My design of cache

- Cache 设计：

cache 放于每一个 SDFSFileChannel 中，于特定的文件请求相绑定，并且实现了 LRU 的替换策略。

```
public class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private final int fileDataBlockCacheSize;
    private UUID fileuuid;
```

Cache 中有 cache 的 block 数量的属性 fileDataBlockCacheSize，还有对应文件请求的 UUID 的属性 fileuuid。实现的 LRUCache 是继承了 LinkedHashMap，特点为，以 LocatedBlock 的 hashcode 为 key 值，Block 类为 value 值（Block 实现后文介绍）；并且可以实现键值对超过规定数量后自动清除时间最远使用的键值对的功能，也就是可以方便的实现 LRU 替换策略，对应的 LRU 代码如下：

```
@Override
protected boolean removeEldestEntry(Map.Entry eldest) {
    if (size() > fileDataBlockCacheSize) {
        //判断是否dirty,然后写回disk
        Block block = (Block) eldest.getValue();
        if (block.isDirty()) {
            //write back
            block.writeBack(fileuuid);
        }
        return true;
    } else {
        return false;
    }

}
```

- Cache 实现中的数据块 Block 的实现：

Cache 中的键值对的 value 是 Block 对象，其实现为：

```java
public class Block {
    //应该封装Locatedblock?
    private final byte[] data = new byte[DataNode.BLOCK_SIZE];//da
    private boolean isDirty = false;
    private int limit = data.length;
    private int position = 0;
    //每次写回都只写回一个locatedBlock，所以只存一个位置的副本就好了
    private final InetAddress inetAddress;
    private final int blockNumber;
```

由于每次替换只需要写回到一个副本中，所以只存放对应 locatedBlock 的地址属性，并且有一个位置标记是否为脏 ;另外为了读写操作, 还会有 limit 和 position 属性, 与 blockbuffer 中的属性对应。

属性设计：

- data :初始化的时候就设置其大小为 DataNode 的 blockSize 的大小，但是 block 实际代表的数据大小由 limit 控制。
- isDirty :当改变这个 block 的内容的时候, 就会设置脏位, 当 channel 关闭或者 cache 替换这个块出去的时候，就会看是否为脏。
- limit :才是真正的 block 的数据大小, , 每次写的时候都会改变，读的时候会以 limit 为界限防止读过多的零, 这样就不需要每次写入更多的数据的时候再动态申请内存大小，影响效率。
- position：游标位置。读写操作的时候，position 指向正在读或写的位置，并以 position 来实现 append 和 overwrite，每次写的时候，都会在 position 的位置后面去覆盖数据，所以只要设定好游标的位置，就可以实现 append 操作。

Block 实现的主要方法有：

- **int write(ByteBuffer src)**：将 src 中的数据一位一位写入 block 的数据属性中，直到 src 末尾或者写满整个 block 为止，返回写入的 byte 数量，position 相应的会移动；
- **int read(ByteBuffer dst)**：将 block 的数据写入 dst 中，直到 dst 满了或者数据写完为止，返回写入的 byte 数量；
- **void writeBack(UUID fileuuid)**：将本 block 的内容写回 dataNode 的存储中。应用于替换策略时，或者 channel 关闭时写回所有脏的数据块。

Describe the problem I met during developing and how you solve it.

1. UUID 的使用在文件系统中所起的作用：

UUID 标识着一次文件请求，该请求时长从 open 开始一直到 close 结束。一开始本以为是和文件名称一样的作用，实现到最后才发现一次请求可以与文件名称无关，于是在 NameNode 中，实现了对于权限的检验，将 UUID 和 fileNode 一起放于 hashmap 里面;

final Map<UUID, FileNode> readonlyFile = new HashMap<>();

final Map<UUID, FileNode> readwritePFile = new HashMap<>();

2. 利用 ByteBuffer 对数据进行读写，以此为启发实现缓存数据块 Block 的读写实现：

本以为 ByteBuffer 是一个简单数组的包装类，然而发现其功能很多，并且不是简单的包装, 不是随便写写代码就可以用好的, 在不求甚解乱写了代码之后, 终于详细的考察其文档，发现 ByteBuffer 的 posiiton 非常方便的可以用来读写。所以用来实现文件系统的 append 和 overwrite 就可以用 position 来方便的实现，不需要每次读写都计算出应该从哪里开始，并且，每个数据块都是固定大小的，那么当一个文件的最后一个块没有填满的话，可以用 limit 这个属性来限制每次写入的时候的 position 移动的范围，这也是从 ByteBuffer 中给出的启示。

3. 缓存与实际存储上的差异：

每次打开文件，会将文件的元信息存储在 channel 中，那么问题来了，如果增加了一个 locatedblock 后，文件元信息就会改变，是在 channel 中改变了，还是马上写回 NameNode 中，我的实现是，同时去做。因为分配 locatedBlock 这件事情本来就需要和 NameNode 交互，那么为了保证 channel 里面缓存信息的有效性，不如同时都改了。但是文件大小不是立即在 nameNode 中立刻更改的，而是 channel 关闭时，与 NameNode 交互然后写回数据大小。

同样，dataNode 的数据就比较统一，在 Channel 里面改变了，在最后 close 时或者替换出去时才做最后的写回。

4. 利用测试用例进行需求的明确和系统的完善：

整个文件系统实现下来会非常的享受，原因在于从对系统结构的不明确到基本完全理解，只能在一遍遍手撕代码的过程中获得。但是，整个文件系统还是要明确很多细节上的实现，不能只停留在对于整个系统的把握上，所以与其检查代码的逻辑，不如直接写几个测试用例来的直截了当。所以在 SimpleDistributedFileSystem 的 main 函数的注释里面，写了一些测试用例，并打印了系统的状态，才测试中又发现自己对于 Block 的大小和游标的细节没有完善清楚，导致读出来的结果会有偏差。