Spark源码分析之-Storage模块

arhitecture (1) (/categories.html#arhitecture-ref)

cloud 7 (/tags.html#cloud-ref) spark 8 (/tags.html#spark-ref)

08 October 2013

Background

前段时间琐事颇多，一直没有时间整理自己的博客，Spark源码分析写到一半也搁置了。之前介绍了**deploy** (http://jerryshao.me/architecture/2013/04/21/Spark%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90%E4%B9%8B-scheduler%E6%A8%A1%E5%9D%97/)和**scheduler** (http://jerryshao.me/architecture/2013/04/21/Spark%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90%E4%B9%8B-scheduler%E6%A8%A1%E5%9D%97/)两大模块，这次介绍Spark中的另一大模块 - storage模块。

在写Spark程序的时候我们常常和**RDD** (*Resilient Distributed Dataset*) 打交道，通过RDD为我们提供的各种transformation和action接口实现我们的应用，RDD的引入提高了抽象层次，在接口和实现上进行有效地隔离，使用户无需关心底层的实现。但是RDD提供给我们的仅仅是一个“形”, 我们所操作的数据究竟放在哪里，如何存取？它的“体”是怎么样的？这是由storage模块来实现和管理的，接下来我们就要剖析一下storage模块。

Storage模块整体架构

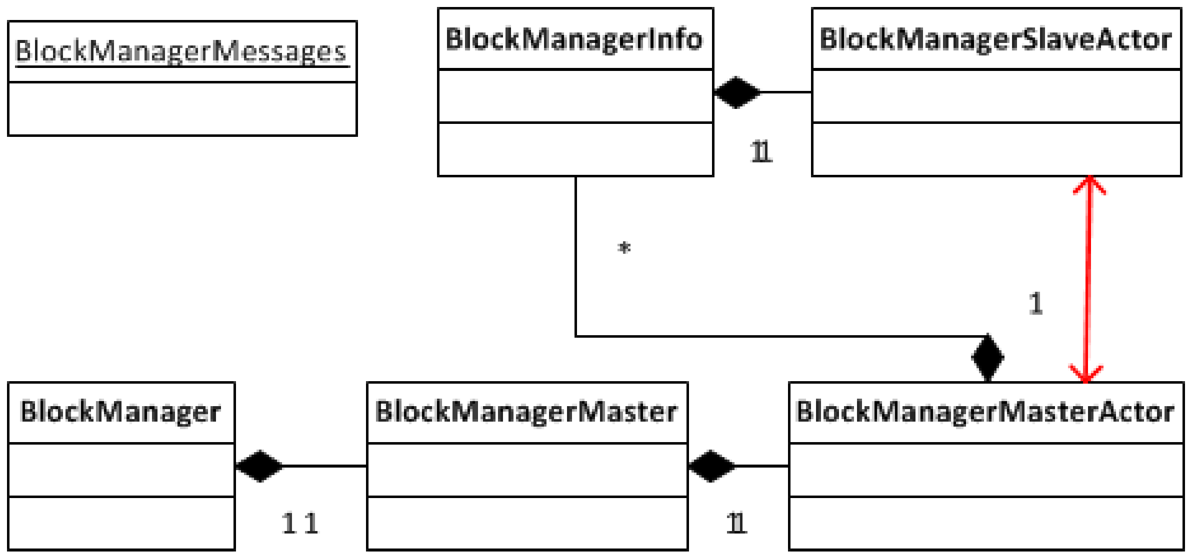
Storage模块主要分为两层：

1. 通信层：storage模块采用的是master-slave结构来实现通信层，master和slave之间传输控制信息、状态信息，这些都是通过通信层来实现的。
2. 存储层：storage模块需要把数据存储到disk或是memory上面，有可能还需replicate到远端，这都是由存储层来实现和提供相应接口。

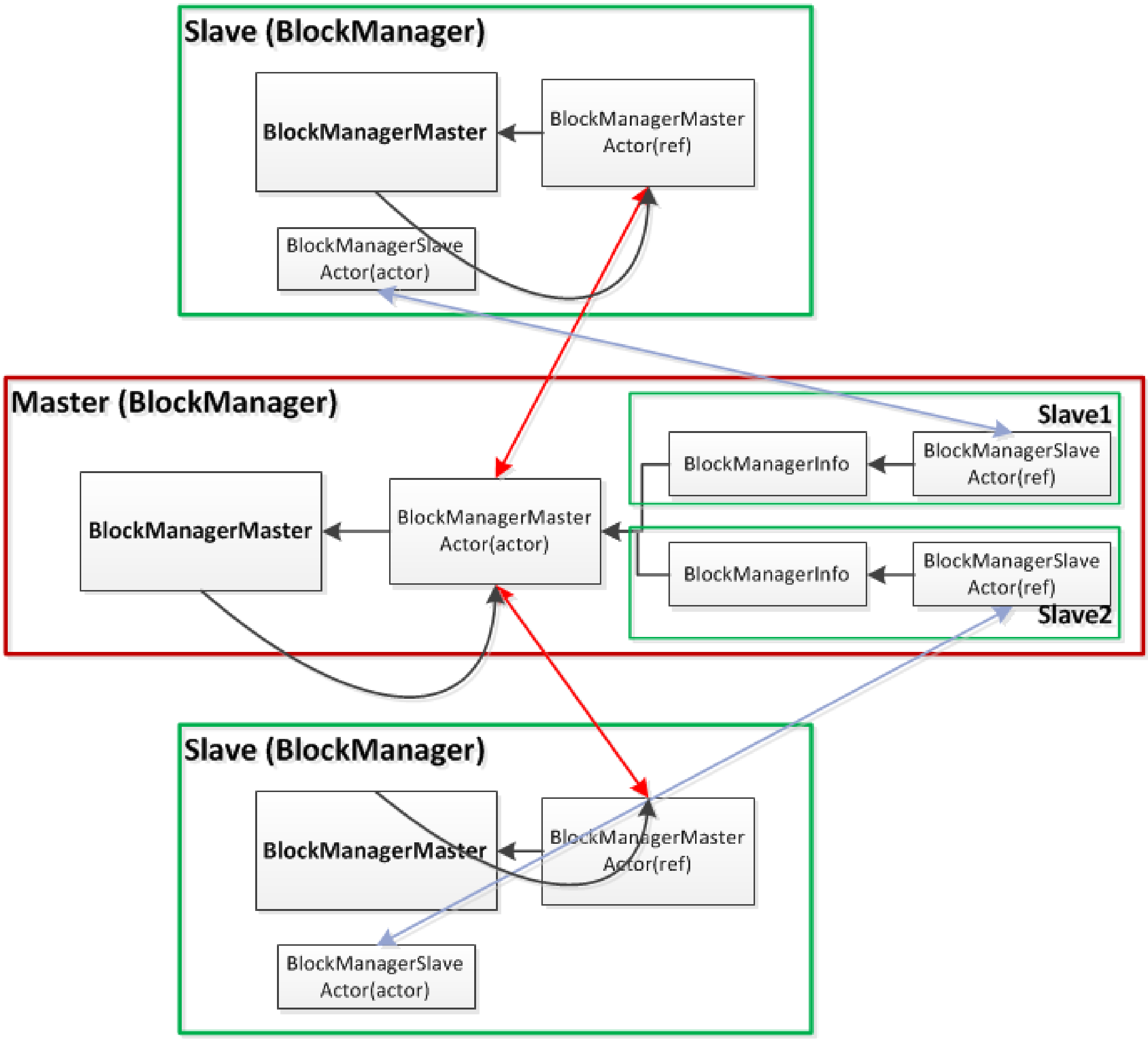
而其他模块若要和storage模块进行交互，storage模块提供了统一的操作类`BlockManager`，外部类与storage模块打交道都需要通过调用`BlockManager`相应接口来实现。

Storage模块通信层

首先来看一下通信层的UML类图:



其次我们来看看各个类在master和slave上所扮演的不同角色：



对于master和slave，`BlockManager` 的创建有所不同：

- Master (client driver)

  `BlockManagerMaster` 拥有 `BlockManagerMasterActor` 的*actor*和所有 `BlockManagerSlaveActor` 的*ref*。

- Slave (executor)

  对于slave，`BlockManagerMaster` 则拥有 `BlockManagerMasterActor` 的*ref*和自身 `BlockManagerSlaveActor` 的*actor*。

`BlockManagerMasterActor`在*ref*和*actor*之间进行通信；`BlockManagerSlaveActor`在*ref*和*actor*之间通信。

> *actor*和*ref*:
> *actor*和*ref*是**Akka** (http://akka.io/)中的两个不同的actor reference，分别由`actorOf`和`actorFor`所创建。*actor*类似于网络服务中的server端，它保存所有的状态信息，接收client端的请求执行并返回给客户端；*ref*类似于网络服务中的client端，通过向server端发起请求获取结果。

`BlockManager` wrap了`BlockManagerMaster`，通过`BlockManagerMaster`进行通信。Spark会在client driver和executor端创建各自的`BlockManager`，通过`BlockManager`对storage模块进行操作。

`BlockManager`对象在`SparkEnv`中被创建，创建的过程如下所示：

```
def registerOrLookup(name: String, newActor: => Actor): ActorRef = {
  if (isDriver) {
    logInfo("Registering " + name)
    actorSystem.actorOf(Props(newActor), name = name)
  } else {
    val driverHost: String = System.getProperty("spark.driver.host", "localhost")
    val driverPort: Int = System.getProperty("spark.driver.port", "7077").toInt
    Utils.checkHost(driverHost, "Expected hostname")
    val url = "akka://spark@%s:%s/user/%s".format(driverHost, driverPort, name)
    logInfo("Connecting to " + name + ": " + url)
    actorSystem.actorFor(url)
  }
}

val blockManagerMaster = new BlockManagerMaster(registerOrLookup(
  "BlockManagerMaster",
  new BlockManagerMasterActor(isLocal)))
val blockManager = new BlockManager(executorId, actorSystem, blockManagerMaster, serializer)
```

可以看到对于client driver和executor，Spark分别创建了`BlockManagerMasterActor` *actor*和*ref*，并被wrap到`BlockManager`中。

通信层传递的消息

- **BlockManagerMasterActor**

  * **executor** to **client driver**

```
RegisterBlockManager (executor创建BlockManager以后向client driver发送请求注册自身)
HeartBeat
UpdateBlockInfo (更新block信息)
GetPeers (请求获得其他BlockManager的id)
GetLocations (获取block所在的BlockManager的id)
GetLocationsMultipleBlockIds (获取一组block所在的BlockManager id)
```

  * **client driver** to **client driver**

```
GetLocations (获取block所在的BlockManager的id)
GetLocationsMultipleBlockIds (获取一组block所在的BlockManager id)
RemoveExecutor (删除所保存的已经死亡的executor上的BlockManager)
StopBlockManagerMaster (停止client driver上的BlockManagerMasterActor)
```

> 有些消息例如`GetLocations`在executor端和client driver端都会向*actor*请求，而其他的消息比如`RegisterBlockManager`只会由executor端的*ref*向client driver端的*actor*发送，于此同时例如`RemoveExecutor`则只会由client driver端的*ref*向client driver端的*actor*发送。
> 具体消息是从哪里发送，哪里接收和处理请看代码细节，在这里就不再赘述了。

- **BlockManagerSlaveActor**

  * **client driver** to **executor**

  ```
  RemoveBlock (删除block)
  RemoveRdd (删除RDD)
  ```

通信层中涉及许多控制消息和状态消息的传递以及处理，这些细节可以直接查看源码，这里就不在一一罗列。下面就只简单介绍一下exeuctor端的`BlockManager`是如何启动以及向client driver发送注册请求完成注册。

Register BlockManager

前面已经介绍了`BlockManager`对象是如何被创建出来的，当`BlockManager`被创建出来以后需要向client driver注册自己，下面我们来看一下这个流程：

首先`BlockManager`会调用`initialize()`初始化自己

```
private def initialize() {
  master.registerBlockManager(blockManagerId, maxMemory, slaveActor)
  ...
  if (!BlockManager.getDisableHeartBeatsForTesting) {
    heartBeatTask = actorSystem.scheduler.schedule(0.seconds, heartBeatFrequency.milliseconds) {
      heartBeat()
    }
  }
}
```

在`initialized()`函数中首先调用`BlockManagerMaster`向client driver注册自己，同时设置heartbeat定时器，定时发送heartbeat报文。可以看到在注册自身的时候向client driver传递了自身的`slaveActor`，client driver收到`slaveActor`以后会将其与之对应的`BlockManagerInfo`存储到hash map中，以便后续通过`slaveActor`向executor发送命令。

`BlockManagerMaster`会将注册请求包装成`RegisterBlockManager`报文发送给client driver的`BlockManagerMasterActor`，`BlockManagerMasterActor`调用`register()`函数注册`BlockManager`：

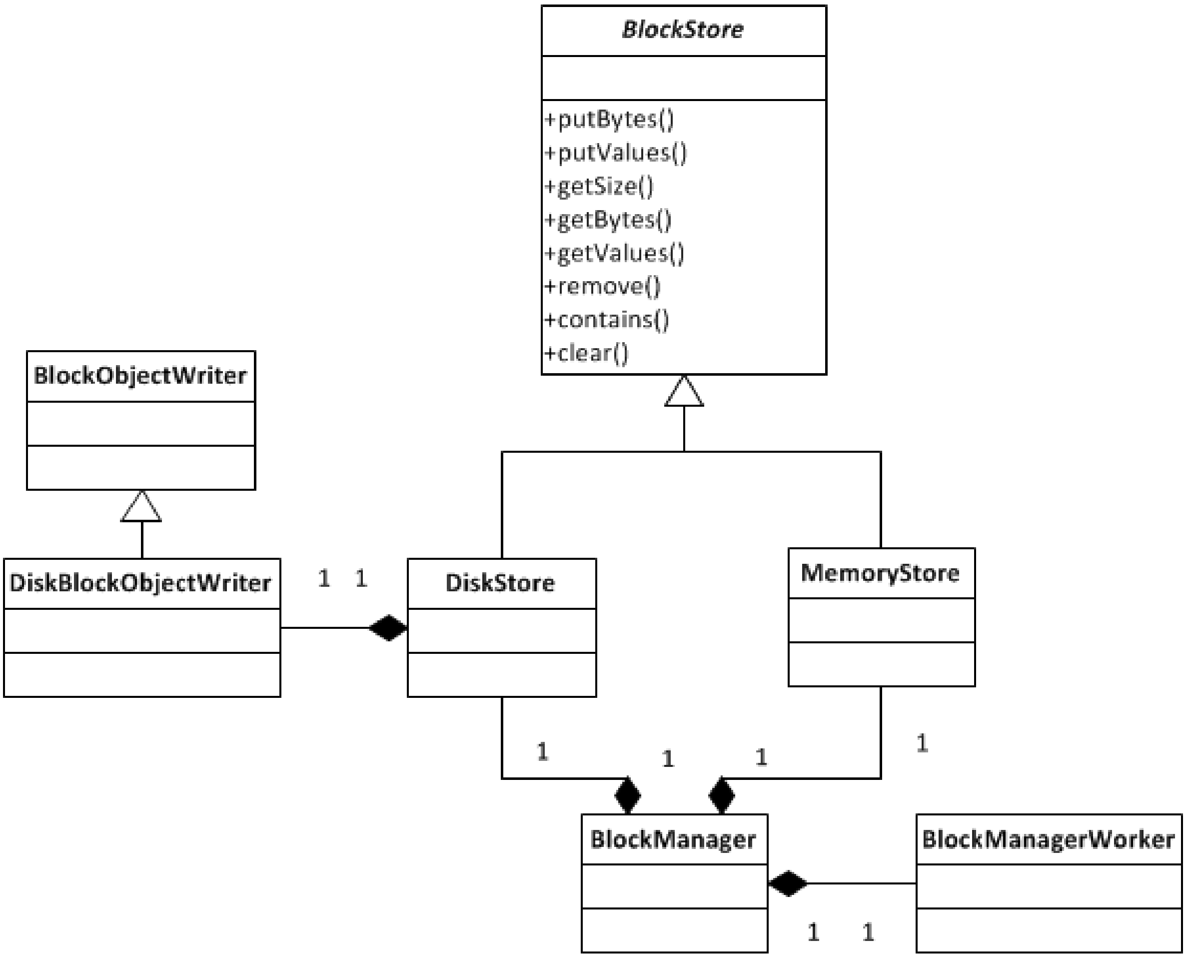
```
private def register(id: BlockManagerId, maxMemSize: Long, slaveActor: ActorRef) {
  if (id.executorId == "<driver>" && !isLocal) {
    // Got a register message from the master node; don't register it
  } else if (!blockManagerInfo.contains(id)) {
    blockManagerIdByExecutor.get(id.executorId) match {
      case Some(manager) =>
        // A block manager of the same executor already exists.
        // This should never happen. Let's just quit.
        logError("Got two different block manager registrations on " + id.executorId)
        System.exit(1)
      case None =>
        blockManagerIdByExecutor(id.executorId) = id
    }
    blockManagerInfo(id) = new BlockManagerMasterActor.BlockManagerInfo(
      id, System.currentTimeMillis(), maxMemSize, slaveActor)
  }
}
```

需要注意的是在client driver端也会执行上述过程，只是在最后注册的时候如果判断是`"<driver>"`就不进行任何操作。可以看到对应的`BlockManagerInfo`对象被创建并保存在hash map中。

Storage模块存储层

在RDD层面上我们了解到RDD是由不同的partition组成的，我们所进行的transformation和action是在partition上面进行的；而在storage模块内部，RDD又被视为由不同的block组成，对于RDD的存取是以block为单位进行的，本质上partition和block是等价的，只是看待的角度不同。在Spark storage模块中中存取数据的最小单位是block，所有的操作都是以block为单位进行的。

首先我们来看一下存储层的UML类图:



`BlockManager`对象被创建的时候会创建出`MemoryStore`和`DiskStore`对象用以存取block，同时在`initialize()`函数中创建`BlockManagerWorker`对象用以监听远程的block存取请求来进行相应处理。

```
private[storage] val memoryStore: BlockStore = new MemoryStore(this, maxMemory)
private[storage] val diskStore: DiskStore =
  new DiskStore(this, System.getProperty("spark.local.dir", System.getProperty("java.io.tmpdir")))

private def initialize() {
  ...
  BlockManagerWorker.startBlockManagerWorker(this)
  ...
}
```

下面就具体介绍一下对于`DiskStore`和`MemoryStore`，block的存取操作是怎样进行的。

DiskStore如何存取block

`DiskStore`可以配置多个folder，Spark会在不同的folder下面创建Spark文件夹，文件夹的命名方式为(spark-local-yyyyMMddHHmmss-xxxx, xxxx是一个随机数)，所有的block都会存储在所创建的folder里面。`DiskStore`会在对象被创建时调用`createLocalDirs()`来创建文件夹：

```
private def createLocalDirs(): Array[File] = {
  logDebug("Creating local directories at root dirs '" + rootDirs + "'")
  val dateFormat = new SimpleDateFormat("yyyyMMddHHmmss")
  rootDirs.split(",").map { rootDir =>
    var foundLocalDir = false
    var localDir: File = null
    var localDirId: String = null
    var tries = 0
    val rand = new Random()
    while (!foundLocalDir && tries < MAX_DIR_CREATION_ATTEMPTS) {
      tries += 1
      try {
        localDirId = "%s-%04x".format(dateFormat.format(new Date), rand.nextInt(65536))
        localDir = new File(rootDir, "spark-local-" + localDirId)
        if (!localDir.exists) {
          foundLocalDir = localDir.mkdirs()
        }
      } catch {
        case e: Exception =>
          logWarning("Attempt " + tries + " to create local dir " + localDir + " failed", e)
      }
    }
    if (!foundLocalDir) {
      logError("Failed " + MAX_DIR_CREATION_ATTEMPTS +
        " attempts to create local dir in " + rootDir)
      System.exit(ExecutorExitCode.DISK_STORE_FAILED_TO_CREATE_DIR)
    }
    logInfo("Created local directory at " + localDir)
    localDir
  }
}
```

在`DiskStore`里面，每一个block都被存储为一个file，通过计算block id的hash值将block映射到文件中，block id与文件路径的映射关系如下所示：

```scala
private def getFile(blockId: String): File = {
  logDebug("Getting file for block " + blockId)

  // Figure out which local directory it hashes to, and which subdirectory in that
  val hash = Utils.nonNegativeHash(blockId)
  val dirId = hash % localDirs.length
  val subDirId = (hash / localDirs.length) % subDirsPerLocalDir

  // Create the subdirectory if it doesn't already exist
  var subDir = subDirs(dirId)(subDirId)
  if (subDir == null) {
    subDir = subDirs(dirId).synchronized {
      val old = subDirs(dirId)(subDirId)
      if (old != null) {
        old
      } else {
        val newDir = new File(localDirs(dirId), "%02x".format(subDirId))
        newDir.mkdir()
        subDirs(dirId)(subDirId) = newDir
        newDir
      }
    }
  }

  new File(subDir, blockId)
}
```

根据block id计算出hash值，将hash取模获得`dirId`和`subDirId`，在`subDirs`中找出相应的`subDir`，若没有则新建一个`subDir`，最后以`subDir`为路径、block id为文件名创建file handler，`DiskStore`使用此file handler将block写入文件内，代码如下所示：

```scala
override def putBytes(blockId: String, _bytes: ByteBuffer, level: StorageLevel) {
  // So that we do not modify the input offsets !
  // duplicate does not copy buffer, so inexpensive
  val bytes = _bytes.duplicate()
  logDebug("Attempting to put block " + blockId)
  val startTime = System.currentTimeMillis
  val file = createFile(blockId)
  val channel = new RandomAccessFile(file, "rw").getChannel()
  while (bytes.remaining > 0) {
    channel.write(bytes)
  }
  channel.close()
  val finishTime = System.currentTimeMillis
  logDebug("Block %s stored as %s file on disk in %d ms".format(
    blockId, Utils.bytesToString(bytes.limit), (finishTime - startTime)))
}
```

而获取block则非常简单，找到相应的文件并读取出来即可：

```
override def getBytes(blockId: String): Option[ByteBuffer] = {
  val file = getFile(blockId)
  val bytes = getFileBytes(file)
  Some(bytes)
}
```

因此在`DiskStore`中存取block首先是要将block id映射成相应的文件路径，接着存取文件就可以了。

MemoryStore如何存取block

相对于`DiskStore`需要根据block id hash计算出文件路径并将block存放到对应的文件里面，`MemoryStore`管理block就显得非常简单：`MemoryStore`内部维护了一个hash map来管理所有的block，以block id为key将block存放到hash map中。

```
case class Entry(value: Any, size: Long, deserialized: Boolean)

private val entries = new LinkedHashMap[String, Entry](32, 0.75f, true)
```

在`MemoryStore`中存放block必须确保内存足够容纳下该block，若内存不足则会将block写到文件中，具体的代码如下所示：

```
override def putBytes(blockId: String, _bytes: ByteBuffer, level: StorageLevel) {
  // Work on a duplicate - since the original input might be used elsewhere.
  val bytes = _bytes.duplicate()
  bytes.rewind()
  if (level.deserialized) {
    val values = blockManager.dataDeserialize(blockId, bytes)
    val elements = new ArrayBuffer[Any]
    elements ++= values
    val sizeEstimate = SizeEstimator.estimate(elements.asInstanceOf[AnyRef])
    tryToPut(blockId, elements, sizeEstimate, true)
  } else {
    tryToPut(blockId, bytes, bytes.limit, false)
  }
}
```

在`tryToPut()`中，首先调用`ensureFreeSpace()`确保空闲内存是否足以容纳block，若可以则将该block放入hash map中进行管理；若不足以容纳则通过调用`dropFromMemory()`将block写入文件。

```scala
private def tryToPut(blockId: String, value: Any, size: Long, deserialized: Boolean): Boolean = {
  // TODO: Its possible to optimize the locking by locking entries only when selecting blocks
  // to be dropped. Once the to-be-dropped blocks have been selected, and lock on entries has been
  // released, it must be ensured that those to-be-dropped blocks are not double counted for
  // freeing up more space for another block that needs to be put. Only then the actually dropping
  // of blocks (and writing to disk if necessary) can proceed in parallel.
  putLock.synchronized {
    if (ensureFreeSpace(blockId, size)) {
      val entry = new Entry(value, size, deserialized)
      entries.synchronized {
        entries.put(blockId, entry)
        currentMemory += size
      }
      if (deserialized) {
        logInfo("Block %s stored as values to memory (estimated size %s, free %s)".format(
          blockId, Utils.bytesToString(size), Utils.bytesToString(freeMemory)))
      } else {
        logInfo("Block %s stored as bytes to memory (size %s, free %s)".format(
          blockId, Utils.bytesToString(size), Utils.bytesToString(freeMemory)))
      }
      true
    } else {
      // Tell the block manager that we couldn't put it in memory so that it can drop it to
      // disk if the block allows disk storage.
      val data = if (deserialized) {
        Left(value.asInstanceOf[ArrayBuffer[Any]])
      } else {
        Right(value.asInstanceOf[ByteBuffer].duplicate())
      }
      blockManager.dropFromMemory(blockId, data)
      false
    }
  }
}
```

而从`MemoryStore`中取得block则非常简单，只需从hash map中取出block id对应的value即可。

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

Put or Get block through BlockManager

上面介绍了`DiskStore`和`MemoryStore`对于block的存取操作，那么我们是要直接与它们交互存取数据吗，还是封装了更抽象的接口使我们无需关心底层？

`BlockManager`为我们提供了`put()`和`get()`函数，用户可以使用这两个函数对block进行存取而无需关心底层实现。

首先我们来看一下`put()`函数的实现：

```
def put(blockId: String, values: ArrayBuffer[Any], level: StorageLevel,
  tellMaster: Boolean = true) : Long = {

  ...

  // Remember the block's storage level so that we can correctly drop it to disk if it needs
  // to be dropped right after it got put into memory. Note, however, that other threads will
  // not be able to get() this block until we call markReady on its BlockInfo.
  val myInfo = {
    val tinfo = new BlockInfo(level, tellMaster)
    // Do atomically !
    val oldBlockOpt = blockInfo.putIfAbsent(blockId, tinfo)

    if (oldBlockOpt.isDefined) {
      if (oldBlockOpt.get.waitForReady()) {
        logWarning("Block " + blockId + " already exists on this machine; not re-adding it")
        return oldBlockOpt.get.size
      }

      // TODO: So the block info exists - but previous attempt to load it (?) failed. What do we do now ? Retry on it ?
      oldBlockOpt.get
    } else {
      tinfo
    }
  }

  val startTimeMs = System.currentTimeMillis

  // If we need to replicate the data, we'll want access to the values, but because our
  // put will read the whole iterator, there will be no values left. For the case where
  // the put serializes data, we'll remember the bytes, above; but for the case where it
  // doesn't, such as deserialized storage, let's rely on the put returning an Iterator.
  var valuesAfterPut: Iterator[Any] = null

  // Ditto for the bytes after the put
  var bytesAfterPut: ByteBuffer = null

  // Size of the block in bytes (to return to caller)
  var size = 0L

  myInfo.synchronized {
    logTrace("Put for block " + blockId + " took " + Utils.getUsedTimeMs(startTimeMs)
      + " to get into synchronized block")

    var marked = false
    try {
      if (level.useMemory) {
        // Save it just to memory first, even if it also has useDisk set to true; we will later
```

```scala
          // drop it to disk if the memory store can't hold it.
          val res = memoryStore.putValues(blockId, values, level, true)
          size = res.size
          res.data match {
            case Right(newBytes) => bytesAfterPut = newBytes
            case Left(newIterator) => valuesAfterPut = newIterator
          }
        } else {
          // Save directly to disk.
          // Don't get back the bytes unless we replicate them.
          val askForBytes = level.replication > 1
          val res = diskStore.putValues(blockId, values, level, askForBytes)
          size = res.size
          res.data match {
            case Right(newBytes) => bytesAfterPut = newBytes
            case _ =>
          }
        }

        // Now that the block is in either the memory or disk store, let other threads read it,
        // and tell the master about it.
        marked = true
        myInfo.markReady(size)
        if (tellMaster) {
          reportBlockStatus(blockId, myInfo)
        }
      } finally {
        // If we failed at putting the block to memory/disk, notify other possible readers
        // that it has failed, and then remove it from the block info map.
        if (! marked) {
          // Note that the remove must happen before markFailure otherwise another thread
          // could've inserted a new BlockInfo before we remove it.
          blockInfo.remove(blockId)
          myInfo.markFailure()
          logWarning("Putting block " + blockId + " failed")
        }
      }
    }
    logDebug("Put block " + blockId + " locally took " + Utils.getUsedTimeMs(startTimeMs))

    // Replicate block if required
    if (level.replication > 1) {
      val remoteStartTime = System.currentTimeMillis
      // Serialize the block if not already done
      if (bytesAfterPut == null) {
        if (valuesAfterPut == null) {
          throw new SparkException(
            "Underlying put returned neither an Iterator nor bytes! This shouldn't happen.")
        }
        bytesAfterPut = dataSerialize(blockId, valuesAfterPut)
      }
      replicate(blockId, bytesAfterPut, level)
      logDebug("Put block " + blockId + " remotely took " + Utils.getUsedTimeMs(remoteStartTime))
    }
    BlockManager.dispose(bytesAfterPut)
```

```
    return size
  }
```

对于`put()`操作，主要分为以下3个步骤：

1. 为block创建`BlockInfo`结构体存储block相关信息，同时将其加锁使其不能被访问。
2. 根据block的storage level将block存储到memory或是disk上，同时解锁标识该block已经ready，可被访问。
3. 根据block的replication数决定是否将该block replicate到远端。

接着我们来看一下`get()`函数的实现：

```
def get(blockId: String): Option[Iterator[Any]] = {
  val local = getLocal(blockId)
  if (local.isDefined) {
    logInfo("Found block %s locally".format(blockId))
    return local
  }
  val remote = getRemote(blockId)
  if (remote.isDefined) {
    logInfo("Found block %s remotely".format(blockId))
    return remote
  }
  None
}
```

`get()`首先会从local的`BlockManager`中查找block，如果找到则返回相应的block，若local没有找到该block，则发起请求从其他的executor上的`BlockManager`中查找block。在通常情况下Spark任务的分配是根据block的分布决定的，任务往往会被分配到拥有block的节点上，因此`getLocal()`就能找到所需的block；但是在资源有限的情况下，Spark会将任务调度到与block不同的节点上，这样就必须通过`getRemote()`来获得block。

我们先来看一下`getLocal()`：

```
def getLocal(blockId: String): Option[Iterator[Any]] = {
  logDebug("Getting local block " + blockId)
  val info = blockInfo.get(blockId).orNull
  if (info != null) {
    info.synchronized {

      // In the another thread is writing the block, wait for it to become ready.
      if (!info.waitForReady()) {
        // If we get here, the block write failed.
        logWarning("Block " + blockId + " was marked as failure.")
        return None
      }

      val level = info.level
      logDebug("Level for block " + blockId + " is " + level)

      // Look for the block in memory
      if (level.useMemory) {
        logDebug("Getting block " + blockId + " from memory")
        memoryStore.getValues(blockId) match {
          case Some(iterator) =>
            return Some(iterator)
```

```
          case None =>
            logDebug("Block " + blockId + " not found in memory")
        }
      }

      // Look for block on disk, potentially loading it back into memory if required
      if (level.useDisk) {
        logDebug("Getting block " + blockId + " from disk")
        if (level.useMemory && level.deserialized) {
          diskStore.getValues(blockId) match {
            case Some(iterator) =>
              // Put the block back in memory before returning it
              // TODO: Consider creating a putValues that also takes in a iterator ?
              val elements = new ArrayBuffer[Any]
              elements ++= iterator
              memoryStore.putValues(blockId, elements, level, true).data match {
                case Left(iterator2) =>
                  return Some(iterator2)
                case _ =>
                  throw new Exception("Memory store did not return back an iterator")
              }
            case None =>
              throw new Exception("Block " + blockId + " not found on disk, though it should be")
          }
        } else if (level.useMemory && !level.deserialized) {
          // Read it as a byte buffer into memory first, then return it
          diskStore.getBytes(blockId) match {
            case Some(bytes) =>
              // Put a copy of the block back in memory before returning it. Note that we can't
              // put the ByteBuffer returned by the disk store as that's a memory-mapped file.
              // The use of rewind assumes this.
              assert (0 == bytes.position())
              val copyForMemory = ByteBuffer.allocate(bytes.limit)
              copyForMemory.put(bytes)
              memoryStore.putBytes(blockId, copyForMemory, level)
              bytes.rewind()
              return Some(dataDeserialize(blockId, bytes))
            case None =>
              throw new Exception("Block " + blockId + " not found on disk, though it should be")
          }
        } else {
          diskStore.getValues(blockId) match {
            case Some(iterator) =>
              return Some(iterator)
            case None =>
              throw new Exception("Block " + blockId + " not found on disk, though it should be")
          }
        }
      }
    }
  } else {
    logDebug("Block " + blockId + " not registered locally")
  }
  return None
}
```

`getLocal()` 首先会根据block id获得相应的 `BlockInfo` 并从中取出该block的storage level，根据storage level的不同 `getLocal()` 又进入以下不同分支：

1. level.useMemory == true：从memory中取出block并返回，若没有取到则进入分支2。
2. level.useDisk == true:
    - level.useMemory == true: 将block从disk中读出并写入内存以便下次使用时直接从内存中获得，同时返回该block。
    - level.useMemory == false: 将block从disk中读出并返回
3. level.useDisk == false: 没有在本地找到block，返回None。

接下来我们来看一下 `getRemote()` ：

```
def getRemote(blockId: String): Option[Iterator[Any]] = {
  if (blockId == null) {
    throw new IllegalArgumentException("Block Id is null")
  }
  logDebug("Getting remote block " + blockId)
  // Get locations of block
  val locations = master.getLocations(blockId)

  // Get block from remote locations
  for (loc <- locations) {
    logDebug("Getting remote block " + blockId + " from " + loc)
    val data = BlockManagerWorker.syncGetBlock(
      GetBlock(blockId), ConnectionManagerId(loc.host, loc.port))
    if (data != null) {
      return Some(dataDeserialize(blockId, data))
    }
    logDebug("The value of block " + blockId + " is null")
  }
  logDebug("Block " + blockId + " not found")
  return None
}
```

`getRemote()` 首先取得该block的所有location信息，然后根据location向远端发送请求获取block，只要有一个远端返回block该函数就返回而不继续发送请求。

至此我们简单介绍了 `BlockManager` 类中的 `get()` 和 `put()` 函数，使用这两个函数外部类可以轻易地存取block数据。

Partition如何转化为Block

在storage模块里面所有的操作都是和block相关的，但是在RDD里面所有的运算都是基于partition的，那么partition是如何与block对应上的呢？

RDD计算的核心函数是 `iterator()` 函数：

```
final def iterator(split: Partition, context: TaskContext): Iterator[T] = {
  if (storageLevel != StorageLevel.NONE) {
    SparkEnv.get.cacheManager.getOrCompute(this, split, context, storageLevel)
  } else {
    computeOrReadCheckpoint(split, context)
  }
}
```

如果当前RDD的storage level不是NONE的话，表示该RDD在`BlockManager`中有存储，那么调用`CacheManager`中的`getOrCompute()`函数计算RDD，在这个函数中partition和block发生了关系：

首先根据RDD id和partition index构造出block id (rdd_xx_xx)，接着从`BlockManager`中取出相应的block。

- 如果该block存在，表示此RDD在之前已经被计算过和存储在`BlockManager`中，因此取出即可，无需再重新计算。
- 如果该block不存在则需要调用RDD的`computeOrReadCheckpoint()`函数计算出新的block，并将其存储到`BlockManager`中。

需要注意的是block的计算和存储是阻塞的，若另一线程也需要用到此block则需等到该线程block的loading结束。

```
def getOrCompute[T](rdd: RDD[T], split: Partition, context: TaskContext, storageLevel: StorageLevel)
    : Iterator[T] = {
  val key = "rdd_%d_%d".format(rdd.id, split.index)
  logDebug("Looking for partition " + key)
  blockManager.get(key) match {
    case Some(values) =>
      // Partition is already materialized, so just return its values
      return values.asInstanceOf[Iterator[T]]

    case None =>
      // Mark the split as loading (unless someone else marks it first)
      loading.synchronized {
        if (loading.contains(key)) {
          logInfo("Another thread is loading %s, waiting for it to finish...".format (key))
          while (loading.contains(key)) {
            try {loading.wait()} catch {case _ : Throwable =>}
          }
          logInfo("Finished waiting for %s".format(key))
          // See whether someone else has successfully loaded it. The main way this would fail
          // is for the RDD-level cache eviction policy if someone else has loaded the same RDD
          // partition but we didn't want to make space for it. However, that case is unlikely
          // because it's unlikely that two threads would work on the same RDD partition. One
          // downside of the current code is that threads wait serially if this does happen.
          blockManager.get(key) match {
            case Some(values) =>
              return values.asInstanceOf[Iterator[T]]
            case None =>
              logInfo("Whoever was loading %s failed; we'll try it ourselves".format  (key))
              loading.add(key)
          }
        } else {
          loading.add(key)
        }
      }
      try {
        // If we got here, we have to load the split
        logInfo("Partition %s not found, computing it".format(key))
        val computedValues = rdd.computeOrReadCheckpoint(split, context)
        // Persist the result, so long as the task is not running locally
        if (context.runningLocally) { return computedValues }
        val elements = new ArrayBuffer[Any]
        elements ++= computedValues
        blockManager.put(key, elements, storageLevel, true)
        return elements.iterator.asInstanceOf[Iterator[T]]
      } finally {
        loading.synchronized {
          loading.remove(key)
          loading.notifyAll()
        }
      }
  }
}
```

这样RDD的transformation、action就和block数据建立了联系，虽然抽象上我们的操作是在partition层面上进行的，但是partition最终还是被映射成为block，因此实际上我们的所有操作都是对block的处理和存取。

End

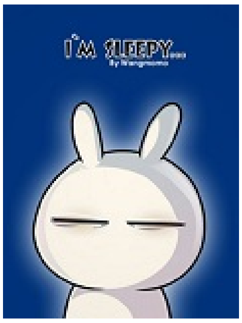
本文就storage模块的两个层面进行了介绍-通信层和存储层。通信层中简单介绍了类结构和组成以及类在通信层中所扮演的不同角色，还有不同角色之间通信的报文，同时简单介绍了通信层的启动和注册细节。存储层中分别介绍了`DiskStore`和`MemoryStore`中对于block的存和取的实现代码，同时分析了`BlockManager`中`put()`和`get()`接口，最后简单介绍了Spark RDD中的partition与`BlockManager`中的block之间的关系，以及如何交互存取block的。

本文从整体上分析了storage模块的实现，并未就具体实现做非常细节的分析，相信在看完本文对storage模块有一个整体的印象以后再去分析细节的实现会有事半功倍的效果。

← Previous (/functional%20programming/2013/08/30/Functional-Abstraction-of-Sequence-Spark-API-Design)

Archive (/archive.html) Next →

Wangda Tan • 14 days ago
楼主太强大了，这些正是最近想学的，谢谢楼主的分享！！



CATEGORIES

lessons (1) (/categories.html#lessons-ref)

test (1) (/categories.html#test-ref)

architecture (6) (/categories.html#architecture-ref)

functional programming (1) (/categories.html#functional programming-ref)

arhitecture (1) (/categories.html#arhitecture-ref)

LINKS

阮一峰的网络日志 (http://www.ruanyifeng.com/blog/)

刘未鹏 (http://mindhacks.cn/)

酷壳 (http://coolshell.cn/)

BeiYuu.com (http://beiyuu.com/)

MY FAVORITES