HDFS副本放置策略

在HDFS中将每个文件进行分块存储，数据块保存多个副本来保证可用性，同时能够提高性能：

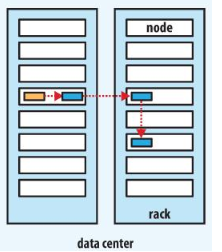
1. 文件分块存储之后按照数据块来读，提高文件随机读和并发读的性能
2. 保证数据块多副本到不同机器节点实现可靠性的同时，也提高同一数据块的并发读效率
3. 数据分块契合MapReduce的任务切分的思想

当前副本放置策略，有两大继承子类：BlockPlacementPolicyDefault及PolicyWithNodeGroup，其继承关系如下图所示：



# BlockPlacementPolicyDefault

默认使用的最经典的BlockPlacementPolicyDefault类，3副本存放的策略如下图：



其基本思想是：

* 第一副本在复制和client哪里node于（假设client它不是群集的范围内，则这第一个node是随机选取的。当然系统会尝试不选择哪些太满或者太忙的node）
* 第二个副本放置在与第一个节点不同的机架中的node中（随机选择）
* 第三个副本和第二个在同一个机架，随机放在不同的node中

策略的核心方法chooseTargets

*DatanodeStorageInfo[] chooseTarget(*

*String src,*

*int numOfReplicas,*

*Node writer,*

*Set<Node> excludedNodes,*

*long blocksize,*

*List<DatanodeDescriptor> favoredNodes,*

*BlockStoragePolicy storagePolicy)*

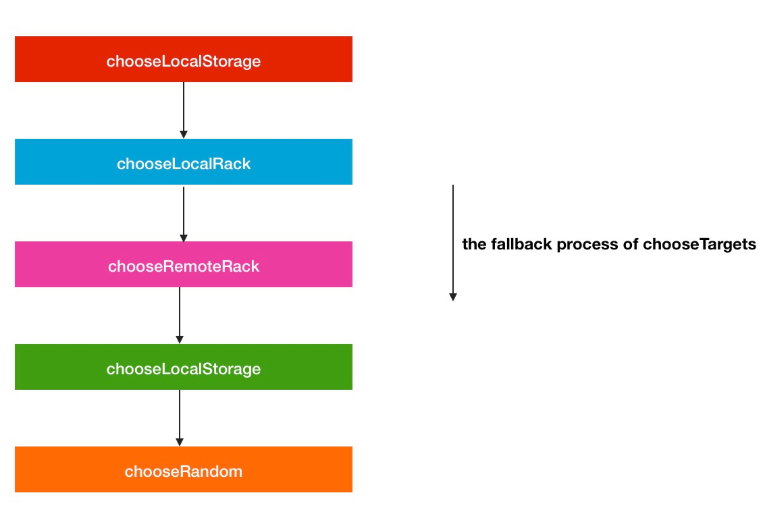
执行时如果配置favoredNodes，则优选选取favoredNodes中的节点，其执行流程如下：

1. 初始化操作

*//如果目标完成副本数为0或则主机节点数量为0，则返回null*

*if (numOfReplicas == 0 || clusterMap.getNumOfLeaves()==0) {  
 return DatanodeStorageInfo.EMPTY\_ARRAY;  
}  
//创建黑名单类表  
if (excludedNodes == null) {  
 excludedNodes = new HashSet<>();  
}  
//计算每个机架允许的最大副本数  
int[] result = getMaxNodesPerRack(chosenStorage.size(), numOfReplicas);  
numOfReplicas = result[0];  
int maxNodesPerRack = result[1];*

1. 选择目标节点，流程如下图：



*inal int numOfResults = results.size();*

*if (numOfResults == 0) {*

*//1.选择本地节点*

*writer = chooseLocalStorage(writer, excludedNodes, blocksize,*

*maxNodesPerRack, results, avoidStaleNodes, storageTypes, true)*

*.getDatanodeDescriptor();*

*.....*

*}*

*final DatanodeDescriptor dn0 = results.get(0).getDatanodeDescriptor();*

*if (numOfResults <= 1) {*

*//2.选择remote机架上的节点*

*chooseRemoteRack(1, dn0, excludedNodes, blocksize, maxNodesPerRack,*

*results, avoidStaleNodes, storageTypes);*

*if (--numOfReplicas == 0) {*

*return writer;*

*}*

*}*

*if (numOfResults <= 2) {*

*final DatanodeDescriptor dn1 = results.get(1).getDatanodeDescriptor();*

*if (clusterMap.isOnSameRack(dn0, dn1)) {*

*chooseRemoteRack(1, dn0, excludedNodes, blocksize, maxNodesPerRack,*

*results, avoidStaleNodes, storageTypes);*

*} else if (newBlock){*

*chooseLocalRack(dn1, excludedNodes, blocksize, maxNodesPerRack,*

*results, avoidStaleNodes, storageTypes);*

*} else {*

*//3.选择与第二个节点相同机架上的节点*

*chooseLocalRack(writer, excludedNodes, blocksize, maxNodesPerRack,*

*results, avoidStaleNodes, storageTypes);*

*}*

*if (--numOfReplicas == 0) {*

*return writer;*

*}*

*}*

*//4. 副本数超过3个，剩余副本则随机选择*

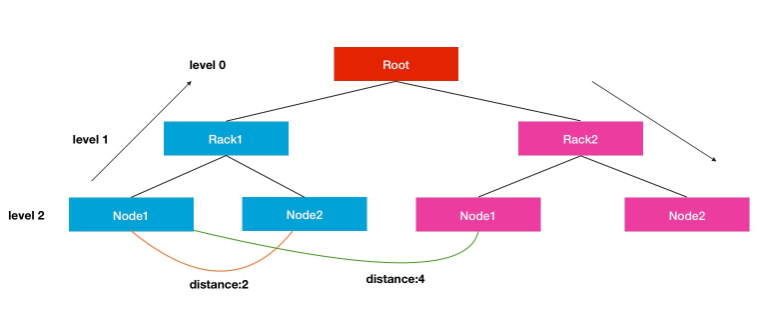
*chooseRandom(numOfReplicas, NodeBase.ROOT, excludedNodes, blocksize,*

*maxNodesPerRack, results, avoidStaleNodes, storageTypes);*

*return writer;*

1. 目标节点列表的排序，形成pipeline

Pipe的处理就是从Writer所在节点开始，寻找相对目标最短的目标节点，最终形成pipeline，在Hadoop上网络拓扑结构如下图所示：



最短路径的计算时TSP旅行商问题，源码实现如下：

*private DatanodeStorageInfo[] getPipeline(Node writer,  
 DatanodeStorageInfo[] storages) {  
 if (storages.length == 0) {  
 return storages;  
 }  
  
 synchronized(clusterMap) {  
 int index=0;*

*//首先如果writer请求本身不是dn，则默认选取第一个dn作为起始节点  
 if (writer == null || !clusterMap.contains(writer)) {  
 writer = storages[0].getDatanodeDescriptor();  
 }  
 for(; index < storages.length; index++) {*

*//获取当前idnex下标所属的storage作为最近目标的storage  
 DatanodeStorageInfo shortestStorage = storages[index];*

*//计算当前距离  
 int shortestDistance = clusterMap.getDistance(writer,  
 shortestStorage.getDatanodeDescriptor());  
 int shortestIndex = index;  
 for(int i = index + 1; i < storages.length; i++) {*

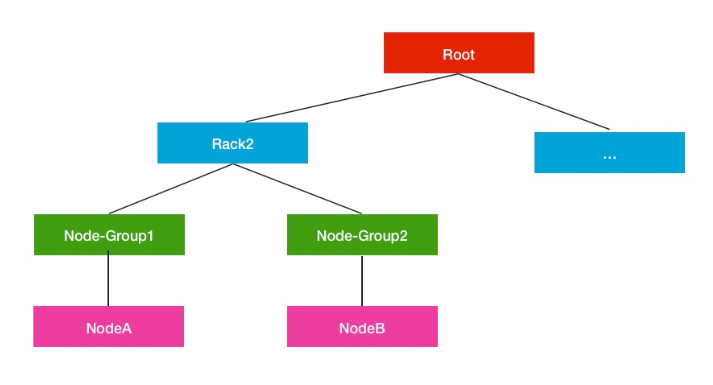
*//遍历计算后的距离  
 int currentDistance = clusterMap.getDistance(writer,  
 storages[i].getDatanodeDescriptor());  
 if (shortestDistance>currentDistance) {  
 shortestDistance = currentDistance;  
 shortestStorage = storages[i];  
 shortestIndex = i;  
 }  
 }  
 //找到新的最新距离，并进行下标替换  
 if (index != shortestIndex) {  
 storages[shortestIndex] = storages[index];  
 storages[index] = shortestStorage;  
 }  
 writer = shortestStorage.getDatanodeDescriptor();  
 }  
 }  
 return storages;  
}*

选出一个源节点,根据这个节点,遍历当前可选的下一个目标节点,找出一个最短距离的节点,作为下一轮选举的源节点,这样每2个节点之间的距离总是最近的,于是整个pipeline节点间的距离和就保证是足够小的了.那么现在另外一个问题还没有解决,如何定义和计算2个节点直接的距离,就是下面这行代码

*clusterMap.getDistance(writer, shortestStorage.getDatanodeDescriptor());*

# BlockPlacementPolicyWithNodeGroup继承类

BlockPlacementPolicyWithNodeGroup是BlockPlacementPolicyDefault的继承子类，与前者原理十分类似，前者从机架是否相同进行Node-Group的判断。在网络拓扑中，Rack机架层还多了Node-Group层，结果如下图所示：



与父类的逻辑没有很多差别，如下图所示：



差别在于第三个副本选择同一机架不同NodeGroup上的节点。

# AvailabeSpaceBlockPlacementPolicy

以上介绍的的副本策略随机选择节点，会有以下问题：

* 当集群中有异构节点（磁盘空间不同）可能造成节点之间负载不均衡
* 集群扩容时，节点负载不均衡，原数据节点空间使用率高于新增节点

AvailableSpaceBlockPlacementPolicy策略倾向于选择磁盘使用率最低的节点，其实现是覆盖chooseDataNode方法，核心原理代码如下所示：

*protected DatanodeDescriptor chooseDataNode(final String scope,  
 final Collection<Node> excludedNode) {  
 DatanodeDescriptor a =*

*DatanodeDescriptor) clusterMap.chooseRandom(scope, excludedNode);  
 DatanodeDescriptor b =  
 (DatanodeDescriptor) clusterMap.chooseRandom(scope, excludedNode);  
 return select(a, b);  
}*

从集群中随机选择两个节点，根据磁盘使用量选择使用率较低的节点

*protected int compareDataNode(final DatanodeDescriptor a,final DatanodeDescriptor b) {  
 if (a.equals(b) || Math.abs(a.getDfsUsedPercent() - b.getDfsUsedPercent()) < 5) {  
 return 0;  
 }  
 return a.getDfsUsedPercent() < b.getDfsUsedPercent() ? -1 : 1;  
}*

<https://blog.csdn.net/androidlushangderen/article/details/51178253>

主机多磁盘分摊磁盘IO负载相关配置

1. dfs.datanode.data.dir，配置节点多磁盘

*<property>  
    <name>dfs.datanode.data.dir</name>  
    <value>file:/data2/hadoop/hd\_space/dfs/data,file:/data3/hadoop/hd\_space/dfs/data</value>  
</property>*

1. dfs.datanode.fsdataset.volume.choosing.policy

数据副本存放磁盘的选择策略，有两种方式:

* RoundRobinVolumeChoosingPolicy，轮询方式
* AvailableSpaceVolumeSpaceVolumeChoosingPolicy，选择使用量最小的磁盘

AvailableSpaceBlockPlacementPolicy