一：Broadcast彻底解析

**HttpServer：文件共享服务器。**

**Broadcast可以广播哪些内容：**

**数据表，RDD**

1. **Broadcast**：就是将数据从一个节点发送到其他的节点上。例如：Driver上有一张表，而executor中的每个并行执行的Task(100万个Task)都要查询这张表，那我们通过Broadcast的方式就只需要往每个Executor把这张表发送一次就行了。Executor中的每个运行的Task查询这张唯一的表，而不是每次执行的时候都从Driver获得这张表！
2. **广播变量**：广播变量允许程序员将一个只读的变量缓存在每台机器上，而不用在任务之间传递变量。广播变量可被用于有效地给每个节点一个大输入数据集的副本。Spark还尝试使用高效地广播算法来分发变量，进而减少通信的开销。
3. 这就好比ServeletContext的具体作用，只是Broadcast是分布式的共享数据。默认情况下只要程序在运行Broadcast变量就会存在，因为Broadcast在底层是通过BlockManager管理的！应用程序存的时候就存在了。但是你可以手动指定或者配置具体周期来销毁Broadcast变量!
4. Broadcast一般用于处理共享配置文件，通用的数据集DataSet，常用的数据结构等待，但是不适合存在太大的数据在Broadcast。Broadcast不会内存溢出，因为其数据的保存的StorageLevel是MEMORY\_AND\_DISK的方式。虽然如此，我们也不可以放太大的数据在Broadcast中，因为网络IO和可能的单点压力会非常大。
5. Broadcast变量是只读变量。因为广播变量是共享变量，如果在一个节点上更新的话，其他节点就会更新，如果多节点更新的话，大家更新的顺序怎么办？最为轻松的保持了数据的一致性。
6. Broadcast的使用：

*\** {{{  
*\* scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))  
\* broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)  
\*  
\* scala> broadcastVar.value  
\* res0: Array[Int] = Array(1, 2, 3)  
\** }}}

1. Broadcast的实现？

分为两种：HttpBroadcast，TorrentBroadcast

**HttpBroadcast**:最开始的时候放在Driver的本地文件中。Driver在本地会创建一个文件夹来存放Broadcast中的data，然后启动HttpServer来访问文件夹中的数据，同时写入到BlockManager(StorageLevel是MEMORY\_AND\_DISK)中获得BlockId(BroadcastBlockId)，当第一次Executor中的Task要访问Broadcast要访问变量的时候会向Driver通过HttpServer来访问数据，然后会在Executor中的BlockManager中注册该Broadcast中的数据给BlockManager，这样之后续的Task需要访问Broadcast的变量的时候会首先查询BlockManager中有没有该数据，如果有则直接访问。

**HttpBroadcast存在的问题：**

**单点故障的问题(都向Driver获得**BlockId**)。**

**大家都到该节点要数据，存在网络瓶颈**

1. BroadcastManager是用来管理Broadcast的。该实例对象是在SparkEnv的时候创建的。Broadcast中的initialize()是**构造方法**是一定会被调用的。

// Called by SparkContext or Executor before using Broadcast  
**private def** initialize() {  
 synchronized {  
 **if** (!*initialized*) {  
 **val** broadcastFactoryClass = //来获得broadcastFactory是哪一个  
 conf.get("spark.broadcast.factory", "org.apache.spark.broadcast.TorrentBroadcastFactory")  
  
 *broadcastFactory* =  
 Utils.*classForName*(broadcastFactoryClass).newInstance.asInstanceOf[BroadcastFactory]  
  
 // Initialize appropriate BroadcastFactory and BroadcastObject  
 *broadcastFactory*.initialize(isDriver, conf, securityManager)  
  
 *initialized* = **true** }  
 }  
}

在实例化BroadcastManager的时候会创建BroadcastFactory工厂来构建具体实际的Broadcast类型，默认情况下是TorrentBroadcastFactory;

**为啥默认是TorrentBroadcastFactory？**

1. HttpBroadcast存在单点故障，和网络IO性能问题，所以默认使用TorrentBroadcast的方式，开始的时候数据在Driver中，假设A节点用了这个数据，A节点也会作为一个数据的来源和供应源，这个时候你再需要的时候就有两个供应源，以此类推，当你用的节点越多的时候，你的数据节点来源就越多。你获取数据的选择性就越多。用的越多，速度就越快。
2. TorrentBroadcast安照BLOCK\_SIZE(默认是4MB)将Broadcast中的数据划分为不同的Block，然后将分块信息也就是meta信息存放到Driver的BlockManager中，同时会告诉Driver中的BlockManagerMaster(在BlockManagerMaster中的信息都是全局级别的，这时候Driver和Worker都可以访问)说明meta信息存放完毕，在这之后每个分块的dataBlock都会存放在Driver中的BlockManager中，也会告诉BlockManagerMaster说数据存放好了。
3. 当Executor从Driver端拿到数据的时候，当采用TorrentBroadcast的方式的时候，会通知BlockManagerMaster数据多了一份副本，下一个Task获取数据的时候，就有两种选择了，但是会优先从本地获取，所以会先查看本地BlockManager中有没有这个数据，如果有直接读取，如果是远程的话就采用Shuffle的方式.
4. BT:耗带宽

程序执行流程：

下面从源码角度看broadcast三方面：初始化，创建，使用

**初始化：**

1. Spark初始化的时候创建SparkEnv对象env，初始化Env配置环境时创建BroadcastManager对象。

// NB: blockManager is not valid until initialize() is called later.  
**val** blockManager = **new** BlockManager(executorId, rpcEnv, blockManagerMaster,  
 serializer, conf, memoryManager, mapOutputTracker, shuffleManager,  
 blockTransferService, securityManager, numUsableCores)  
  
**val** broadcastManager = **new** BroadcastManager(isDriver, conf, securityManager)  
  
**val** cacheManager = **new** CacheManager(blockManager)

1. 调用BroadcastManager.initialize()方法

initialize()  
  
// Called by SparkContext or Executor before using Broadcast  
**private def** initialize() {  
 synchronized {  
 **if** (!*initialized*) { //只有第一次初始化才进入  
 **val** broadcastFactoryClass = //创建broadcastFactory  
 conf.get("spark.broadcast.factory", "org.apache.spark.broadcast.TorrentBroadcastFactory")//默认是TB  
  
 *broadcastFactory* =  
 Utils.*classForName*(broadcastFactoryClass).newInstance.asInstanceOf[BroadcastFactory]  
 //初始化*broadcastFactory，调用initialize，分为两种*  
 // Initialize appropriate BroadcastFactory and BroadcastObject  
 *broadcastFactory*.initialize(isDriver, conf, securityManager)  
  
 *initialized* = **true** }  
 }  
}

这里就要分两种讨论了，HttpBroadcast.initialize()，TorrentBroadcast.initialize()

1. HttpBroadcast.initialize()方法：做两件事：在Driver端创建createServer，创建一个MeradataCleaner对象。

**def** initialize(isDriver: Boolean, conf: SparkConf, securityMgr: SecurityManager) {  
 synchronized {  
 **if** (!*initialized*) {  
 *bufferSize* = conf.getInt("spark.buffer.size", 65536)  
 *compress* = conf.getBoolean("spark.broadcast.compress", **true**)  
 *securityManager* = securityMgr  
 **if** (isDriver) {  
 *createServer*(conf)//调用createServer方法，在Driver创建  
 conf.set("spark.httpBroadcast.uri", *serverUri*)  
 }  
 *serverUri* = conf.get("spark.httpBroadcast.uri")

**//创建MetadataCleaner对象**  
 *cleaner* = **new** MetadataCleaner(MetadataCleanerType.*HTTP\_BROADCAST*, *cleanup*, conf)  
 *compressionCodec* = CompressionCodec.*createCodec*(conf)  
 *initialized* = **true** }  
 }  
}

createServer方法：在Driver端创建，其内部创建了HttpServer实例，其用于访问Driver在本地文件系统中创建的文件夹中的数据。

**private def** createServer(conf: SparkConf) {  
 *broadcastDir* = Utils.*createTempDir*(Utils.*getLocalDir*(conf), "broadcast")  
 **val** broadcastPort = conf.getInt("spark.broadcast.port", 0)  
 *server* =

**//创建HttpServer**  
 **new** HttpServer(conf, *broadcastDir*, *securityManager*, broadcastPort, "HTTP broadcast server")  
 *server*.start()//启动HttpServer  
 *serverUri* = *server*.uri  
 logInfo("Broadcast server started at " + *serverUri*)  
}

1. 而T

**override def** initialize(isDriver: Boolean, conf: SparkConf, securityMgr: SecurityManager) { }

**创建：**

1. SparkCotext在初始化的时候调用broadcast方法。

*/\*\*  
 \* Broadcast a read-only variable to the cluster, returning a  
 \** [[*org.apache.spark.broadcast.Broadcast*]] *object for reading it in distributed functions.  
 \* The variable will be sent to each cluster only once.  
 \*/***def** broadcast[T: ClassTag](value: T): Broadcast[T] = {  
 assertNotStopped()  
 **if** (*classOf*[RDD[\_]].isAssignableFrom(*classTag*[T].runtimeClass)) {  
 // This is a warning instead of an exception in order to avoid breaking user programs that  
 // might have created RDD broadcast variables but not used them:  
 logWarning("Can not directly broadcast RDDs; instead, call collect() and "  
 + "broadcast the result (see SPARK-5063)")  
 }

*//创建newBroadcast实例*  
 **val** bc = env.broadcastManager.newBroadcast[T](value, isLocal)  
 **val** callSite = getCallSite  
 logInfo("Created broadcast " + bc.id + " from " + callSite.shortForm)  
 cleaner.foreach(\_.registerBroadcastForCleanup(bc))  
 bc  
}

BlockManager.newBroadcast -> BroadcastFactory.newBroadcast最后传递到HttpBroadcastFactory中，返回HttpBroadcast对象。

**override def** newBroadcast[T: ClassTag](value\_ : T, isLocal: Boolean, id: Long): Broadcast[T] =  
 **new** HttpBroadcast[T](value\_, isLocal, id)

* 1. 此时的HttpBroadcast中主要做了两件事：

1. 将blockId放到BlockManager中，但并不通知BlockManagerMaster
2. 调用write方法。

/\*  
 \* Broadcasted data is also stored in the BlockManager of the driver. The BlockManagerMaster  
 \* does not need to be told about this block as not only need to know about this data block.  
 \*/

//将blockId放到BlockManager中，但并不通知BlockManagerMaster  
HttpBroadcast.synchronized {  
 SparkEnv.*get*.blockManager.putSingle(  
 *blockId*, value\_, StorageLevel.*MEMORY\_AND\_DISK*, tellMaster = **false**)  
}  
//调用write方法  
**if** (!isLocal) {  
 HttpBroadcast.*write*(id, value\_)  
}

**Write方法源码如下：**

**private def** write(id: Long, value: Any) {  
 **val** file = *getFile*(id)  
 **val** fileOutputStream = **new** FileOutputStream(file)  
 Utils.*tryWithSafeFinally* {  
 **val** out: OutputStream = {  
 **if** (*compress*) {

**//包装压缩方式**  
 *compressionCodec*.compressedOutputStream(fileOutputStream)  
 } **else** {  
 **new** BufferedOutputStream(fileOutputStream, *bufferSize*)  
 }  
 }

**//序列化的将文件写入到指定的文件中。**  
 **val** ser = SparkEnv.*get*.serializer.newInstance()  
 **val** serOut = ser.serializeStream(out)  
 Utils.*tryWithSafeFinally* {  
 serOut.writeObject(value)  
 } {  
 serOut.close()  
 }  
 *files* += file  
 } {  
 fileOutputStream.close()  
 }  
}

* + 1. **TorrentBroadcastFactory的newBroadcast方法**

**override def** newBroadcast[T: ClassTag](value\_ : T, isLocal: Boolean, id: Long): Broadcast[T] = {  
 **new** TorrentBroadcast[T](value\_, id)  
}

1. **返回TorrentBroadcast对象，通过writeBlocks将值交给BlockManager进行注册，并序列化到本地存储。**

*/\*\*  
 \* Divide the object into multiple blocks and put those blocks in the block manager.  
 \** ***@param value*** *the object to divide  
 \** ***@return*** *number of blocks this broadcast variable is divided into  
 \*/***private def** writeBlocks(value: T): Int = {  
 // Store a copy of the broadcast variable in the driver so that tasks run on the driver  
 // do not create a duplicate copy of the broadcast variable's value.

//将BlockId注册到BlockManager中，并不告诉BlockManagerMaster  
 SparkEnv.*get*.blockManager.putSingle(*broadcastId*, value, StorageLevel.*MEMORY\_AND\_DISK*,  
 tellMaster = **false**)  
 **val** blocks =

// *blockifyObject*数据切分的方法。  
 TorrentBroadcast.*blockifyObject*(value, *blockSize*, SparkEnv.*get*.serializer, *compressionCodec*)  
 blocks.zipWithIndex.foreach { **case** (block, i) =>  
//通过putBytes将值序列化到本地存储。

SparkEnv.*get*.blockManager.putBytes(  
 *BroadcastBlockId*(id, "piece" + i),  
 block,  
 StorageLevel.*MEMORY\_AND\_DISK\_SER*, //存储级别  
 tellMaster = **true**) //并将存储好的值告诉BlockManagerMaster  
 }  
 blocks.length  
}

**补充：**

blockifyObject方法是完成Block数据切分.

**def** blockifyObject[T: ClassTag](  
 obj: T,  
 blockSize: Int, //切分后的大小  
 serializer: Serializer,  
 compressionCodec: Option[CompressionCodec]): Array[ByteBuffer] = {

**//定义输出流**  
 **val** bos = **new** ByteArrayChunkOutputStream(blockSize)

**//包装压缩方式**  
 **val** out: OutputStream = compressionCodec.map(c => c.compressedOutputStream(bos)).getOrElse(bos)

**//创建序列化对象**  
 **val** ser = serializer.newInstance()

**//包装序列化输出流**  
 **val** serOut = ser.serializeStream(out)

//将值写到ByteArray中  
 serOut.writeObject[T](obj).close()  
 bos.toArrays.map(ByteBuffer.*wrap*)  
}

1. putBykeys方法源码如下：

*/\*\*  
 \* Put a new block of serialized bytes to the block manager.  
 \* Return a list of blocks updated as a result of this put.  
 \*/***def** putBytes(  
 blockId: BlockId, //blockId  
 bytes: ByteBuffer, //block大小，默认是4MB  
 level: StorageLevel, //存储级别，默认是MEMONY\_AND\_DISK  
 tellMaster: Boolean = **true**, //默认序列化到BlockManager中是要告诉//BlockManagerMaster  
 effectiveStorageLevel: Option[StorageLevel] = None): Seq[(BlockId, BlockStatus)] = {  
 *require*(bytes != **null**, "Bytes is null")  
 doPut(blockId, *ByteBufferValues*(bytes), level, tellMaster, effectiveStorageLevel)  
}

**使用：**

1. 当Executor从Driver端拿到数据的时候，当采用TorrentBroadcast的方式的时候，会通知BlockManagerMaster数据多了一份副本，下一个Task获取数据的时候，就有两种选择了，但是会优先从本地获取，所以会先查看本地BlockManager中有没有这个数据，如果有直接读取，如果是远程的话就采用Shuffle的方式.
2. TorrentBroadcast.readBlocks()读取Block

*/\*\* Fetch torrent blocks from the driver and/or other executors. \*/***private def** readBlocks(): Array[ByteBuffer] = {  
 // Fetch chunks of data. Note that all these chunks are stored in the BlockManager and reported  
 // to the driver, so other executors can pull these chunks from this executor as well.

*//定义数据块集合*  
 **val** blocks = **new** Array[ByteBuffer](*numBlocks*)

*//引用*blockManager  
 **val** bm = SparkEnv.*get*.blockManager  
//*循环遍历所有块，避免同时访问一个快，*随机读取，为了负载均衡  
 **for** (pid <- Random.shuffle(*Seq*.range(0, *numBlocks*))) {

*//构建Block快Id*  
 **val** pieceId = *BroadcastBlockId*(id, "piece" + pid)  
 logDebug(s"Reading piece **$**pieceId of **$***broadcastId*")  
 // First try getLocalBytes because there is a chance that previous attempts to fetch the  
 // broadcast blocks have already fetched some of the blocks. In that case, some blocks  
 // would be available locally (on this executor).

*//优先从本地读取*  
 **def** getLocal: Option[ByteBuffer] = bm.getLocalBytes(pieceId)

*//从远程读取*  
 **def** getRemote: Option[ByteBuffer] = bm.getRemoteBytes(pieceId).map { block =>  
 // If we found the block from remote executors/driver's BlockManager, put the block  
 // in this executor's BlockManager.

*//将数据序列化到本地文件系统中*  
 SparkEnv.*get*.blockManager.putBytes(  
 pieceId,  
 block,  
 StorageLevel.*MEMORY\_AND\_DISK\_SER*,  
 tellMaster = **true**)//然后告诉BlockManagerMaster，Block副本数增加  
 block  
 }  
 **val** block: ByteBuffer = getLocal.orElse(getRemote).getOrElse(  
 **throw new** SparkException(s"Failed to get **$**pieceId of **$***broadcastId*"))  
 blocks(pid) = block  
 }  
 blocks  
}