Broadcast

顾名思义，broadcast 就是将数据从一个节点发送到其他各个节点上去。这样的场景很多，比如 driver 上有一张表，其他节点上运行的 task 需要 lookup 这张表，那么 driver 可以先把这张表 copy 到这些节点，这样 task 就可以在本地查表了。如何实现一个可靠高效的 broadcast 机制是一个有挑战性的问题。先看看 Spark 官网上的一段话：

Broadcast variables allow the programmer to keep a **read-only** variable cached on each **machine** rather than shipping a copy of it with **tasks**. They can be used, for example, to give every node a copy of a **large input dataset** in an efficient manner. Spark also attempts to distribute broadcast variables using **efficient** broadcast algorithms to reduce communication cost.

问题：为什么只能 broadcast 只读的变量?

这就涉及一致性的问题，如果变量可以被更新，那么一旦变量被某个节点更新，其他节点要不要一块更新？如果多个节点同时在更新，更新顺序是什么？怎么做同步？还会涉及 fault-tolerance 的问题。为了避免维护数据一致性问题，Spark 目前只支持 broadcast 只读变量。

问题：broadcast 到节点而不是 broadcast 到每个 task?

因为每个 task 是一个线程，而且同在一个进程运行 tasks 都属于同一个 application。因此每个节点（executor）上放一份就可以被所有 task 共享。

问题： 具体怎么用 broadcast?

driver program 例子：

```
val data = List(1, 2, 3, 4, 5, 6)
val bdata = sc.broadcast(data)

val rdd = sc.parallelize(1 to 6, 2)
val observedSizes = rdd.map(_ => bdata.value.size)
```

driver 使用 `sc.broadcast()` 声明要 broadcast 的 data，bdata 的类型是 Broadcast。

当 `rdd.transformation(func)` 需要用 bdata 时，直接在 func 中调用，比如上面的例子中的 map() 就使用了 bdata.value.size。

问题：怎么实现 broadcast?

broadcast 的实现机制很有意思：

1. 分发 task 的时候先分发 bdata 的元信息

Driver 先建一个本地文件夹用以存放需要 broadcast 的 data，并启动一个可以访问该文件夹的 HttpServer。当调用 `val bdata = sc.broadcast(data)` 时就把 data 写入文件夹，同时写入 driver 自己的 blockManger 中（StorageLevel 为内存 + 磁盘），获得一个 blockId，类型为 BroadcastBlockId。当调用 `rdd.transformation(func)` 时，如果 func 用到了 bdata，那么 driver submitTask() 的时候会将 bdata 一同 func 进行序列化得到 serialized task，**注意序列化的时候不会序列化 bdata 中包含的 data**。上一章讲到 serialized task 从 driverActor 传递到 executor 时使用 Akka 的传消息机制，消息不能太大，而实际的 data 可能很大，所以这时候还不能 broadcast data。

> driver 为什么会同时将 data 放到磁盘和 blockManager 里面？放到磁盘是为了让 HttpServer 访问到，放到 blockManager 是为了让 driver program 自身使用 bdata 时方便（其实我觉得不放到 blockManger 里面也行）。

那么什么时候传送真正的 data? 在 executor 反序列化 task 的时候，会同时反序列化 task 中的 bdata 对象，这时候会调用 bdata 的 readObject() 方法。该方法先去本地 blockManager 那里询问 bdata 的 data 在不在 blockManager 里面，如果不在就使用下面的两种 fetch 方式之一去将 data fetch 过来。得到 data 后，将其存放到 blockManager 里面，这样后面运行的 task 如果需要 bdata 就不需要再去 fetch data 了。如果在，就直接拿来用了。

下面探讨 broadcast data 时候的两种实现方式：

2. HttpBroadcast

顾名思义，HttpBroadcast 就是每个 executor 通过的 http 协议连接 driver 并从 driver 那里 fetch data。

Driver 先准备好要 broadcast 的 data，调用 `sc.broadcast(data)` 后会调用工厂方法建立一个 HttpBroadcast 对象。该对象做的第一件事就是将 data 存到 driver 的 blockManager 里面，StorageLevel 为内存 + 磁盘，blockId 类型为 BroadcastBlockId。

同时 driver 也会将 broadcast 的 data 写到本地磁盘，例如写入后得到 `/var/folders/87/grpn1_fn4xq5wdqmxk31v0l00000gp/T/spark-6233b09c-3c72-4a4d-832b-6c0791d0eb9c/broadcast_0`，这个文件夹作为 HttpServer 的文件目录。

> Driver 和 executor 启动的时候，都会生成 broadcastManager 对象，调用 HttpBroadcast.initialize()，driver 会在本地建立一个临时目录用来存放 broadcast 的 data，并启动可以访问该目录的 httpServer。

Fetch data：在 executor 反序列化 task 的时候，会同时反序列化 task 中的 bdata 对象，这时候会调用 bdata 的 readObject() 方法。该方法先去本地 blockManager 那里询问 bdata 的 data 在不在 blockManager 里面，**如果不在就使用 http 协议连接 driver 上的 httpServer，将 data fetch 过来**。得到 data 后，将其存放到 blockManager 里面，这样后面运行的 task 如果需要 bdata 就不需要再去 fetch data 了。如果在，就直接拿来用了。

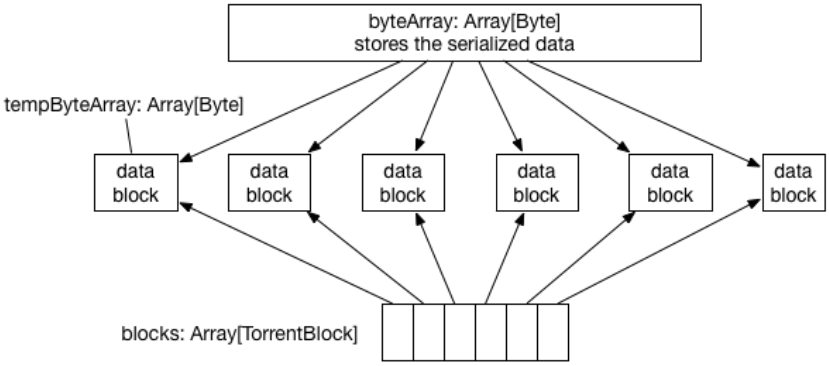
HttpBroadcast 最大的问题就是 **driver 所在的节点可能会出现网络拥堵**，因为 worker 上的 executor 都会去 driver 那里 fetch 数据。

3. TorrentBroadcast

为了解决 HttpBroadast 中 driver 单点网络瓶颈的问题，Spark 又设计了一种 broadcast 的方法称为 TorrentBroadcast，**这个类似于大家常用的 BitTorrent 技术**。基本思想就是将 data 分块成 data blocks，然后假设有 executor fetch 到了一些 data blocks，那么这个 executor 就可以被当作 data server 了，随着 fetch 的 executor 越来越多，有更多的 data server 加入，data 就很快能传播到全部的 executor 那里去了。

HttpBroadcast 是通过传统的 http 协议和 httpServer 去传 data，在 TorrentBroadcast 里面使用在上一章介绍的 blockManager.getRemote() => NIO ConnectionManager 传数据的方法来传递，读取数据的过程与读取 cached rdd 的方式类似，可以参阅 CacheAndCheckpoint 中的最后一张图。

下面讨论 TorrentBroadcast 的一些细节：



driver 端：

Driver 先把 data 序列化到 byteArray，然后切割成 BLOCK_SIZE（由 `spark.broadcast.blockSize = 4MB` 设置）大小的 data block，每个 data block 被 TorrentBlock 对象持有。切割完 byteArray 后，会将其回收，因此内存消耗虽然可以达到 2 * Size(data)，但这是暂时的。

完成分块切割后，就将分块信息（称为 meta 信息）存放到 driver 自己的 blockManager 里面，StorageLevel 为内存 + 磁盘，同时会通知 driver 自己的 blockManagerMaster 说 meta 信息已经存放好。**通知 blockManagerMaster 这一步很重要，因为 blockManagerMaster 可以被 driver 和所有 executor 访问到，信息被存放到 blockManagerMaster 就变成了全局信息。**

之后将每个分块 data block 存放到 driver 的 blockManager 里面，StorageLevel 为内存 + 磁盘。存放后仍然通知 blockManagerMaster 说 blocks 已经存放好。到这一步，driver 的任务已经完成。

Executor 端：

executor 收到 serialized task 后，先反序列化 task，这时候会反序列化 serialized task 中包含的 bdata 类型是 TorrentBroadcast，也就是去调用 TorrentBroadcast.readObject()。这个方法首先得到 bdata 对象，**然后发现 bdata 里面没有包含实际的 data。怎么办？**先询问所在的 executor 里的 blockManager 是会否包含 data（通过查询 data 的 broadcastId），包含就直接从本地 blockManager 读取 data。否则，就通过本地 blockManager 去连接 driver 的 blockManagerMaster 获取 data 分块的 meta 信息，获取信息后，就开始了 BT 过程。

BT 过程：`**task 先在本地开一个数组用于存放将要 fetch 过来的 data blocks arrayOfBlocks = new Array[TorrentBlock](totalBlocks)`，**TorrentBlock 是对 data block 的包装。然后打乱要 fetch 的 data blocks 的顺序，比如如果 data block 共有 5 个，那么打乱后的 fetch 顺序可能是 3-1-2-4-5。然后按照打乱后的顺序去 fetch 一个个 data block。fetch 的过程就是通过 “本地 blockManager – 本地 connectionManager – driver/executor 的 connectionManager – driver/executor 的 blockManager – data” 得到 data，这个过程与 fetch cached rdd 类似。每 fetch 到一个 block 就将其存放到 executor 的 blockManager 里面，同时通知 driver 上的 blockManagerMaster 说该 data block 多了一个存储地址。**这一步通知非常重要，意味着 blockManagerMaster 知道 data block 现在在 cluster 中由多份，下一个不同节点上的 task 再去 fetch 这个 data block 的时候，可以有两个选择了，而且会随机选择一个去 fetch。这个过程持续下去就是 BT 协议，随着下载的客户端越来越多，data block 服务器也越来越多，就变成 p2p下载了。关于 BT 协议，Wikipedia 上有一个动画。

整个 fetch 过程结束后，task 会开一个大 Array[Byte]，大小为 data 的总大小，然后将 data block 都 copy 到这个 Array，然后对 Array 中 bytes 进行反序列化得到原始的 data，这个过程就是 driver 序列化 data 的反过程。

最后将 data 存放到 task 所在 executor 的 blockManager 里面，StorageLevel 为内存 + 磁盘。显然，这时候 data 在 blockManager 里存了两份，不过等全部 executor 都 fetch 结束，存储 data blocks 那份可以删掉了。

问题：broadcast RDD 会怎样?

@Andrew-Xia 回答道：不会怎样，就是这个rdd在每个executor中实例化一份。

Discussion

公共数据的 broadcast 是很实用的功能，在 Hadoop 中使用 DistributedCache，比如常用的 `-libjars` 就是使用 DistributedCache 来将 task 依赖的 jars 分发到每个 task 的工作目录。不过分发前 DistributedCache 要先将文件上传到 HDFS。这种方式的主要问题是**资源浪费**，如果某个节点上要运行来自同一 job 的 4 个 mapper，那么公共数据会在该节点上存在 4 份（每个 task 的工作目录会有一份）。但是通过 HDFS 进行 broadcast 的好处在于**单点瓶颈不明显**，因为公共 data 首先被分成多个 block，然后不同的 block 存放在不同的节点。这样，只要所有的 task 不是同时去同一个节点 fetch 同一个 block，网络拥塞不会很严重。

对于 Spark 来讲，broadcast 时考虑的不仅是如何将公共 data 分发下去的问题，还要考虑如何让同一节点上的 task 共享 data。

对于第一个问题，Spark 设计了两种 broadcast 的方式，传统存在单点瓶颈问题的 HttpBroadcast，和类似 BT 方式的 TorrentBroadcast。HttpBroadcast 使用传统的 client-server 形式的 HttpServer 来传递真正的 data，而 TorrentBroadcast 使用 blockManager 自带的 NIO 通信方式来传递 data。TorrentBroadcast 存在的问题是**慢启动**和**占内存**，慢启动指的是刚开始 data 只在 driver 上有，要等 executors fetch 很多轮 data block 后，data server 才会变得可观，后面的 fetch 速度才会变快。占内存executor 在 fetch 完 data blocks 后进行反序列化时需要将近两倍 data size 的内存消耗。不管那一种方式，driver 在分块时会有两倍 data size 的内存消耗。

对于第二个问题，每个 executor 都包含一个 blockManager 用来管理存放在 executor 里的数据，将公共数据存放在 blockManager 中（StorageLevel 为内存 + 磁盘），可以保证在 executor 执行的 tasks 能够共享 data。

其实 Spark 之前还尝试了一种称为 TreeBroadcast 的机制，详情可以见技术报告 Performance and Scalability of Broadcast in Spark。

更深入点，broadcast 可以用多播协议来做，不过多播使用 UDP，不是可靠的，仍然需要应用层的设计一些可靠性保障机制。