAllOutstanding 方法。

```

public void start() {
    fetchAllOutstanding();
}

```

fetchAllOutstanding 方法的实现如代码清单 6-105 所示。

代码清单6-105 fetchAllOutstanding的实现

```

private void fetchAllOutstanding() {
    // Start by retrieving our shared state within a synchronized block.
    String[] blockIdsToFetch;
    int numRetries;
    RetryingBlockFetchListener myListener;
    synchronized (this) {
        blockIdsToFetch = outstandingBlocksIds.toArray(new String[outstanding-
            BlocksIds.size()]);
        numRetries = retryCount;
        myListener = currentListener;
    }

    // 启动对Block的获取，并可能对获取失败的Block进行重试
    try {
        fetchStarter.createAndStart(blockIdsToFetch, myListener);
    } catch (Exception e) {

```

```
logger.error(String.format("Exception while beginning fetch of %s outstanding
    blocks %s",
    blockIdsToFetch.length, numRetries > 0 ? "(after " + numRetries + "
        retries)" : ""), e);

if (shouldRetry(e)) {
    initiateRetry();
} else {
    for (String bid : blockIdsToFetch) {
        listener.onBlockFetchFailure(bid, e);
    }
}
}
```

根据代码清单 6-105，fetchAllOutstanding 方法的执行步骤如下。

- 1) 调用 `fetchStarter` (即 `blockFetchStarter`) 的 `createAndStart` 方法。其中 `myListener` 为 `RetryingBlockFetchListener`, `RetryingBlockFetchListener` 是 `BlockFetchingListener` 的实现类。
 - 2) 如果上一步执行时抛出了异常，则调用 `shouldRetry` 方法判断是否需要重试。`shouldRetry` 方法 (见代码清单 6-106) 的判断依据是：异常是 `IOException` 并且当前的重试次数 `retryCount` 小于最大重试次数 `maxRetries`。

代码清单6-106 判断是否需要重试

```
private synchronized boolean shouldRetry(Throwable e) {  
    boolean isIOException = e instanceof IOException  
        || (e.getCause() != null && e.getCause() instanceof IOException);  
    boolean hasRemainingRetries = retryCount < maxRetries;  
    return isIOException && hasRemainingRetries;  
}
```

3) 当需要重试时调用 `initiateRetry` 方法再次重试, 如代码清单 6-107 所示。

代码清单6-107 重试的实现

```
private synchronized void initiateRetry() {
    retryCount += 1;
    currentListener = new RetryingBlockFetchListener();

    logger.info("Retrying fetch ({}/{}) for {} outstanding blocks after {} ms",
        retryCount, maxRetries, outstandingBlocksIds.size(), retryWaitTime);

    executorService.submit(new Runnable() {
        @Override
        public void run() {
            Uninterruptibles.sleepUninterruptibly(retryWaitTime, TimeUnit.MILLISECONDS);
            fetchAllOutstanding();
        }
    });
}
```

根据代码清单 6-107，重试的步骤如下。

- ① 重试次数 retryCount 加 1。
- ② 新建 RetryingBlockFetchListener。
- ③ 使用线程池 executorService (由 Executors.newCachedThreadPool 方法创建) 提交新的任务, 任务实际为调用 fetchAllOutstanding 方法。由此可以看出, 下载远端 Block 的重试机制是异步的。

根据对发送获取远端 Block 的请求的分析, 无论是请求一次还是异步多次重试, 最后都落实到调用 blockFetchStarter 的 createAndStart 方法。blockFetchStarter 的 createAndStart 方法 (见代码清单 6-104) 首先创建 TransportClient, 然后创建 OneForOneBlockFetcher 并调用其 start 方法。

客户端在需要下载远端节点的 Block 时, 将创建 OneForOneBlockFetcher, 然后使用 OneForOneBlockFetcher 完成 Block 的下载。了解 OneForOneBlockFetcher 之前, 应该首先理解其中的各个成员属性。

- client: 用于向服务端发送请求的 TransportClient。
- openMessage: 即 OpenBlocks。OpenBlocks 将携带远端节点的 appId(应用程序标识)、execId (Executor 标识) 和 blockIds (BlockId 的数组), 这表示从远端的那个实例获取哪些 Block, 并且知道是哪个远端 Executor 生成的 Block。
- blockIds: BlockId 的数组。与 openMessage 的 blockIds 属性一致。
- listener: 类型为 BlockFetchingListener, 将在获取 Block 成功或失败时被回调。
- chunkCallback: 获取块成功或失败时回调, 配合 BlockFetchingListener 使用。
- streamHandle: 根据代码清单 6-103, 客户端给服务端发送 OpenBlocks 消息后, 服务端会在 OneForOneStreamManager 的 streams 缓存中缓存从存储体系中读取到的 ManagedBuffer 序列, 并生成与 ManagedBuffer 序列对应的 streamId, 然后将 streamId 和 ManagedBuffer 序列的大小封装为 StreamHandle 消息返回给客户端, 客户端的 streamHandle 属性将持有此 StreamHandle 消息。

有了对 OneForOneBlockFetcher 的各个属性的了解, 现在来看看 OneForOneBlockFetcher 是如何获取远端 Block 的。OneForOneBlockFetcher 的 start 方法用于获取远端 Block, 其实现如代码清单 6-108 所示。

代码清单 6-108 获取远端 Block

```

public void start() {
    if (blockIds.length == 0) {
        throw new IllegalArgumentException("Zero-sized blockIds array");
    }

    client.sendRpc(openMessage.toByteBuffer(), new RpcResponseCallback() {
        @Override
        public void onSuccess(ByteBuffer response) {
            try {
                streamHandle = (StreamHandle) BlockTransferMessage.Decoder.fromByteBuffer

```

```
        (response);
    logger.trace("Successfully opened blocks {}, preparing to fetch chunks.",
                 streamHandle);

    for (int i = 0; i < streamHandle.numChunks; i++) { // 多次请求获取Block
        client.fetchChunk(streamHandle.streamId, i, chunkCallback);
    }
} catch (Exception e) {
    logger.error("Failed while starting block fetches after success", e);
    failRemainingBlocks(blockIds, e);
}
}

@Override
public void onFailure(Throwable e) {
    logger.error("Failed while starting block fetches", e);
    failRemainingBlocks(blockIds, e);
}
});
```

根据代码清单 6-108，OneForOneBlockFetcher 获取远端 Block 的步骤如下。

1) 校验要获取的 Block 的数量。如果要获取的 BlockId 的数组的大小为零，则抛出 IllegalArgumentException 异常。

2) 调用 TransportClient 的 sendRpc 方法发送 openMessage (即 OpenBlocks) 消息，并向客户端的 outstandingRpcs 缓存注册匿名的 RpcResponseCallback 实现。

3) 客户端等待接收服务端的响应(此时服务端正在进行代码清单 6-103 所介绍的处理)。

4) 当客户端收到服务端的响应时, 将从 outstandingRpcs 缓存取出代码清单 6-108 中的匿名 RpcResponseCallback 实现, 以服务端的响应类别 (RpcResponse 或 RpcFailure) 分别回调匿名 RpcResponseCallback 的不同方法 (onSuccess 或 onFailure)。

5) 如果回调 onSuccess，则先将服务端的 response 反序列化为 StreamHandle 类型，然后根据 StreamHandle 的 numChunks 属性的大小，按照索引由低到高，遍历调用 TransportClient 的 fetchChunk 方法逐个获取 Block。

TransportClient 的 sendRpc 方法和 fetchChunk 方法已在 3.2.8 节详细介绍过，此处不再赘述。

客户端发送下载 Block 的 RPC 请求及服务端接收此请求的内容，目前都已经进行了分析和讲解，为了能让读者把客户端和服务端的内容串起来，这里以图 6-9 来展示整个 Block 的下载流程。

图 6-9 中有很多序号。它们所代表的处理逻辑如下。

序号①：客户端调用 ShuffleClient（默认为 NettyBlockTransferService）的 fetchBlocks 方法下载远端节点的 Block，由于未指定 spark.shuffle.io.maxRetries 属性，所以将直接调用 BlockFetchStarter 的 createAndStart 方法。

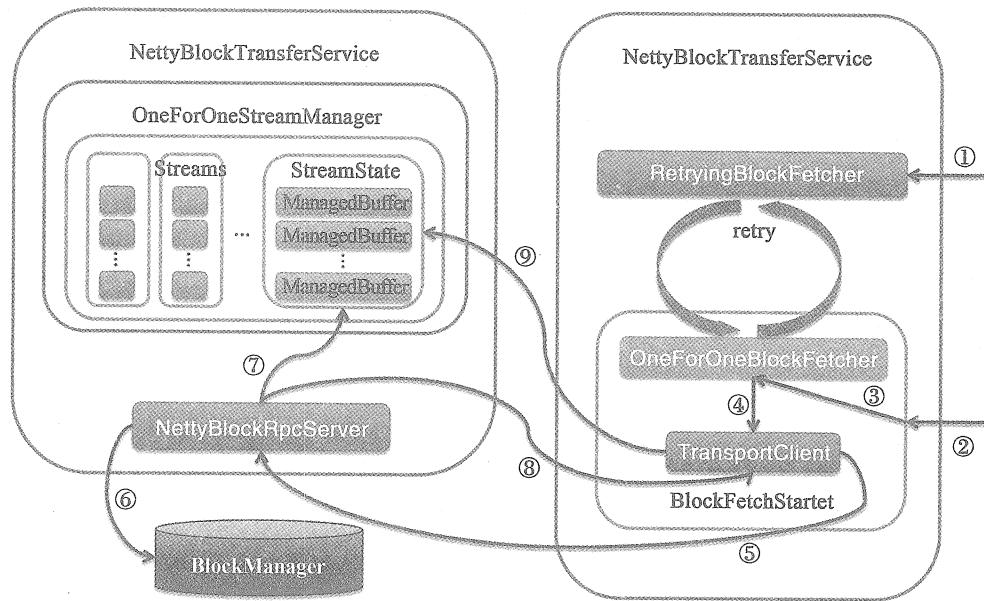


图 6-9 Shuffle 之 Block 的下载流程

序号②：客户端调用 ShuffleClient（默认为 NettyBlockTransferService）的 fetchBlocks 方法下载远端节点的 Block，由于未指定 park.shuffle.io.maxRetries 属性，所以首先调用 RetryingBlockFetcher 的 start 方法。RetryingBlockFetcher 内部实际也将调用 BlockFetch Starter 的 createAndStart 方法，并且在发生 IOException 且重试次数 retryCount 小于最大重试次数 maxRetries 时，多次尝试调用 BlockFetchStarter 的 createAndStart 方法。

序号③：BlockFetchStarter 的 createAndStart 方法实际将调用 OneForOneBlockFetcher 的 start 方法。

序号④：OneForOneBlockFetcher 将调用 TransportClient 的 sendRpc 方法发送 OpenBlocks 消息。OpenBlocks 携带着远端节点的 appId（应用程序标识）、execId（Executor 标识）和 blockIds（BlockId 的数组）等信息。此外，TransportClient 还将向 outstandingRpcs 中注册本次请求对应的匿名回调类 RpcResponseCallback。

序号⑤：NettyBlockRpcServer 接收到客户端发来的 OpenBlocks 消息，并获取携带的 BlockId 的数组信息。

序号⑥：NettyBlockRpcServer 对 BlockId 数组中的每个 BlockId，通过调用 BlockManager 的 getBlockData 方法获取数据块的信息（ManagedBuffer），所有读取的 Block 数据将构成一个 ManagedBuffer 类型的序列。

序号⑦：NettyBlockRpcServer 调用 OneForOneStreamManager 的 registerStream 方法生成 streamId 和 StreamState，并将 streamId 与 StreamState 的映射关系缓存在 streams 中。StreamState 中将包含着 appId（应用程序标识）和 ManagedBuffer 序列。

序号⑧：创建 StreamHandle（包含着 streamId 和 ManagedBuffer 序列的大小），并通过回调方法将 StreamHandle 封装为 RpcResponse，最后向客户端发送 RpcResponse。

序号⑨：TransportClient 接收到 RpcResponse 消息后，从 outstandingRpcs 中查找到本次请求对应的匿名回调类 RpcResponseCallback，此匿名 RpcResponseCallback 的 onSuccess 方法将 RpcResponse 消息解码为 StreamHandle，并根据 StreamHandle 的 numChunks（块数，即 ManagedBuffer 序列的大小），按照索引逐个调用 TransportClient 的 fetchChunk 方法从远端节点的缓存 streams 中找到与 streamId 对应的 StreamState，并根据索引返回 StreamState 的 ManagedBuffer 序列中的某一个 ManagedBuffer。

2. 同步下载远端 Block

由于 fetchBlocks 是异步下载的，因此 NettyBlockTransferService 的父类 BlockTransferService 对 fetchBlocks 封装后，提供了同步下载远端 Block 的方法，其实现如代码清单 6-109 所示。

代码清单 6-109 同步下载远端 Block

```
def fetchBlockSync(host: String, port: Int, execId: String, blockId: String): ManagedBuffer = {
    val result = Promise[ManagedBuffer]()
    fetchBlocks(host, port, execId, Array(blockId),
    new BlockFetchingListener {
        override def onBlockFetchFailure(blockId: String, exception: Throwable):
            Unit = {
            result.failure(exception)
        }
        override def onBlockFetchSuccess(blockId: String, data: ManagedBuffer):
            Unit = {
            val ret = ByteBuffer.allocate(data.size.toInt)
            ret.put(data.nioByteBuffer())
            ret.flip()
            result.success(new NioManagedBuffer(ret))
        }
    })
    ThreadUtils.awaitResult(result.future, Duration.Inf)
}
```

根据代码清单 6-109，其实质为调用 fetchBlocks。

3. 发送向远端上传 Block 的请求

NettyBlockTransferService 的 uploadBlock 方法的实现如代码清单 6-110 所示。

代码清单 6-110 uploadBlock 的实现

```
override def uploadBlock(
    hostname: String,
    port: Int,
    execId: String,
    blockId: BlockId,
    blockData: ManagedBuffer,
    level: StorageLevel,
```

```

    classTag: ClassTag[_]): Future[Unit] = {
      val result = Promise[Unit]()
      val client = clientFactory.createClient(hostname, port)
      val metadata = JavaUtils.bufferToArray(serializer.newInstance().serialize
        ((level, classTag)))
      val array = JavaUtils.bufferToArray(blockData.nioByteBuffer())
      client.sendRpc(new UploadBlock(appId, execId, blockId.toString, metadata,
        array).toByteBuffer,
        new RpcResponseCallback {
          override def onSuccess(response: ByteBuffer): Unit = {
            logTrace(s"Successfully uploaded block $blockId")
            result.success((): Unit)
          }
          override def onFailure(e: Throwable): Unit = {
            logError(s"Error while uploading block $blockId", e)
            result.failure(e)
          }
        })
      result.future
    }
}

```

根据代码清单 6-110，uploadBlock 方法的执行步骤如下。

- 1) 创建一个空 Promise，调用方将持有此 Promise 的 Future。
- 2) 创建 TransportClient。
- 3) 将存储级别 StorageLevel 和类型标记 classTag 等元数据序列化。
- 4) 调用 ManagedBuffer (实际是实现类 NettyManagedBuffer) 的 nioByteBuffer 方法将 Block 的数据转换或者复制为 Nio 的 ByteBuffer 类型。
- 5) 调用 TransportClient 的 sendRpc 方法发送 RPC 消息 UploadBlock。UploadBlock 消息的 appId 是应用程序的标识，execId 是上传目的地的 Executor 的标识，RpcResponseCallback 是匿名的实现类。上传成功则回调匿名 RpcResponseCallback 的 onSuccess 方法，进而调用 Promise 的 success 方法；上传失败则回调匿名 RpcResponseCallback 的 onFailure 方法，进而调用 Promise 的 failure 方法。

客户端发送上传 Block 的 RPC 请求及服务端接收此请求的内容，目前都已经进行了分析和讲解，为了能让读者把客户端和服务端的内容串起来，这里以图 6-10 来展示整个 Block 的上传流程。

图 6-10 中有很多序号，它们所代表的处理逻辑如下。

序号①：客户端从本地的 BlockManager 中读取 BlockId 对应的 Block，并转换为 ManagedBuffer 类型。

序号②：客户端调用 ShuffleClient (默认为 NettyBlockTransferService) 的 uploadBlock 方法上传 Block 到远端节点。NettyBlockTransferService 将 Block、Block 的存储级别及类型标记等信息进行序列化，生成 metadata 和 array。

序号③：NettyBlockTransferService 调用 TransportClient 的 sendRpc 方法发送 UploadBlock 消息。UploadBlock 携带着目标节点的 appId (应用程序标识)，execId (Executor 标识)，

BlockId 及 metadata 和 array 等信息。此外，TransportClient 还将向 outstandingRpcs 中注册本次请求对应的匿名回调类 RpcResponseCallback。

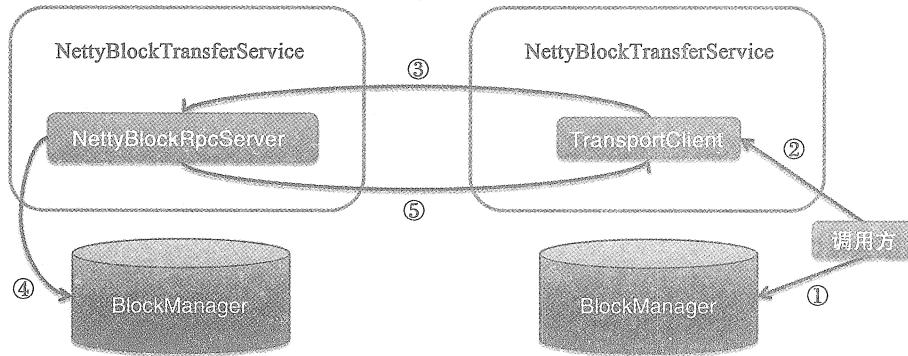


图 6-10 Block 的上传流程

序号④：NettyBlockRpcServer 接收到客户端发来的 UploadBlock 消息，将 UploadBlock 携带的 metadata 反序列化得到 lock 的存储级别及类型标记，将 UploadBlock 携带的 Block 数据（即 array）封装为 NioManagedBuffer，最后调用 BlockManager 的 putBlockData 方法将 Block 数据写入服务端本地的存储体系。

序号⑤：NettyBlockRpcServer 将处理成功的结果返回给客户端，客户端从 outstandingRpcs 中查找到本次请求对应的匿名回调类 RpcResponseCallback，并调用此匿名回调类 RpcResponseCallback 的 onSuccess 方法处理正确的响应。

4. 同步向远端上传 Block

由于 uploadBlock 是异步的，因此 NettyBlockTransferService 的父类 BlockTransferService 提供了以同步方式向远端上传 Block 的方法，其实现如代码清单 6-111 所示。

代码清单 6-111 同步向远端上传 Block

```

def fetchBlockSync(host: String, port: Int, execId: String, blockId: String): ManagedBuffer = {
    val result = Promise[ManagedBuffer]()
    fetchBlocks(host, port, execId, Array(blockId),
    new BlockFetchingListener {
        override def onBlockFetchFailure(blockId: String, exception: Throwable): Unit = {
            result.failure(exception)
        }
        override def onBlockFetchSuccess(blockId: String, data: ManagedBuffer): Unit = {
            val ret = ByteBuffer.allocate(data.size.toInt)
            ret.put(data.nioByteBuffer())
            ret.flip()
            result.success(new NioManagedBuffer(ret))
        }
    })
}

```

```

    ThreadUtils.awaitResult(result.future, Duration.Inf)
}

```

根据代码清单 6-111，其实质为调用 uploadBlock。

6.10 DiskBlockObjectWriter 详解

我们知道，BlockManager 的 getDiskWriter 方法用于创建 DiskBlockObjectWriter。DiskBlockObjectWriter 将在 Shuffle 阶段将 map 任务的输出写入磁盘，这样 reduce 任务就能够从磁盘中获取 map 任务的中间输出了。由于 DiskBlockObjectWriter 是通过 BlockManager 获取或创建的，所以本书将 DiskBlockObjectWriter 放在本章介绍。

DiskBlockObjectWriter 用于将 JVM 中的对象直接写入磁盘文件中。DiskBlockObjectWriter 允许将数据追加到现有 Block。为了提高效率，DiskBlockObjectWriter 保留了跨多个提交的底层文件通道。

分析 DiskBlockObjectWriter，应当从 DiskBlockObjectWriter 的属性开始。DiskBlockObjectWriter 包含的构造器属性如下。

- file：要写入的文件。
- serializerManager：即 SerializerManager。
- serializerInstance：Serializer 的实例。
- bufferSize：缓冲大小。
- syncWrites：是否同步写。
- writeMetrics：类型为 ShuffleWriteMetrics，用于对 Shuffle 中间结果写入到磁盘的度量与统计。
- blockId：即块的唯一身份标识 BlockId。

DiskBlockObjectWriter 的其他属性如下。

- channel：即 FileChannel。
- mcs：即 ManualCloseOutputStream。
- bs：即 OutputStream。
- fos：即 FileOutputStream。
- objOut：即 ObjectOutputStream。
- ts：即 TimeTrackingOutputStream。
- initialized：是否已经初始化。
- streamOpen：是否已经打开流。
- hasBeenClosed：是否已经关闭。
- committedPosition：提交的文件位置。
- reportedPosition：报告给度量系统的文件位置。

- numRecordsWritten：已写的记录数。

了解了属性，现在就可以方便地介绍 DiskBlockObjectWriter 中提供的方法了。

1. open

DiskBlockObjectWriter 的 open 方法（见代码清单 6-112）用于打开要写入文件的各种输出流及管道。

代码清单6-112 打开DiskBlockObjectWriter

```

private def initialize(): Unit = {
    fos = new FileOutputStream(file, true)
    channel = fos.getChannel()
    ts = new TimeTrackingOutputStream(writeMetrics, fos)
    class ManualCloseBufferedOutputStream
        extends BufferedOutputStream(ts, bufferSize) with ManualCloseOutputStream
    mcs = new ManualCloseBufferedOutputStream
}

def open(): DiskBlockObjectWriter = {
    if (hasBeenClosed) {
        throw new IllegalStateException("Writer already closed. Cannot be reopened.")
    }
    if (!initialized) {
        initialize()
        initialized = true
    }

    bs = serializerManager.wrapStream(blockId, mcs)
    objOut = serializerInstance.serializeStream(bs)
    streamOpen = true
    this
}

```

2. recordWritten

DiskBlockObjectWriter 的 recordWritten 方法（见代码清单 6-113）用于对写入的记录数进行统计和度量。

代码清单6-113 写入记录数的度量

```

def recordWritten(): Unit = {
    numRecordsWritten += 1
    writeMetrics.incRecordsWritten(1)

    if (numRecordsWritten % 16384 == 0) {
        updateBytesWritten()
    }
}
private def updateBytesWritten() {
    val pos = channel.position()

```

```

    writeMetrics.incBytesWritten(pos - reportedPosition)
    reportedPosition = pos
}

```

3. write

DiskBlockObjectWriter 的 write 方法（见代码清单 6-114）用于向输出流中写入键值对。

代码清单6-114 写入键值对

```

def write(key: Any, value: Any) {
    if (!streamOpen) {
        open()
    }

    objOut.writeKey(key)
    objOut.writeValue(value)
    recordWritten()
}

```

4. commitAndGet

DiskBlockObjectWriter 的 commitAndGet 方法（见代码清单 6-115）用于将输出流中的数据写入到磁盘。

代码清单6-115 提交并获取文件段

```

def commitAndGet(): FileSegment = {
    if (streamOpen) {
        objOut.flush()
        bs.flush()
        objOut.close()
        streamOpen = false

        if (syncWrites) {
            val start = System.nanoTime()
            fos.getFD.sync()
            writeMetrics.incWriteTime(System.nanoTime() - start)
        }

        val pos = channel.position()
        val fileSegment = new FileSegment(file, committedPosition, pos - committedPosition)
        committedPosition = pos
        writeMetrics.incBytesWritten(committedPosition - reportedPosition)
        reportedPosition = committedPosition
        fileSegment
    } else {
        new FileSegment(file, committedPosition, 0)
    }
}

```

根据代码清单 6-115，commitAndGet 方法将数据写入到文件后，创建 FileSegment 并返回。FileSegment 包含文件、写入的起始偏移量、写入的长度等信息。

6.11 小结

本章首先对构成 BlockManager 的各个组件进行了展示，然后逐一分析每个组件的实现。这些组件相互配合，进而实现了整个存储体系的功能。DiskBlockManager 和 DiskStore 一同负责磁盘存储；MemoryManager 和 MemoryStore 一同负责内存存储；BlockInfoManager 负责 Block 的元数据信息及锁的管理；BlockManager 本身则对上述组件的功能进行整合和封装；BlockManagerMaster 和 BlockManagerMasterEndpoint 一同管理分散在不同节点上的 BlockManager，BlockManagerSlaveEndpoint 则可以接收 BlockManagerMasterEndpoint 下发的各种命令；BlockTransferService 提供 Block 的上传与下载服务，Shuffle 客户端则提供 Block 的上传与下载的客户端实现。

MapOutputTracker 和 ShuffleManager 与存储体系也有一定的关系，MapOutputTracker 已在第 5 章介绍，ShuffleManager 由于与 Shuffle 的关系非常紧密，因此放在第 8 章介绍。