源码逻辑：

class JobSubmitter{

List<InputSplit> splits = input.getSplits(job);

}

public abstract class CombineFileInputFormat<K, V> extends FileInputFormat<K, V>{

getMoreSplits(job, stats, maxSize, minSizeNode, minSizeRack, splits);

files = new OneFileInfo[stats.size()];

for (FileStatus stat : stats) {

files[i] = new OneFileInfo(stat, conf, isSplitable(job, stat.getPath()),

rackToBlocks, blockToNodes, nodeToBlocks,

rackToNodes, maxSize);

-----对每一个读入文件调用：

do {

if (maxSize == 0) {

myLength = left;

} else {

if (left > maxSize && left < 2 \* maxSize) {

// if remainder is between max and 2\*max - then

// instead of creating splits of size max, left-max we

// create splits of size left/2 and left/2. This is

// a heuristic to avoid creating really really small

// splits.

myLength = left / 2;

} else {

myLength = Math.min(maxSize, left);

}

}

OneBlockInfo oneblock = new OneBlockInfo(stat.getPath(),

myOffset, myLength, locations[i].getHosts(),

locations[i].getTopologyPaths());

left -= myLength;

myOffset += myLength;

blocksList.add(oneblock);

} while (left > 0);

-------

totLength += files[i].getLength();

}

createSplits(nodeToBlocks, blockToNodes, rackToBlocks, totLength,

maxSize, minSizeNode, minSizeRack, splits);

if (maxSize != 0 && curSplitSize >= maxSize) {

// create an input split and add it to the splits array

addCreatedSplit(splits, Collections.singleton(node), validBlocks);

totalLength -= curSplitSize;

curSplitSize = 0;

splitsPerNode.add(node);

// Remove entries from blocksInNode so that we don't walk these

// again.

blocksInCurrentNode.removeAll(validBlocks);

validBlocks.clear();

// Done creating a single split for this node. Move on to the next

// node so that splits are distributed across nodes.

break;

}

}

}

切片逻辑：

CombineTextInputFormat用于小文件过多的场景，生成切片过程包括读入文件形成虚拟存储划分和切片2部分

虚拟存储根据设置的值CombineTextInputFormat.setMaxInputSplitSize(job, 4194304);// 4m

来进行文件划分。

第一部分读入文件大小并产生虚拟存储的过程，核心逻辑为：

for (FileStatus stat : stats) {

files[i] = new OneFileInfo(stat, conf, isSplitable(job, stat.getPath()),

rackToBlocks, blockToNodes, nodeToBlocks,

rackToNodes, maxSize);

totLength += files[i].getLength();

}

将输入目录下所有文件循环读入，记录文件大小，此时看做一个大的文件。

在读入文件的过程中会对读入的每一个文件的进行虚拟存储划分，核心逻辑为：

do {

if (maxSize == 0) {

myLength = left;

} else {

if (left > maxSize && left < 2 \* maxSize) {

// if remainder is between max and 2\*max - then

// instead of creating splits of size max, left-max we

// create splits of size left/2 and left/2. This is

// a heuristic to avoid creating really really small

// splits.

myLength = left / 2;

} else {

myLength = Math.min(maxSize, left);

}

}

OneBlockInfo oneblock = new OneBlockInfo(stat.getPath(),

myOffset, myLength, locations[i].getHosts(),

locations[i].getTopologyPaths());

left -= myLength;

myOffset += myLength;

blocksList.add(oneblock);

} while (left > 0);

根据是否设置setMaxInputSplitSize值，将每个文件划分成一个一个setMaxInputSplitSize值大小的文件。

注意：当剩余数据大小超过setMaxInputSplitSize值且不大于2倍setMaxInputSplitSize值，此时将文件均分成2个虚拟存储块（防止出现过小切片）

例如setMaxInputSplitSize值为4M，最后剩余的文件大小为4.02M，如果按照4M逻辑划分，就会出现0.02M的小的虚拟存储文件。

第二部分为切片阶段，核心逻辑为：

curSplitSize += oneblock.length;

if (maxSize != 0 && curSplitSize >= maxSize) {

// create an input split and add it to the splits array

addCreatedSplit(splits, Collections.singleton(node), validBlocks);

totalLength -= curSplitSize;

curSplitSize = 0;

splitsPerNode.add(node);

// Remove entries from blocksInNode so that we don't walk these

// again.

blocksInCurrentNode.removeAll(validBlocks);

validBlocks.clear();

// Done creating a single split for this node. Move on to the next

// node so that splits are distributed across nodes.

break;

}

curSplitSize += oneblock.length;这一步将刚才形成的一个个虚拟存储的文件块读入

if()逻辑是真正切片过程。判断虚拟存储的文件大小是否大于setMaxInputSplitSize值，大于等于则单独形成一个切片。

如果不大于则跟下一个虚拟存储文件进行合并，共同形成一个切片。

测试举例：有4个小文件大小分别为1.7M、5.1M、3.4M以及6.8M这四个小文件，则虚拟存储之后形成6个文件块，大小分别为：

1.7M，（2.05M、2.05M）、3.4M以及（3.4M、3.4M）

最终会形成3个切片，大小分别为（1.7+2.05）M、（2.05+3.4）M、（3.4+3.4）M