Hadoop 集中式缓存管理

——Centralized Cache Management

Hadoop的计算框架借鉴了MapReduce思想：移动计算的成本远小于移动数据的成本，所以会尽可能将计算调度到拥有数据的节点上，计算和数据通常是在同一个DN节点上，即存在大量的本地读写。但是DFSClient在读取数据时不区分本地读和远程读，二者的实现方式完全一样，都是由DN读取然后，然后通过DFSClient与DN之间的Socket管道进行数据交互，这种实现方式经过DN的中转对数据的读性能有一定的影响。在Hadoop中通过两个方案来提升性能：

1. HDFS-2246， DFSClient和DN处于同一个节点，对数据发出读请求后，DN提供给DFSClient文件路径、偏移量和长度三元组，DFSClient根据这些信息直接从文件系统中直接数据数据，从而绕过DN中转数据的过程。但是存在的问题如下:

* HDFS需要为所有用户配置白名单，赋予其可读权限，增加新用户需要更新白名单，维护不方便
* 用户赋权后，意味着用户拥有了访问所有数据的权限，相当于超级用户，从而导致数据存在安全漏洞

1. HDFS-347，使用UNIX提供的Unix Domain Socket进程通讯机制实现安全的本地短路读取。DFSClient向DN请求数据时，DN打开块文件和元数据文件，通过Unix Domain Socket将对应的文件描述符传给DFSClient，而不再是路径、偏移量和长度三元组。文件描述符是只读的，DFSClient不能随意修改接收到的文件。由于DFSClient自身无法访问块所在的目录，也不能访问未授权的数据。虽然性能有了显著提升，但是从全集群上看，依然存在几个性能问题：

* DFSClient向DN发起数据读请求后，DN在OS Bufffer对数据进行Cache，但是数据的分布式情况不会显示提供给上层，对任务调度时透明的。
* 由于Cache的分布对任务调度透明，一些低优先级任务的读请求有可能将高优先级的任务正在使用的数据从Cache中淘汰出去，造成数据必须从磁盘读，增加读数据的开销从而影响任务的完成时间。

针对以上问题，提出了集中式缓存方案（HDFS-4949），由NN对DN的Cache进行统一集中管理，并将Cache接口显式暴露给上层应用。HDFS集中式缓存管理允许用户指定要缓存HDFS路径的缓存机制，在运行过程中NN会和保存需要数据的所有DN通信，并控制块数据的缓存。其具有几大优势：

* 用户可以根据自己的逻辑指定经常被使用的数据或者高优先级的任务对应的数据，让他们常驻内存而不被淘汰到磁盘。例如Hive/Impala构建的数据仓库中的fact表会频繁与其他表做JOIN计算，显然应该让该表常驻内存，这样DN内存使用紧张也不会把这些数据淘汰出去，同时实现对于Mixed Workloads情况下的SLA
* Centralized Cache由NN统一管理，可以根据Block在Cache中的分布情况去调度任务，该策略可以成为memory-locality
* 常规情况下，NN处理请求是根据拓扑远近决定去哪个DN读取数据，使用集中式缓存，可以加入Speed因素，当Block在DN中缓存时，客户端可以使用效率更高的Zero-Copy读API直接从内存中读取数据，由于从磁盘上加载数据时已经进行了数据校验，所以节省了读时间。
* 使用Centralized Cache可以提高集群的内存利用率，重复读取Block会将集群中所有的副本加载到内存中，使用集中式缓存，用户可以配置m（共n副本）副本到内存中，节省n-m个内存。

可以将频繁使用的文件加载到集中式缓存中，提高其使用效率。

# 概念及系统架构

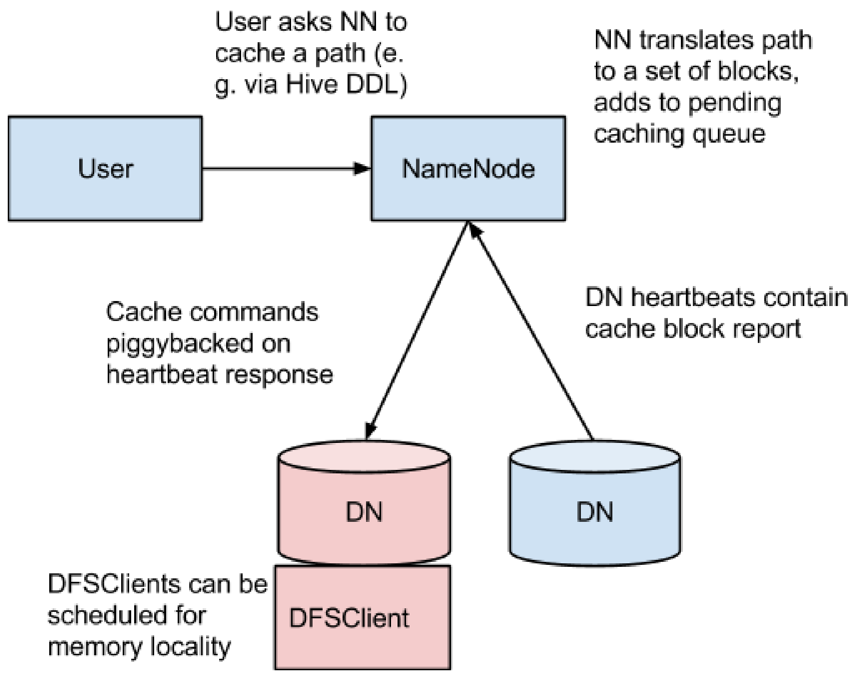
为了支持集中式缓存，引入了两个重要概念CacheDirective,CachePool：

1. CacheDirective，表示要被Cache到内存的文件或者目录，但是不支持嵌套目录。Directive支持的参数包括：

* Replication, Cache副本数目，如果多个Cache Director指向同一个文件
* Expiration, Cache过期时间（TTL）,目前该时间为绝对时间，但是不会自动缓存列表中删除，只是不再反馈给DFSClient进行读操作

1. CachePool，Cache管理单元，将属性类似的一组CacheDirective组成缓存池，在缓存池上可以进行权限、Quota、缓存策略和统计信息等灵活控制。CachePool上也可以进行资源管理，控制Directive的缓存在集群上的内存。

集中式缓存系统架构如下：



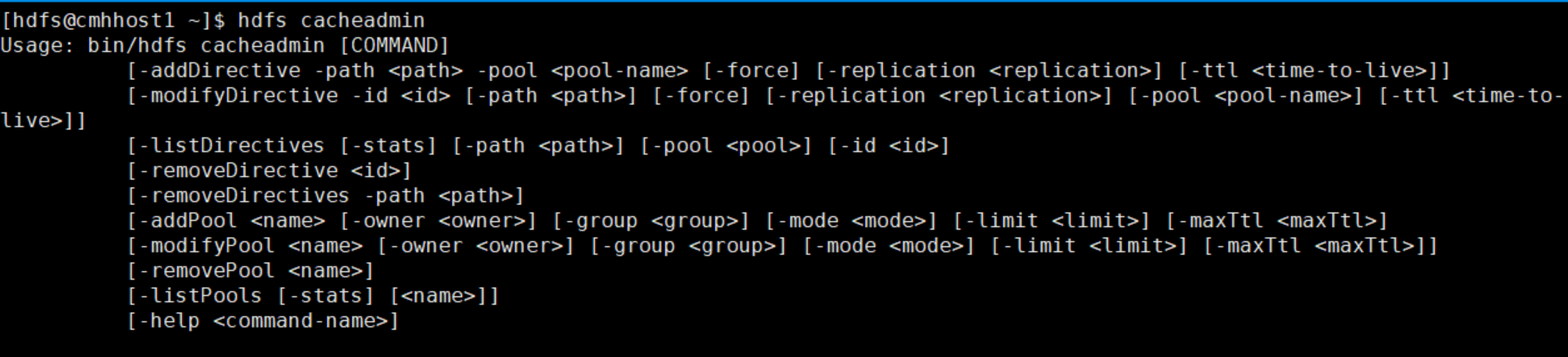
CacheDirective缓存的详细流程如上图所示：

1. 用户通过调用客户端接口向NN发起CacheDirective缓存请求
2. NN接收到缓存请求后，将CacheDirective转换成需要缓存的Block集合，并根据一定的策略并其加入到缓存队列中
3. NN接收到DN心跳后，将缓存Block的指令下发给DN
4. DN接收到NN发下的缓存指令后根据实际情况出发Native JNI进行系统调用，对Block数据进行实际缓存
5. 此后DN定期（默认10s）向NN进行缓存汇报，更新当前节点的缓存现状
6. 上层调度尽可能将任务调度到数据所在的DN，当客户端进行读数据请求时，通过DFSClient直接从内存进行ZeroCopy，从而显著提升性能

NameNode查询自身的缓存指令集来确定应该缓存那个路径，缓存指令存储在fsimage和edit日志中，而且可以通过java和命令行API添加、移除或修改。NN还存储了一组缓存池，用于把资源管理类和强制权限类的缓存指令进行分组的管理实体。

# 集中式缓存的使用

管理员和用户可以使用cacheadmin命令与缓存池进行交互，其指令格式如下：



1. addDirective，命令格式

*bin/hdfs cacheadmin [-addDirective*

*-path <path>*

*-pool <pool-name>*

*[-force]*

*[-replication <replication>]*

*[-ttl <time-to-live>]]*

参数描述如下：

|  |  |
| --- | --- |
| 参数 | 描述 |
| <path> | 要缓存的路径，该路径可以是文件夹或者文件 |
| <pool-name> | 要加入缓存指令的缓存池，必须修改缓存池的读写权限以便添加新的缓存指令 |
| --force | 不检查缓存池的资源限制 |
| <replication> | 要使用的缓存副本因子，默认为1 |
| <time-to-live> | 缓存指令可以有效保持多次时间，有效 |

1. removeDirective

*hdfs cacheadmin -removeDirective <id>*

|  |  |
| --- | --- |
| 参数 | 描述 |
| <id> | 要删除的缓存Directive指令的ID，必须拥有该Directive的写权限，以便删除它 |

1. removeDirectives

*hdfs cacheadmin -removeDirectives <path>*

|  |  |
| --- | --- |
| 参数 | 描述 |
| <path> | 要删除的缓存Directive的路径，拥有写权限，以便删除它 |

1. listDirectives

*hdfs cacheadmin -listDirectives [-stats] [-path <path>] [-pool <pool>]*

|  |  |
| --- | --- |
| 参数 | 描述 |
| <path> | 只列出带有该路径的缓存指令。注意，如果路径path在缓存池中有一条我们没有读权限的缓存指令，那么它就不会被读取 |
| <pool> | 只列出该缓存池的缓存Directive |
| -state | 列出基于path的缓存指令统计信息 |

1. addPool，命令格式：

*bin/hdfs cacheadmin*

*[-addPool <name>*

*[-owner <owner>]*

*[-group <group>]*

*[-mode <mode>]*

*[-limit <limit>]*

*[-maxTtl <maxTtl>]*

|  |  |
| --- | --- |
| 参数 | 描述 |
| <name> | 新缓冲池的名称 |
| <owner> | 该缓冲池所有者的名称，默认为当前用户 |
| <group> | 该缓冲池所属的组，默认为当前用户的主要组名 |
| <mode> | 以UNIX风格表示的该缓存池的权限，权限以八进制表示 |
| <limit> | 该缓存池由指令总计缓存的最大字节数，默认不设设置 |
| <maxTtl> | 添加到该缓存池的指令的最大生存时间。该值以秒，分，时，天的格式来表示，如120s，30m，4h，2d。有效单位为[smhd]。默认不设最大值。“never”表示没有限制 |

1. listPools，执行格式：

*hdfs cacheadmin -listPools [-stats] [<name>]*

|  |  |
| --- | --- |
| 参数 | 描述 |
| -stats | 显示额外的缓存池统计信息 |
| <name> | 若指定，则仅列出该缓存池的信息 |

1. removePool，执行格式：

*hdfs cacheadmin -removePool <name>*

|  |  |
| --- | --- |
| 参数 | 描述 |
| <name> | 要删除的缓存池的名称 |

1. modifyPool

*hdfs cacheadmin*

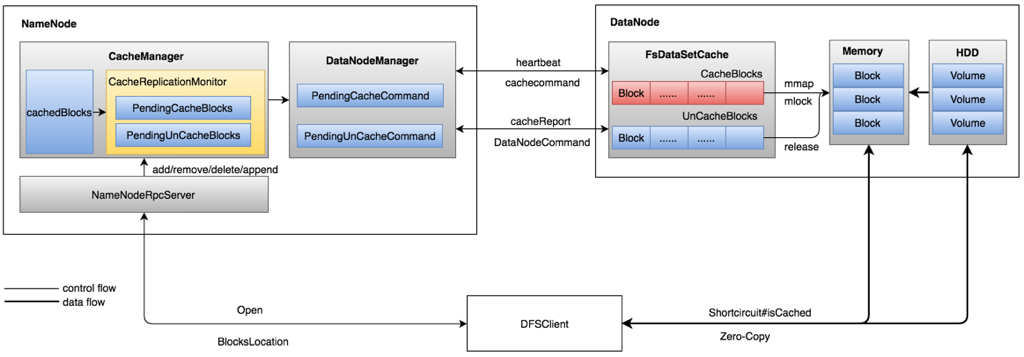
*-modifyPool <name> [-owner <owner>] [-group <group>] [-mode <mode>]*

*[-limit <limit>] [-maxTtl <maxTtl>]*

|  |  |
| --- | --- |
| 参数 | 描述 |
| <name> | 要修改的缓冲池的名称 |
| <owner> | 该缓冲池所有者的名 |
| <group> | 该缓冲池所属的组 |
| <mode> | 以UNIX风格表示的该缓存池的权限，权限以八进制表示 |
| <limit> | 该缓存池要缓存的最大字节数 |
| <maxTtl> | 添加到该缓存池的指令的最大生存时间。该值以秒，分，时，天的格式来表示，如120s，30m，4h，2d。有效单位为[smhd]。默认不设最大值。“never”表示没有限制 |

# 集中式缓存的源码分析

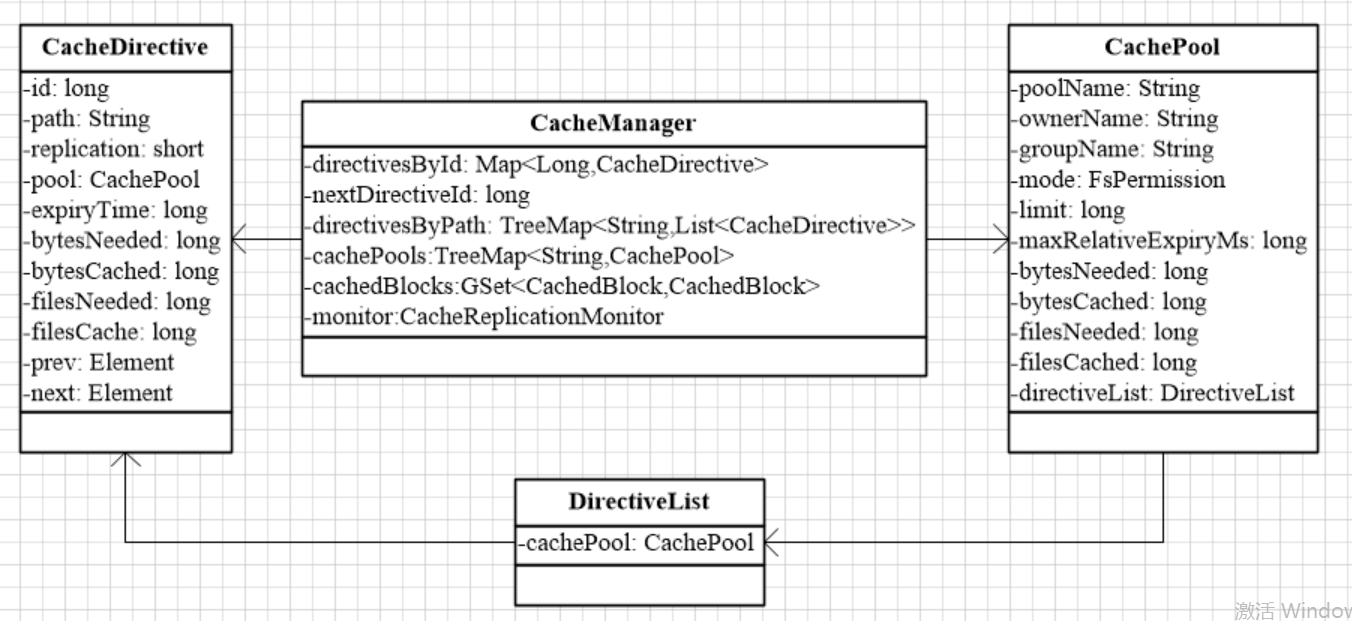
集中式缓存涉及到的NN和DN两侧的管理和实现如下图所示：



NN对数据缓存进行统一集中管理，并根据策略调度合适的DN对具体的数据进行缓存和置换，DN根据NN指令对数据的实际缓存和置换。

## NameNode

NameNode通过CacheManager实现整个集中式缓存在管理段的复杂处理逻辑，其相关类图如下：



在FSNameSystem中通过BlockManager负责管理分布在DN上的Block replica，而CacheManager则负责管理各个DN上的block cache。

DFSClient给NN发送addCacheDirective的RPC请求后，把这个需要被缓存的Path包装成CacheDirective加入CacheManager所管理的directivesByPath中，但是并没有加入cache到内存。一旦CacheMananger那边添加了新的CacheDirective，触发CacheReplicationMonitor的rescan来扫描并将需要缓存的block加入到cacheBlocks映射中，其执行过程如下：

*private void rescan() throws InterruptedException {*

*scannedDirectives = 0;*

*scannedBlocks = 0;*

*try {*

*namesystem.writeLock();*

*.....*

*resetStatistics();*

*rescanCacheDirectives();*

*rescanCachedBlockMap();*

*blockManager.getDatanodeManager().resetLastCachingDirectiveSentTime();*

*} ......*

*}*

1. rescanCacheDirectives，依次遍历每个等待被cache的directive，将相关的block加入到cacheBlocks中：

*CachedBlock ncblock = new CachedBlock(block.getBlockId(),directive.getReplication(), mark);*

*CachedBlock ocblock = cachedBlocks.get(ncblock);*

*if (ocblock == null) {*

*cachedBlocks.put(ncblock);*

*ocblock = ncblock;}*

1. rescanCachedBlockMap，执行如下：

*for (Iterator<CachedBlock> cbIter = cachedBlocks.iterator();*

*cbIter.hasNext(); ) {*

*CachedBlock cblock = cbIter.next();*

*List<DatanodeDescriptor> pendingCached =*

*cblock.getDatanodes(Type.PENDING\_CACHED);*

*List<DatanodeDescriptor> cached =*

*cblock.getDatanodes(Type.CACHED);*

*List<DatanodeDescriptor> pendingUncached =*

*cblock.getDatanodes(Type.PENDING\_UNCACHED);*

*// Remove nodes from PENDING\_UNCACHED if they were actually uncached.*

*for (Iterator<DatanodeDescriptor> iter = pendingUncached.iterator();*

*iter.hasNext(); ) {*

*....*

*BlockInfo blockInfo = blockManager.*

*getStoredBlock(new Block(cblock.getBlockId()));*

*addNewPendingUncached(neededUncached, cblock, cached,*

*pendingUncached);*

*....*

*}*

*}*

*}*

addNewPendingUncached为每个等待被Cache的block选择合适的DataNode去cache，一般选择这三个block中剩余可用内存最多的DN，加入对应的DatanodeDescriptor的pendingCache列表。

1. DN定期向NN发送heartbeat RPC，用于同步block和cache状态，NN处理RPC时检查DN的pendingCache列表是否为空，不为空发送DatanodeProtocol.DNA\_CACHE命令给具体的DN去cache对应的block replica

*addCacheCommands(blockPoolId, nodeinfo, cmds);*

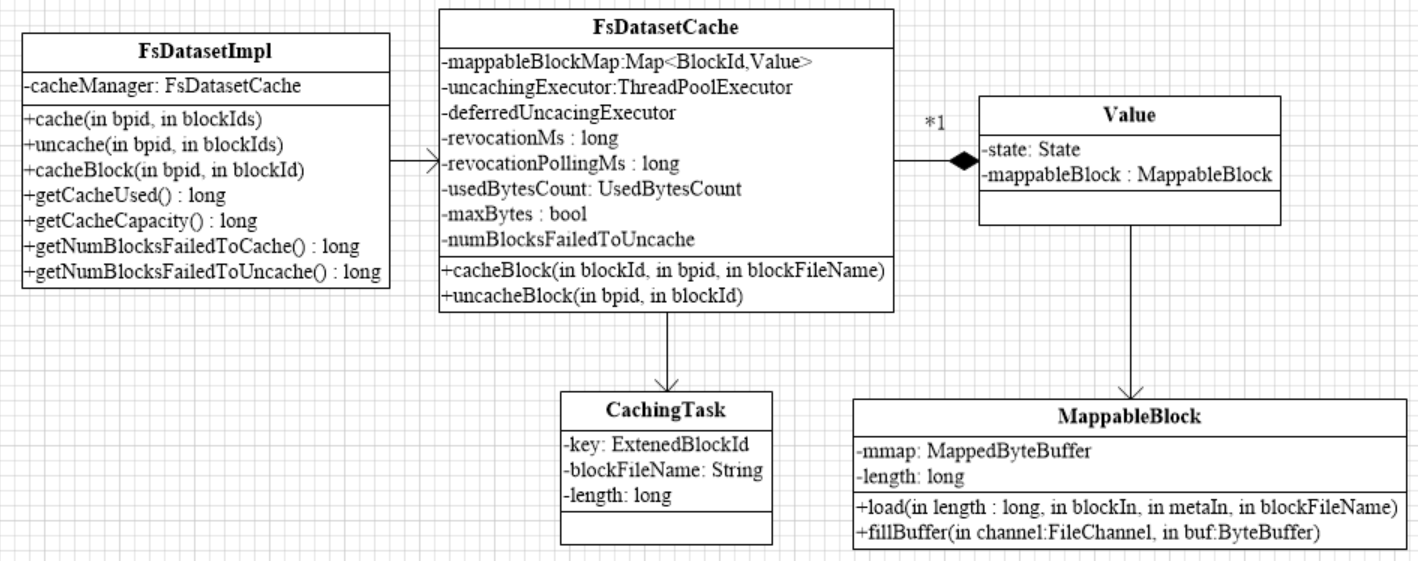
*{...*

*DatanodeCommand pendingCacheCommand = getCacheCommand(  
 nodeinfo.getPendingCached(), DatanodeProtocol.DNA\_CACHE,  
 blockPoolId);  
 if (pendingCacheCommand != null) {  
 cmds.add(pendingCacheCommand);  
 sendingCachingCommands = true;  
 }*

*}*

## **DataNode**

DataNode是执行缓存和置换的具体执行者，具体来说是cacheBlock和uncackBlock的调用，执行类为FsDatasetImpl#FsDatasetCache，相关类图如下：



FsDatasetCache的核心成员变量是mappableBlockMap，用于维护当前缓存的Block集合，其中Key为标记该Block的ExtendedBlockId，为了维护实现与Federation兼容，在blockid的基础上增加了blockpoolid，这样多个blackpool的block不会因为blockid相同产生冲突。Value是具体的缓存数据及缓存状态。核心方法为cacheBlock和uncacheBlock，

1. cacheBlock执行如下：

*synchronized void cacheBlock(long blockId, String bpid,*

*String blockFileName, long length, long genstamp,*

*Executor volumeExecutor) {*

*ExtendedBlockId key = new ExtendedBlockId(blockId, bpid);*

*Value prevValue = mappableBlockMap.get(key);*

*if (prevValue != null) {*

*numBlocksFailedToCache.incrementAndGet();*

*return;*

*}*

*mappableBlockMap.put(key, new Value(null, State.CACHING));*

*volumeExecutor.execute(*

*new CachingTask(key, blockFileName, length, genstamp));*

*}*

在CachingTask中，将block数据从磁盘加载到内存：

*blockIn = (FileInputStream)dataset.getBlockInputStream(extBlk, 0);*

*metaIn = DatanodeUtil.getMetaDataInputStream(extBlk, dataset);*

*mappableBlock = MappableBlock.*

*load(length, blockIn, metaIn, blockFileName);*

在load中调用NativeIO获取数据

*NativeIO.POSIX.getCacheManipulator().mlock(blockFileName, mmap, length);*

1. unCacheBlock

将Block从缓存中移除：

*synchronized (FsDatasetCache.this) {  
 mappableBlockMap.remove(key);  
}*

1. DN通过心跳接收到NN的Cache命令后，执行cacheBlock

*case DatanodeProtocol.DNA\_CACHE:  
 LOG.info("DatanodeCommand action: DNA\_CACHE for " +  
 blockIdCmd.getBlockPoolId() + " of [" +  
 blockIdArrayToString(blockIdCmd.getBlockIds()) + "]");  
 dn.getFSDataset().cache(blockIdCmd.getBlockPoolId(), blockIdCmd.getBlockIds());  
 break;*

## **DFSClient**

客户端本身的读数据逻辑基本上没有变化，与传统读模式的区别在于，当客户端向DN发出REQUEST\_SHORT\_CIRCUIT\_FDS读请求到达DN后，DN会首先判断数据是否被缓存，如果被缓存，则将缓存信息返回给客户端直接进行内存数据的ZeroCopy，如果数据没有缓存则采用传统方式进行ShortCircuit-Read，执行类为Receiver如下：

*protected final void processOp(Op op) throws IOException {*

*....*

*case REQUEST\_SHORT\_CIRCUIT\_FDS:*

*opRequestShortCircuitFds(in);*

*break;*

*...*

*}*

DataXceiver#

*boolean isCached = datanode.data.*

*isCached(blk.getBlockPoolId(), blk.getBlockId());*

*datanode.shortCircuitRegistry.registerSlot(*

*ExtendedBlockId.fromExtendedBlock(blk), slotId, isCached);*

https://blog.csdn.net/hellojoy/article/details/54730261

http://ju.outofmemory.cn/entry/71574

http://hexiaoqiao.github.io/blog/2016/09/04/hdfs-centralized-cache-management/

http://hadoop.apache.org/docs/r2.8.4/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html

https://blog.csdn.net/hellojoy/article/details/54730261