# Smart File System说明文档

宣子涛 18302010015

> 项目github仓库地址：https://github.com/FLLLIGHT/SmartFileSystem
> 本项目开发/测试/运行环境：Linux (Ubuntu 20.04 LTS)

## 一、项目功能完成情况概述

1. 基础部分：全部功能/接口的实现。
2. Bonus部分：文件系统buffer的实现。

## 二、系统基本架构设计

计算机系统工程的重要要义之一便是分层(hierarchy)设计。具有分层架构设计的系统，每一层只与相邻的层次通过特定约定的接口进行联系，上层的功能基于下层提供的功能/接口进行实现，可以显著地控制系统中各个模块和层次之间的互连。此外，每一层/模块内部的功能具有层次相关性，从而实现整个系统的高内聚、低耦合。

本文件系统也采用分层架构设计，自底向上依次分为File I/O层(File Utils)、Block层、Block Manager层、File层、File Manager层和主程序层。接下来，我将首先介绍ID的设计与实现，而后自底向上依次介绍上述几层内部的设计与实现，最后说明采用固定大小的block size设计与采用不固定大小的block size设计各自的优势和缺点。（本系统实现采用固定大小的block size设计）

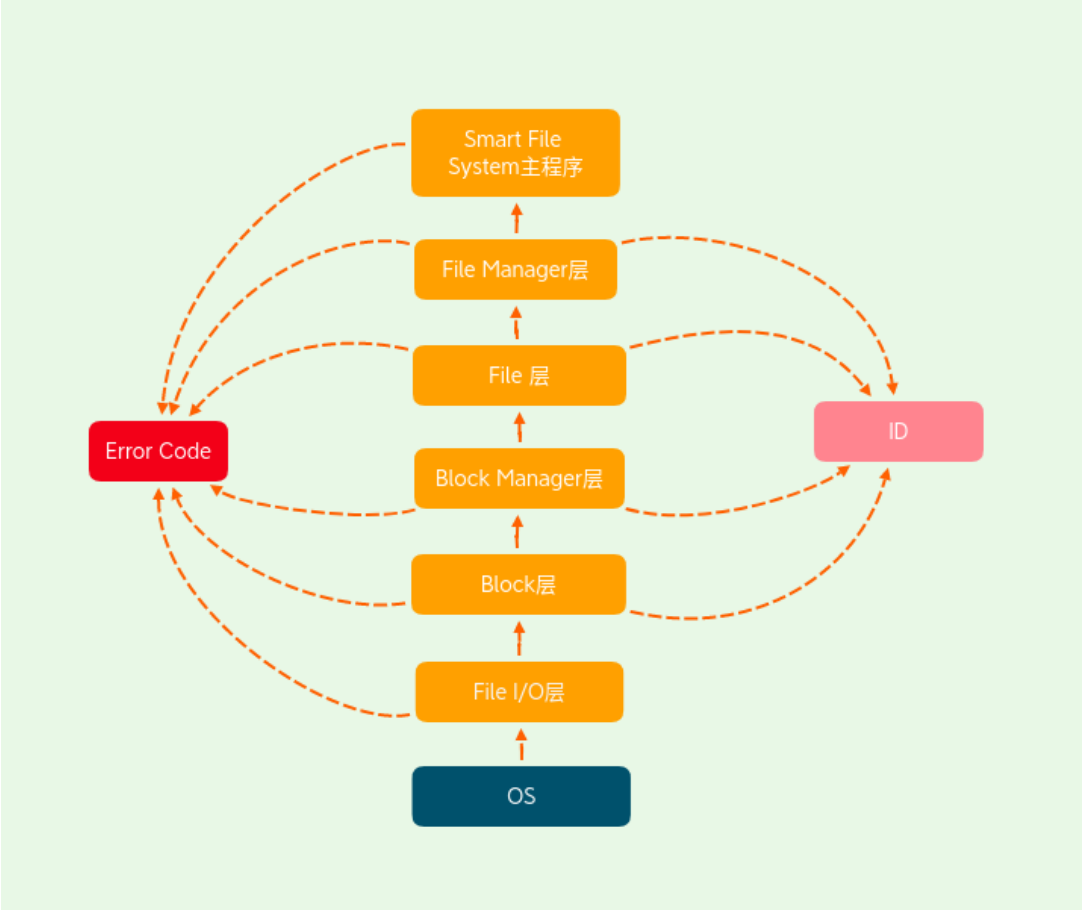


图2-1： 系统分层架构图

## 2.1 ID的设计与实现

IdImpl类实现了Id接口，系统中的Blcok、BlockManager、File和FileManager都以Id作为标识。Id内部存放了一个字符串为标识，重写了toString、hashCode和equals方法，通过这个内部存放的字符串作为比较的依据，从而能够在以Id为key的hashmap中直接通过get方法获取其对应的值（下文中会对其用途进行具体介绍）。此外，这种实现，不直接用一个字符串作为Blcok等具体类的标识，而是使用了Id，相当于在字符串标识和Blcok等具体类的标识中加了一层隔离，从而使系统具有可拓展性。例如，可以在Id中增加一个type属性，表明该Id属于哪个类，并在toString和hashCode中加入这个新的属性，作为判断两个Id是否相等的依据。

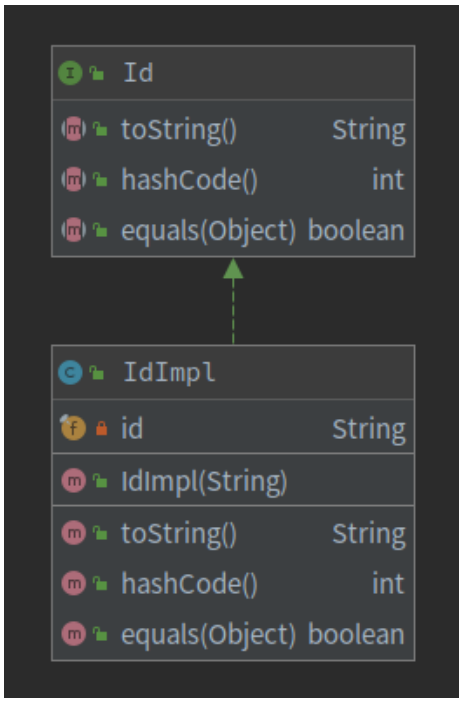


图2-2：ID类图

## 2.2 File I/O层

File I/O层提供了一系列操纵文件的方法。这些方法基于java api，对其进行了一定程度上的封装并进行了异常处理的包装，方便上层系统调用。

这一层作为Smart File System与操作系统文件系统之间唯一的桥梁，所有涉及文件读取和写入的操作都在这一层中进行，上层系统不会也不能越过该层进行I/O操作（从实现的角度来说，就是上层系统不需要java.io.*包）。作为一个Smart File System文件系统，如此分层还有一个额外的好处，即不会将Smart File System内部实现的File与java.io.File搞混，有利于代码的清晰性和可读性。

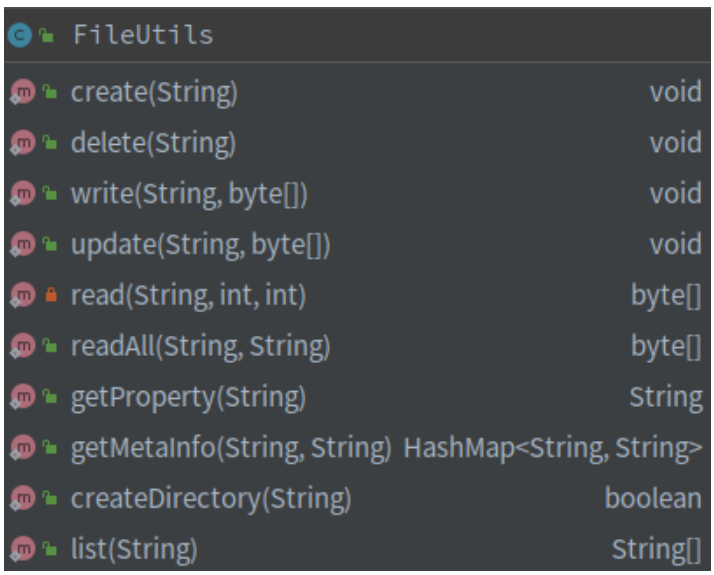


图2-3：File I/O层类图

## 2.3 Block层

Block层主要涉及底层block的meta信息和data信息的读取和写入，BlockImpl类实现了Block接口。

当上层系统第一次创建block时，调用BlockImpl(BlockManager, Id, byte[])构造函数，传入指定为该block的BlockManager, Id以及要存放至该block中的数据，Block层便将传入的数据进行解析、加工，而后写入指定位置（根据BlockManager和Id）的meta和data文件中。而后上层系统需要索引已经存在的block时，则调用BlockImpl(BlockManager, Id)构造函数进行索引即可，具体会在2.4 Block Manager层中进行进一步介绍。

在Block层中，向上层暴露了read和blockSize两个方法，上层系统可以通过这两个方法读取block的数据和大小。其中，read方法读取block的data文件，重新计算其checksum(用SHA-256算法)，并与meta文件中记录的checksum相比对，无误后将data文件中的全部数据传给调用者。blockSize方法即从meta文件中读取相应信息并返回即可。

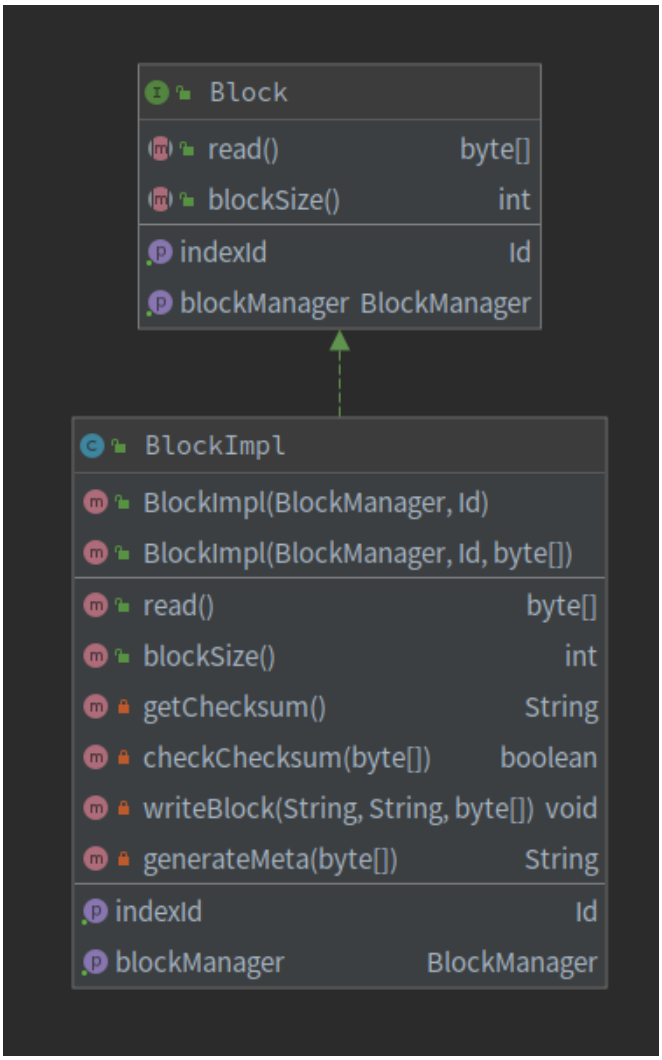


图2-4：Block层类图

## 2.4 Block Manager层

Block Manager层主要负责对Block的管理，能够创建、索引和读取Block的信息，BlockManagerImpl类实现了BlockManager接口。

当上层系统第一次创建block manager时，在构造函数中创建该block manager负责的文件夹，并初始化block manager的id数据（id数据存放了该block manager的下一个可用id）并将之持久化。如果当前block manager已经存在，则对其中的所有block建立索引。每个block manager中维护着一个以block的id为键，block为值的hashmap，对已经存在的block manager建立索引时，则读取已有的block及其id，并将其put入hashmap中。而后每创建一个新的block后，便将该block及其id放入hashmap中。block manager通过这个hashmap对其中的block进行索引与读取。

在Block Manager层中，向上层暴露了getBlock，newBlock和newEmptyBlock三个方法。对于getBlock，只需根据传入的Id参数，从维护的hashmap中get出对应的Block并返回即可；对于newBlock，从id.data文件中获取下一个可用的id，而后调用BlockImpl(BlockManager, Id, byte[])构造函数，新建block，最后将该block及其id放入hashmap中；newEmptyBlock使用newBlock方法，new一个大小为传入的blockSize参数的byte数组，以其作为传入参数调用newBlock即可。

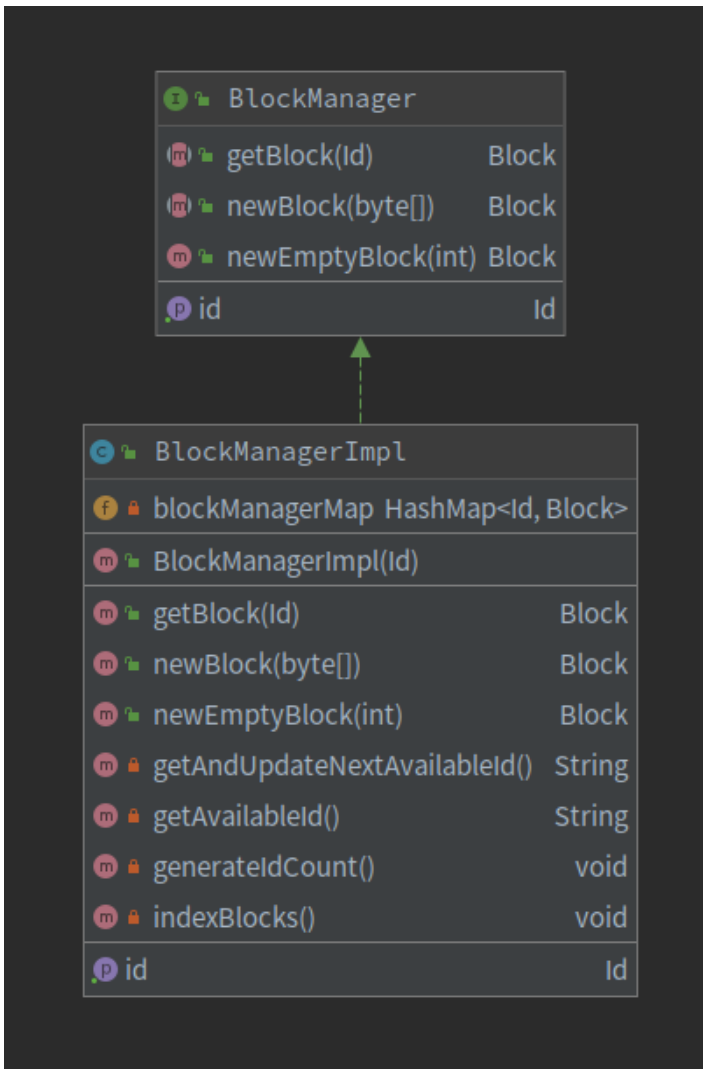


图2-5：Block Manager层类图

## 2.5 File层

File层主要负责维护file的meta数据和指针、对上层系统要求写入的数据根据系统配置的block size进行划分、将划分后的数据随机分配给block manager、根据上层系统要求读入的数据和file的meta数据要求对应的block manager对block进行读取等。FileImpl类实现了File接口。

当上层系统第一次创建file时，调用FileImpl(FileManager, Id, boolean)构造函数，传入指定为该file的FileManager和Id，File层便初始化file的meta信息并将其持久化写入磁盘文件。而后上层系统需要索引已经存在的file时，则调用FileImpl(FileManager, Id)构造函数进行索引即可，具体会在2.6 File Manager层中进行进一步介绍。如果需要创建一个新文件并将其他文件中的信息copy到这个文件中，则调用FileImpl(FileManager, FileManager, Id, String)构造函数，先从来源的file manager中读取要拷贝的file的meta信息，并将此meta信息作为初始化数据，写入新文件的meta信息中去。由于block是不可改变的，因此此后这两个file若有任何变动，则修改对应file的meta信息即可，不会相互影响。

值得注意的是，一个file在创建、初始化meta信息时，会读取此时配置文件的block size信息，并将其写入file的meta信息中，此后，该file的所有block的size均为meta信息中存放的block size大小（除了最后一个block的size可能小于此大小），不可改变。我会在2.7节中说明采用固定大小的block size设计与采用不固定大小的block size设计各自的优势和缺点。

在File层中，我将write和read都各自分为两小层，其中上层暴露给外部，下层作为内部模块，具有通用性和便捷性，可以在内部调用中进行复用。

具体而言，write的上层暴露给外部，接受参数为一个byte数组，表明从当前文件内部维护的指针(pos)开始写byte数据；write的下层接受两个参数，分别为一个byte数组和一个int型的start index，表明从文件的第start index个block开始，将后续的block改为byte数组中数据所新生成的block（新block会在file的meta数据中覆盖原block，若新生成的block数少于原有block数，则删去多余的原block），即调整从第start index个block开始的所有block。上层的write对传入的数组和file中原有内容进行整合，根据光标的位置，读取当前光标位置开始的所有block数据，而后将传入的byte数组插入原数据中的制定位置，最后调用下层的write，使用整合后的所有数据替换原file数据中当前光标位置开始的所有block数据。此外，setSize中，若要set的size大小小于原大小，则也可直接调用下层的write，清除多余的数据；close中，若要将buffer写回磁盘，则也可直接调用下层的write，使用buffer数据替换原有的全部数据，不必考虑buffer中新数据和原数据的差异，具有便捷性，有关buffer的具体内容会在第三部分中进行讨论。

read也分为上下两层。上层的read暴露给外部，接受参数为一个int类型的长度(length)，表明从当前指针位置开始读length个byte的数据。下层的read分为两个模块。第一个模块的read接受参数为一个int类型的length，一个long类型的start index，一个boolean类型的指明是否要从buffer中读的标识参数。该模块的read表明从start index的位置开始读length个字节的数据，如果标志位为true，则直接从buffer中拷贝对应位置和数量的数据并返回即可，如果标志位为false，则计算数据在第一个和最后一个要读的block中的位置，从meta文件中获取对应的block manager和block id而后进行读取即可。如果一个logic block中的部分block损坏，则会抛出异常并继续读取未损坏的block，如果一个logic block中的全部block都损坏了，则会抛出异常（error code：12）并终止读取，具体内容可见第四部分异常处理。第二个模块的read为readAll，表明读取该文件的全部内容，主要用于内部初始化buffer，当read(int)判断配置参数中buffer为true，但当前buffer为null时，则调用readAll，readAll则会指定start index为0，length为file size，buffer标志位为false，调用下层的第一模块的read，从磁盘中读取文件的全部内容并返回，而后返回为read(int)，并设置buffer为返回的文件全部数据。下一次上层read再次读取数据时，则可直接从buffer中读取对应的数据。此外，在执行set sizeu或write时，如果遇到需要buffer但未初始化的情况，也调用readAll对buffer完成初始化，有关buffer的具体内容会在第三部分中进行讨论。

此外，File层还向外部提供了move当前指针，查找当前指针位置，返回文件大小，设置文件大小等方法。这些方法大多基于上述的read和write的实现，对其底层模块进行了复用，再次不再做过多介绍，具体可见代码内注释。

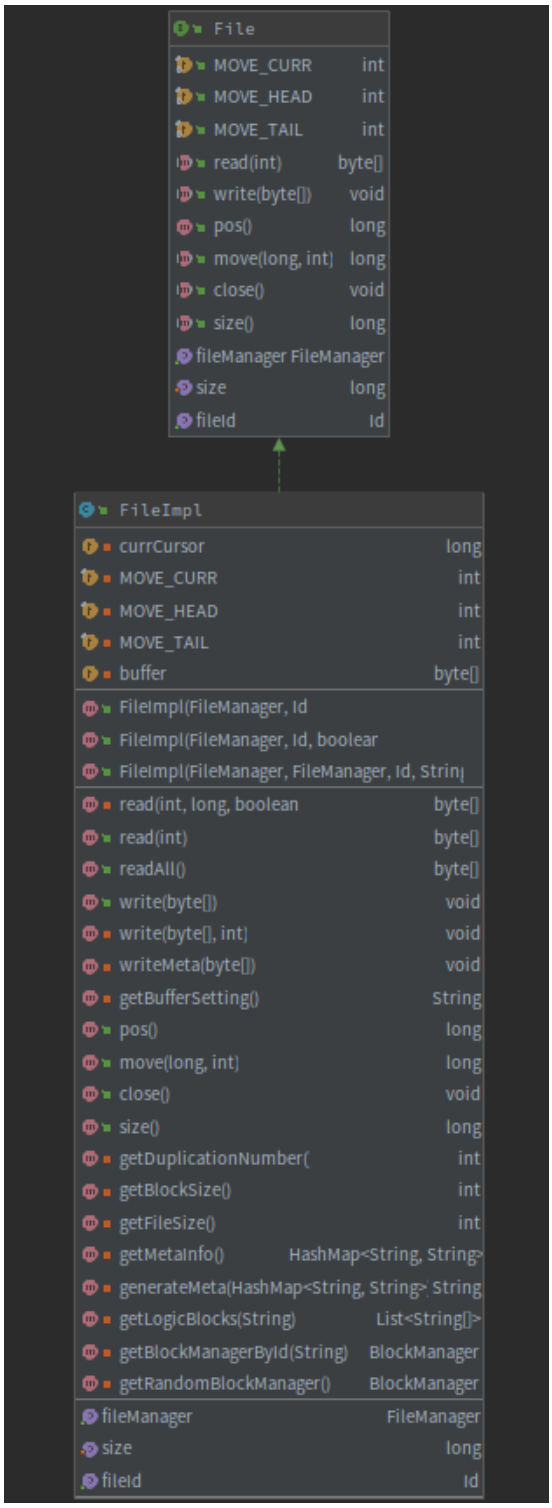


图2-6：File层类图

## 2.6 File Manager层

File Manager层主要负责对File的管理，能够创建、索引、拷贝和读取File的信息，FileManagerImpl类实现了FileManager接口。

当上层系统第一次创建file manager时，在构造函数中创建该file manager负责的文件夹。如果当前file manager已经存在，则对其中的所有file建立索引。每个file manager中维护着一个以file的id为键，file为值的hashmap，对已经存在的file manager建立索引时，则读取已有的file及其id，并将其put入hashmap中。而后每创建一个新的file后，便将该file及其id放入hashmap中。file manager通过这个hashmap对其中的file进行索引与读取。

在File Manager层中，向上层暴露了getFile，newFile和copyFile三个方法。对于getFile，只需根据传入的Id参数，从维护的hashmap中get出对应的File并返回即可；对于newFile，调用2.5节中说明的对应的FileImpl的构造函数，创建新的file，而后将其及其放入hashmap中即

可；对于copyFile，也是调用2.5节中说明的对应的FileImpl的构造函数，创建新的file，而后将其及其放入hashmap中即可。

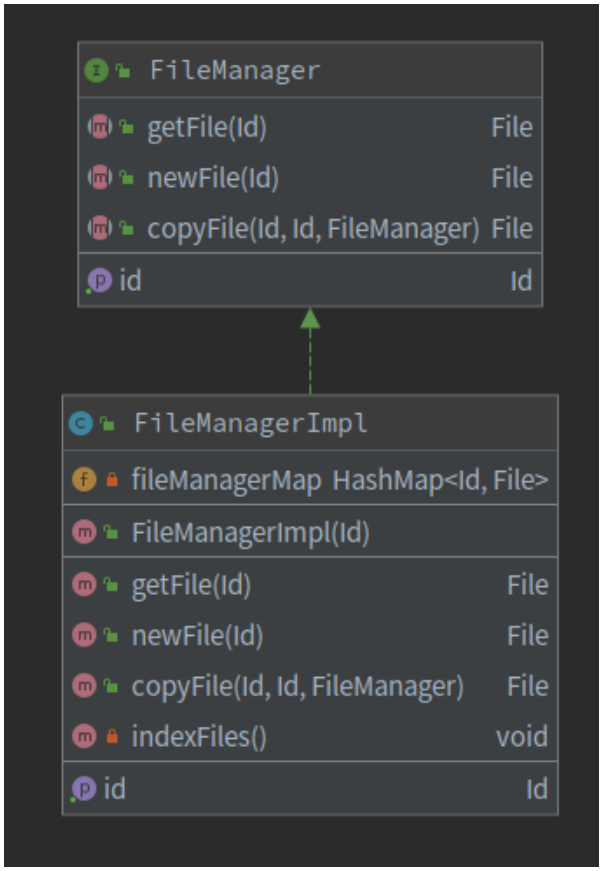


图2-7：File Manager层类图

## 2.7 固定Block Size VS 不固定Block Size

固定block size的好处：

- 可以直接通过数值计算（位置除以block size大小）直接得到要读取/写入的block的位置。

固定block size的缺点：

- 每次修改文件时，需要修改对应位置及其之后的所有block，对于大文件性能较差，也会造成block冗余。（优化/垃圾回收算法可解决block冗余的问题）

不固定block size的好处：

- 每次修改文件时，最多只需要修改对应位置及其相邻位置的block，性能好。

不固定block size的缺点：

- 在读取/写入文件时，要从头遍历block的meta信息，得到从头开始的每个block的实际大小，直至目标block，才能计算出要读取/写入的block的位置，对于大文件性能较差。
- 由于每个block是不可改变的，因此在同一位置大量写入1字节数据，会造成block冗余，尤其是加上duplication。（优化/垃圾回收算法可解决此问题）

## 2.8 Smart File System的本质

该系统建立在操作系统的文件系统的基础之上，还是利用了操作系统的文件系统的目录结构，因此其本质只是一个索引，可以忽略上述Block、File、BlockManager、FileManager等设计，直接在指定目录的指定文件下进行写入和读取，也可以取得相同的效果。Block、File、BlockManager、FileManager的最大用途只是提供其Id作为索引。当然，上述分层的设计能够

让系统的层次结构清晰（比如下面要说明的buffer在file层实现，如果用面向过程的写法，代码结构不会这么清晰）。

## 三、Buffer的设计与实现

本系统的buffer以file为最小单位，在File层中对其进行实现，每一个file会维护一个byte数组，作为该文件的buffer。用户可以在file.properties文件中配置是否要使用buffer(yes/no)。

在对文件进行read操作时，如果配置文件中的buffer参数为yes，则进一步判断当前file中的buffer数组是否为null。若为null，说明系统还从未从磁盘中将数据读入buffer，则调用readAll将该文件中的所有数据读入buffer数组中（有关readAll，详见2.5中的介绍）；若buffer数组不为null，则直接从buffer数组中拷贝要读取的内容并返回即可。

在对文件进行write操作时，如果配置文件中的buffer参数为yes，则进一步判断当前file中的buffer数组是否为null。若为null，同样说明系统还从未从磁盘中将数据读入buffer，则调用readAll将该文件中的所有数据读入buffer数组中；若buffer数组不为null，则直接将要写入的数据插入buffer数组中的对应位置即可（通过System.arrayCopy）。

在对文件进行set size操作时，同样首先进行如上的判断，而后直接更新file维护的buffer数组的内容即可。

## 四、异常处理说明

### 4.1 IO_EXCEPTION (error code number: 1)

普通的文件I/O异常，包括了除下述系统内部指定的异常范围外的所有文件I/O异常。

### 4.2 CHECKSUM_CHECK_FAILED (error code number: 2)

为了防止文件的数据在用户不知情的情况下被篡改，在生成每个block时，会同时使用散列函数/hash函数(SHA-256算法)获取该block内容的信息摘要(checksum)，并将该摘要存放在block的meta文件中。

当再次读取该block时，如果发现对block当前存储的内容进行hash后得到的结果与存放在meta文件中的checksum不同，则说明该block的内容已经被修改，与存入时的内容不一致（当然，也可能是meta文件中的checksum被修改），则此block的内容不再可靠，不能够继续被读取，于是便会报CHECKSUM_CHECK_FAILED。

### 4.3 FILE_ALREADY_EXISTED (error code number: 3)

当使用FileManager的newFile方法，在该FileManager下创建新文件时，若试图创建的文件id已经在该FileManager中存在，则会报FILE_ALREADY_EXISTED。

### 4.4 BLOCK_ALREADY_EXISTED (error code number: 4)

当系统内部调用BlockManager的newBlock方法，在该BlockManager下创建新block时，若试图创建的block id已经在该BlockManager中存在，则会报FILE_ALREADY_EXISTED。

每个BlockManager下会持久化地维护一个id.data文件，该文件保存着该BlockManager的下一个可用id（不与该BlockManager下的已有id重复）。如果报FILE_ALREADY_EXISTED，通常是id.data文件的内容损坏或是被篡改。

### 4.5 FILE_META_NOT_EXISTED (error code number: 5)

当系统内部调用试图读取某file的meta数据，但该file的meta文件却不存在时，会抛出FILE_META_NOT_EXISTED。

该异常属于试图寻找本该存在的数据/文件，但是却找不到该数据/文件。可能出现的情况为：在系统完成初始化索引后，file的meta文件被人为删除，而后系统再试图读取该文件。

### 4.6 BLOCK_META_NOT_EXISTED (error code number: 6)

当系统内部调用试图读取某block的meta数据，但该block的meta文件却不存在时，会抛出BLOCK_META_NOT_EXISTED。

该异常属于试图寻找本该存在的数据/文件，但是却找不到该数据/文件。可能出现的情况为：在系统完成初始化索引后，block的meta文件被人为删除，而后系统再试图读取该文件。

### 4.7 BLOCK_DATA_NOT_EXISTED (error code number: 7)

当系统内部调用试图读取某block的data数据，但该block的data文件却不存在时，会抛出BLOCK_DATA_NOT_EXISTED。

该异常属于试图寻找本该存在的数据/文件，但是却找不到该数据/文件。可能出现的情况为：在系统完成初始化索引后，block的data文件被人为删除，而后系统再试图读取该文件。

### 4.8 ID_DATA_NOT_EXISTED (error code number: 8)

当系统内部调用试图读取id的data数据，但该文件却不存在时，会抛出ID_DATA_NOT_EXISTED。

该异常属于试图寻找本该存在的数据/文件，但是却找不到该数据/文件。可能出现的情况为：BlockManager的id.data文件被删除。

### 4.9 FILE_NOT_EXISTED (error code number: 9)

当使用FileManager的getFile方法，在该FileManager下获取某文件时，若试图获取的文件id在该FileManager中并不存在，则会报FILE_NOT_EXISTED。

该异常属于试图寻找本就不存在的数据/文件。

### 4.10 BLOCK_NOT_EXISTED (error code number: 10)

当系统内部调用试图读取某block的数据（meta/data），但该block的meta/data文件却不存在（不在BlockManager的blockManagerMap中）时，会抛出BLOCK_NOT_EXISTED。

该异常属于试图寻找本就不存在的数据/文件。可能出现的情况为：file的meta文件中的某logic block被篡改为实际上不存在block，而后读取该文件时，对应的BlockManager会发现索引中不存在要找的block，便会抛出BLOCK_NOT_EXISTED。

### 4.11 SETTING_FILE_ERROR (error code number: 11)

当系统内部调用试图读取配置文件，但发现配置文件不存在或配置文件中的对应项不符合要求，则会抛出SETTING_FILE_ERROR。

可能出现的情况为：file.properties文件被删除；buffer项的value值被设为除yes或no的其他值；buffer项/blockSize项/duplicationNumber项被删除。

### 4.12 FILE_DATA_DAMAGED (error code number: 12)

在读取file的数据内容时，若该file的某个logic block的所有副本block均损坏（CHECKSUM_CHECK_FAILED/BLOCK_META_NOT_EXISTED/BLOCK_DATA_NOT_EXISTED/BLOCK_NOT_EXISTED）时，系统无法正常获取该logic block的内容，读取失败，抛出FILE_DATA_DAMAGED异常。

### 4.13 MOVE_OUT_OF_BOUNDARY (error code number: 13)

在使用file的move方法时，若试图将指针移至文件的有效范围(0~file.size())之外，则会抛出MOVE_OUT_OF_BOUNDARY异常。

### 4.14 READ_OUT_OF_BOUNDARY (error code number: 14)

在使用file的move方法或其他相关方法时，若试图读取文件的有效范围(0~file.size())之外的数据，则会抛出READ_OUT_OF_BOUNDARY异常。