Software Engineering Institute

Hadoop 文件系统分析报告

范永刚、何芳

2011 年 1 月 12 日

技术白皮书
XD/SEI/REPACE-2011-TR-001

Hadoop 源代码分析

http://www.xidian.edu.cn

西安电子科技大学

目录

1 修谄谒德

序号	时间	修订人	版本
1	2011 年 1 月 12 日	何芳、范永刚	1.0
2	2011 年 1 月 28 日	鲍亮	1.1

2 擰解

Hadoop 分布式文件系统被设计成适合运行在通用硬件(commodity hardware)上的分布式文件系统。

批注 [BL1]: 这个摘要需要重新总结一下

3 org. apache. hadoop. fs 卧悝道

org.apache.hadoop.fs 包提供了一个抽象文件系统的 API。该包下有 50 多个类，有 7 个包。如图 2-1 所示

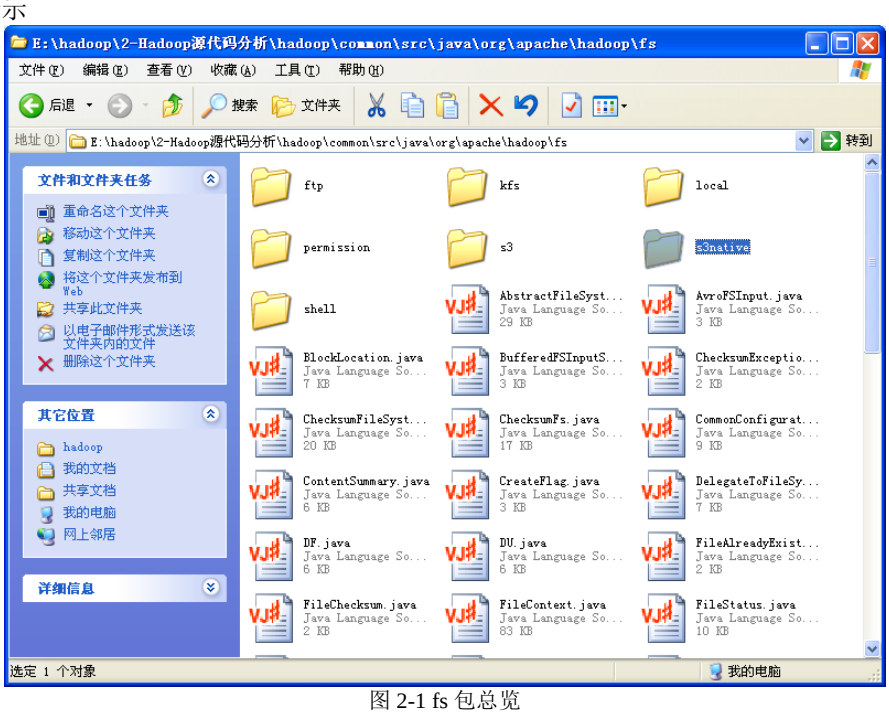

图 2-1 fs 包总览

org.apache.hadoop.fs 包中的 ***FileSystem*** 抽象类和 ***AbstractFileSystem*** 抽象类作为抽象文件系统的基类，提供了基本的抽象操作。其中 FileSystem 类是 0.21 版本之前唯一的基类，但在 0.21 版本中，出现了 ***AbstractFileSystem***，该类似乎来取代 ***FileSystem*** 类原来的部分功能。在这两个基类的基础上形成了两个类继承的层次结构。

org.apache.hadoop.fs 子包 ftp、kfs、local、s3 和 s3native 都是实现的具体的文件系统。Permission 文件夹实现了有关文件访问许可的功能。Shell 文件夹实现了对 shell 命令的调用。

4 Hadoop 早佘綺鋼槤道

4.1 糝峁殃缵桕

Hadoop 文件系统可以访问多个不同的具体的文件系统，如 ***HDFS***、***KFS*** 和 ***S3*** 文件系统。不同的文件系统很难进行通信，Hadoop 抽象文件系统类似 Linux 中的 VFS 虚拟文件系统，它从不同的文件系统中抽取了共同的操作，这些操作是一般的文件系统都具有的操作，如打开文件，创建文件，删除文件，复制文件，获取文件的信息等。这些共同的基本操作组合在一起就形成了 FileSystem 抽象类和 AbstractFileSystem 抽象类。然后从基类派生，以实现各不同的具体的文件系统。如前所述，关于文件系统，有两个类继承的层次结构，如图 3-1 所示：

批注 [BL2]: 这句话是什么意思？是不是说通过该文件系统的接口，这些不同的文件系统之间可以互相访问？
HDFS 和文件系统之间的依赖关系到底是什么？是谁依赖谁？

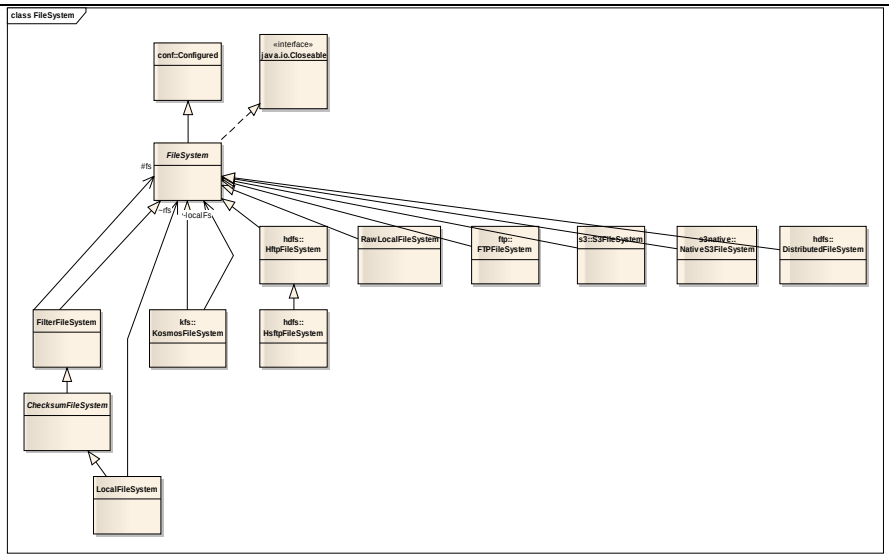

图 3-1. FileSystem 抽象类

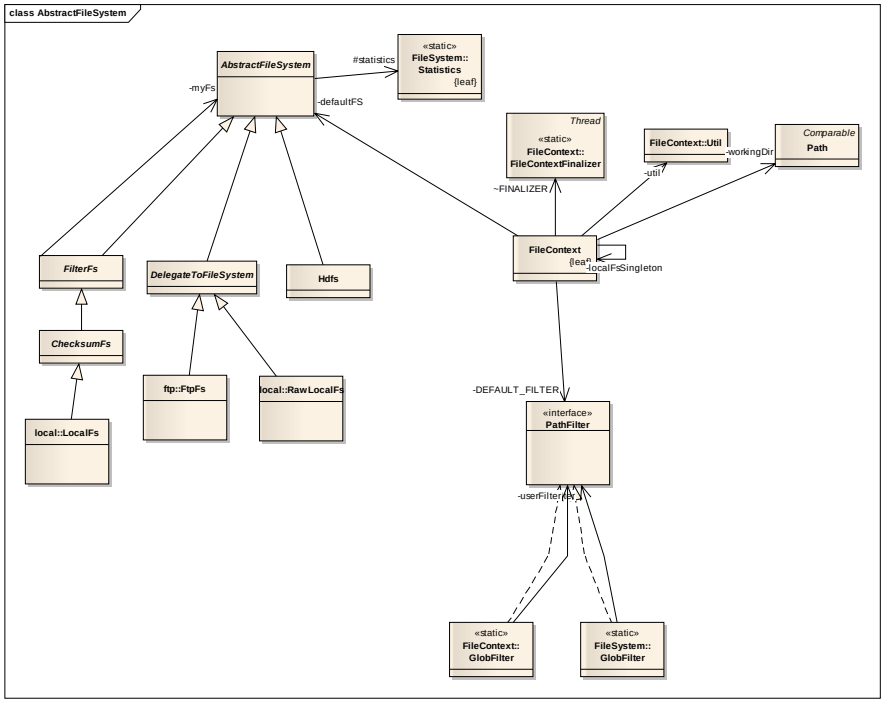

图 3-2. AbstractFileSystem 抽象类

这两类层次结构十分相似，很多类的实现几乎完全一样。

除 FilterFileSystem 外，FileSystem 的直接子类都是具体的文件系统。包括以下文件系统：

- KosmosFileSystem：cloudstore(其前身是 Kosmos 文件系统)是相似于 HDFS 或是 Google 的 GFS 的文件系统，用 C++编写。
- HftpFileSystem：一个在 HTTP 上提供对 HDFS 只读访问的文件系统(虽然其名称为 HFTP，但它与 FTP 无关)。通常与 distcp 结合使用，在运行不同版本 HDFS 的集群间复制数据。

- RawLocalFileSystem：代表本地文件系统。
- FTPFileSystem：由 FTP 服务器支持的文件系统。
- S3FileSystem：由 Amazon S3 支持的文件系统，以块格式存储文件(与 HDFS 很相似)来解决 S3 的 5 GB 文件大小限制。
- NativeS3FileSystem：由 Amazon S3 支持的文件系统。

批注 [BL3]: 这里“本地”的意思是不是说 Hadoop 中 fs 所运行操作系统的文件系统？

批注 [BL4]: 原生的？有 5G 限制的系统？

Hadoop 的使用者可以分为两类，应用程序编写者和文件系统实现者。在 Hadoop 0.21 版本之前，FileSystem 类作为一般（抽象）文件系统的基类，一方面为应用程序编写者提供了使用 Hadoop 文件系统的接口，另一方面，为文件系统实现者提供了实现一个文件系统的接口（如 hdfs，本地文件系统，kfs 等等）。但在 Hadoop 0.21 版本中，出现了 FileContext 类和 AbstractFileSystem 类，通过这两个 API，可以将原来集中于 FileSystem 一个类中的功能分开，让使用者更加方便的在应用程序中使用多个文件系统。FileContext 这个 API 还没有在 hadoop 中被大量的使用，因为还没有被合并到 mapreduce 计算中，但是它包含了正常的 FileSystem 接口没有的新功能，如支持 hdfs 层面的软链接等。FileContext 类是用来取代 FileSystem 类，向***应用程序编写者***提供使用 Hadoop 文件系统的接口，而原来的 FileSystem 则仅由***文件系统实现者***使用。估计 AbstractFileSystem 类将来会取代 FileSystem 类。

从图 3-2 中可以看出 AbstractFileSystem 对应 FileSystem，FilterFs 对应 FiterFileSystem，ChecksumFs 对应 ChecksumFileSystem，LocalFs 对应 LocalFileSystem。

而在 FileSystem 图中的各个具体的文件系统类，在 AbstractFileSystem 图中，除了 FtpFs 和 RawLocalFs 外，则通过 DelegateToFileSystem（代理模式）对各个具体的文件系统进行了封装，具体的实现还是通过各个具体的文件系统来实现的。

4.2 迥凇迥剜涣

Hadoop 中类的设计在很多地方模仿了 Java。典型的就是文件的输入输出流。

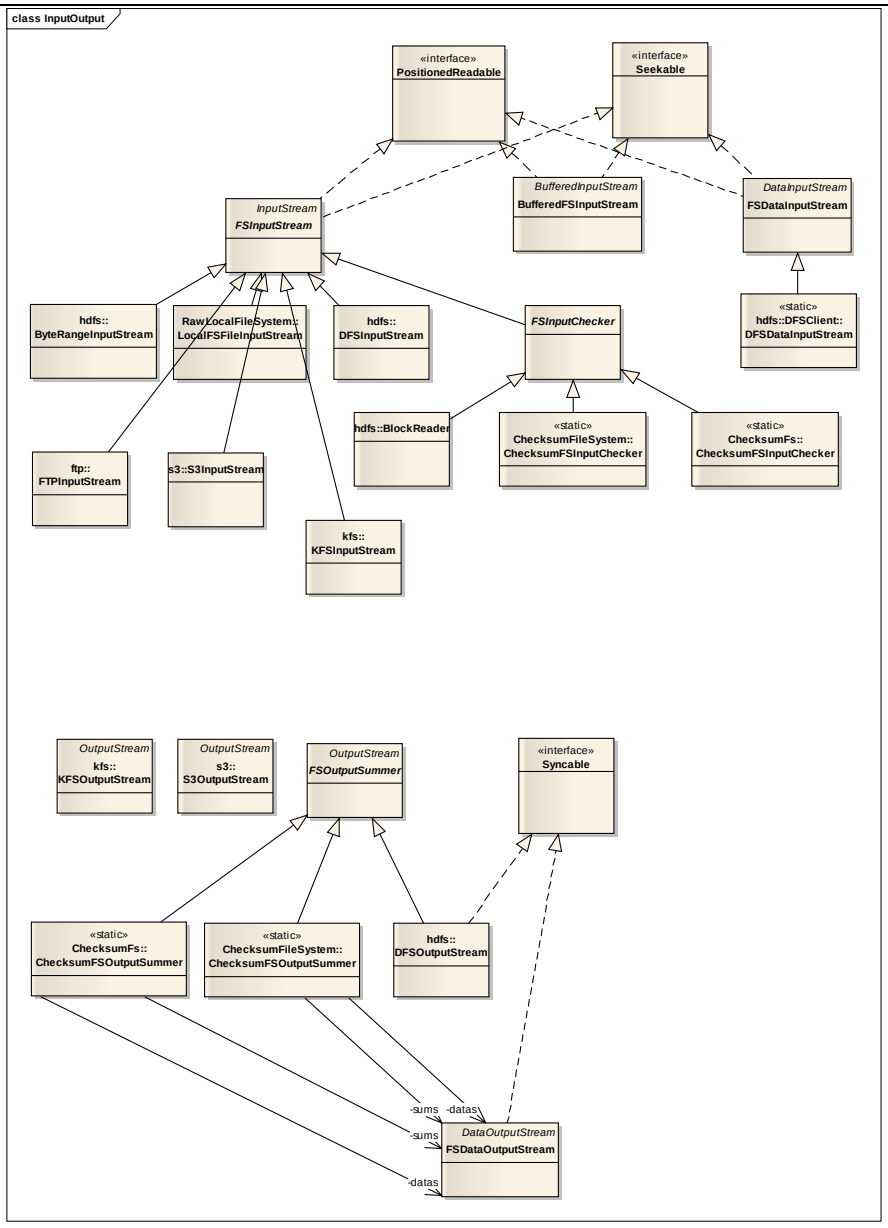

图 3-3. 输入输出流

4.2.1 Java 中的 IO

可将 Java 库的 IO 类分割为输入与输出两个部分，这一点在用 Web 浏览器阅读联机 Java 类文档时便可知道。通过继承，从 InputStream（输入流）衍生的所有类都拥有名为 read()的基本方法，用于读取单个字节或者字节数组。类似地，从 OutputStream 衍生的所有类都拥有基本方法 write()，用于写入单个字节或者字节数组。然而，我们通常不会用到这些方法；它们之所以存在，是因为更复杂的类可以利用它们，以便提供一个更有用的接口。因此，我们很少用单个类创建自己的系统对象。一般情况下，我们都是将多个对象重叠在一起，提供自己期望的功能。我们之所

以感到 Java 的流库（Stream Library）异常复杂，正是由于为了创建单独一个结果流，却需要创建多个对象的缘故。很有必要按照功能对类进行分类。库的设计者首先决定与输入有关的所有类都从 InputStream 继承，而与输出有关的所有类都从 OutputStream 继承。

4.2.1.1 InputStream

InputStream 的作用是标志那些从不同起源地产生输入的类。这些起源地包括（每个都有一个相关的 InputStream 子类）：

(1) 字节数组

(2) String 对象

(3) 文件

(4) “管道”，它的工作原理与现实生活中的管道类似：将一些东西置入一端，它们在另一端出来

(5) 一系列其他流，以便我们将其统一收集到单独一个流内

(6) 其他起源地，如 Internet 连接等

除此以外，FilterInputStream 也属于 InputStream 的一种类型，用它可为“破坏器”类提供一个基础类，以便将属性或者有用的接口同输入流连接到一起。

批注 [BL5]: 什么意思？

ByteArrayInputStream：允许内存中的一个缓冲区作为 InputStream，使用从中提取字节的缓冲区作为一个数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口。

StringBufferInputStream：将一个 String 转换成 InputStream 一个 String（字串）。基础的实施方案实际采用一个 StringBuffer(字串缓冲)作为一个数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口。

FileInputStream：用于从文件读取信息代表文件名的一个 String，或者一个 File FileDescriptor 对象作为一个数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口。

PipedInputString：产生为相关的 PipedOutputStream 写的数据。实现了“管道化”的概念。

PipedOutputStream：作为一个数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口

批注 [BL6]: 什么意思？

SequenceInputStream：将两个或更多的 InputStream 对象转换成单个 InputStream 使用两个 InputStream 对象或者一个 Enumeration，用于 InputStream 对象的一个容器作为一个数据源使用。通过将其同一个 FilterInputStream 对象连接，可提供一个有用的接口 FilterInputStream 对作为破坏器接口使用的类进行抽象；那个破坏器为其他 InputStream 类提供了有用的功能。

批注 [BL7]: 不明白

4.2.1.2 OutputStream

这一类别包括的类决定了我们的输入往何处去：一个字节数组（但没有 String；假定我们可用字节数组创建一个）；一个文件；或者一个“管道”。

除此以外，FilterOutputStream 为“破坏器”类提供了一个基础类，它将属性或者有用的接口同输出流连接起来

批注 [BL8]: 解释

ByteArrayOutputStream：在内存中创建一个缓冲区。我们发送给流的所有数据都会置入这个缓冲区。可选缓冲区的初始大小用于指出数据的目的地。若将其同 FilterOutputStream 对象连接到一起，可提供一个有用的接口。

FileOutputStream：将信息发给一个文件，用一个 String 代表文件名，或选用一个 File 或 FileDescriptor 对象用于指出数据的目的地。若将其同 FilterOutputStream 对象连接到一起，可提供一个有用的接口。

PipedOutputStream：我们写给它的任何信息都会自动成为相关的 PipedInputStream 的输出。实现了“管道化”的概念。PipedInputStream 为多线程处理指出自己数据的目的地，将其同 FilterOutputStream 对象连接到一起，便可提供一个有用的接口

FilterOutputStream 对作为破坏器接口使用的类进行抽象处理；那个破坏器为其 OutputStream 类提供了有用的功能

4.2.1.3 DataInputStream

DataInputStream 从 FilterInputStream 派生，在 FilterInputStream 中 InputStream in 的属性。

批注 [BL9]: 什么意思？

数据输入流允许应用程序以与机器无关方式从底层输入流中读取基本 Java 数据类型。应用程序可以使用数据输出流写入稍后由数据输入流读取的数据。DataInputStream 对于多线程访问不一定是安全的。线程安全是可选的，它由此类方法的使用者负责。

4.2.1.4 DataOutputStream

DataOutputStream 数据输出流允许应用程序以适当方式将基本 Java 数据类型写入输出流中。然后，应用程序可以使用数据输入流将数据读入。

4.2.2 Hadoop 的输入输出流

在 Hadoop 中，FSInputStream、FSDataInputStream 和 FSDataOutputStream 的作用与 InputStream、DataInputStream 和 DataOutputStream 在 Java IO 中的作用类似。用 FileSystem 的 create 方法创建一个输出流时的返回值类型为 FSDataOutputStream，而 open 方法则返回一个 FSDataInputStream 实例。Hadoop 中并没有 FSOutputStream，当有两个从 OutputStream 派生的类：KFSOutputStream 和 S3OutputStream。这两个类也就相当于 OutputStream。很多文件系统都会从 FSInputStream 派生，实现自己特定的输入输出流，如 S3InputStream 等等。

5 FileSystem 溢凇劏秘

5.1 fs 亐奩捺吆

fs 包中的接口不多，有：

FsConstants 表示文件系统有关的常量。

PathFilter 对文件路径进行筛选。

PositionedReadable 和 Seekable 提供了随机读写功能

Syncable 文件同步

5.2 FileSystem

FileSystem 的类图见下图。由此可见 FileSystem 是一个很大的抽象类。在 fs 包中，最重要的可以说是 FileSystem 抽象类。它定义了文件系统中涉及的一些基本操作，如：create，rename，delete... 另外包括 一些分布式文件系统具有的操作：copyFromLocalFile, copyToLocalFile,...类似于 Ftp 中 put 和 get 操作。 LocalFileSystem 和 DistributedFileSystem，继承于此类，分别实现了本地文件系统和分布式文件系统。

批注 [BL10]: 图在哪里？

5.2.1 Configured 基类和 Closeable 接口

FileSystem 抽象类从 Configured 基类派生，并实现了 Closeable 接口。Configured 基类的源代码如下，该基类仅是简单的提供了访问配置文件的方法。

```
/** Base class for things that may be configured with a {@link Configuration}. */
public class Configured implements Configurable {
  private Configuration conf;
  /** Construct a Configured. */
  public Configured() {
    this(null);
  }
  /** Construct a Configured. */
  public Configured(Configuration conf) {
    setConf(conf);
  }
  // inherit javadoc
  public void setConf(Configuration conf) {
    this.conf = conf;
  }
  // inherit javadoc
  public Configuration getConf() {
    return conf;
  }
}
```

图 4-1. Configured 基类的代码片段

Closeable 接口的代码如图 4-2 所示：

批注 [BL11]: 这个接口的作用是什么？

```
public interface Closeable {
      /**
       * Closes this stream and releases any system resources associated
       * with it. If the stream is already closed then invoking this
       * method has no effect.
       *
       * @throws IOException if an I/O error occurs
       */
      public void close() throws IOException;
}
```

图4-2. Closeable接口代码片段

5.2.2 FileSystem 的内部类和属性

图 4-3. FileSystem 的内部类

5.2.2.1 Cache

FileSystem 内部类 Cache 用来缓存文件系统对象。

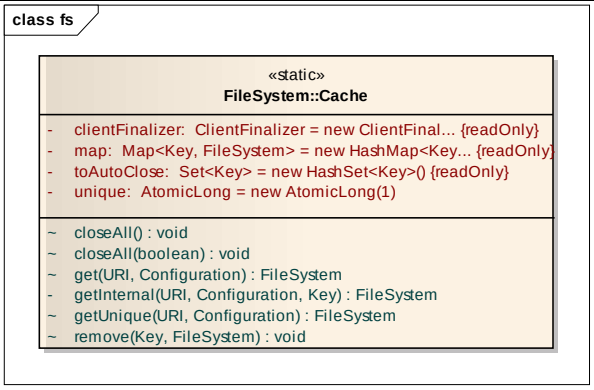

图 4-4. Cache 类

在检索文件系统时，如果缓存未被禁用，则会首先从缓存中读取。

Cache 中有两个内部类，ClientFinalizer 和 Key。

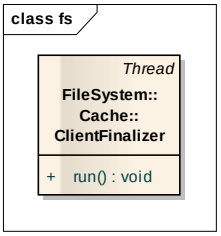

图 4-5 ClientFinalizer 类

ClientFinalizer 类为一线程类，当 Java 虚拟机停止运行时，该线程才会运行。而运行时，run 方法会调用 Cache.closeAll(true)方法，进行清理工作。

内部静态类 Key，顾名思意，它作为 Cache 中 HashMap<Key, FileSystem> 的关键字。保存了有关文件系统的 Uri 的信息，而其中的各个方法也是简单明了。

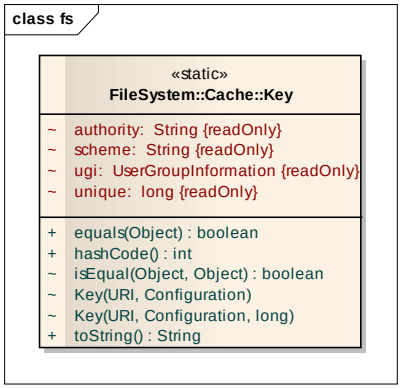

图 4-6 Key 类

集合 toAutoClose 属性用来表示是否需要自动关闭 Key 所对应的文件系统。

Cache 方法 get()和 getUnique()内部仅简单地调用了 getInternal()方法。其代码如图 4-7 所示：

```
private FileSystem getInternal(URI uri, Configuration conf, Key key) throws
IOException{
        FileSystem fs;
        synchronized (this) {
          fs = map.get(key);
        }
        if (fs != null) { //  缓存中已有
          return fs;
        }
        fs = createFileSystem(uri, conf); //创建一个新的文件系统，并初始化
        synchronized (this) { //refetch the lock again
          FileSystem oldfs = map.get(key);
          if (oldfs != null) { //a file system is created while lock is releasing  别人创建了
            fs.close(); // close the new file system
            return oldfs;    // return the old file system
          }
          // now insert the new file system into the map
          if (map.isEmpty() && !clientFinalizer.isAlive()) {
            Runtime.getRuntime().addShutdownHook(clientFinalizer);
          }
          fs.key = key;
          map.put(key, fs);
          if (conf.getBoolean("fs.automatic.close", true)) {
            toAutoClose.add(key); //  该文件系统需要自动关闭
          }
          return fs;
        }
```

```
  private static FileSystem createFileSystem(URI uri, Configuration conf
      ) throws IOException {
    Class<?> clazz = conf.getClass("fs." + uri.getScheme() + ".impl", null);
    if (clazz == null) {
      throw new IOException("No FileSystem for scheme: " + uri.getScheme());
    }
    FileSystem fs = (FileSystem)ReflectionUtils.newInstance(clazz, conf);
    fs.initialize(uri, conf);
    return fs;
  }
```

图 4-7 getInternal 代码片段

ReflectionUtils.newInstance()则利用 Java 的反射机制，调用 clazz 的构造函数，设置配置文件后，将生成的对象返回。

方法 synchronized void remove(Key key, FileSystem fs)用来从映射 map 中删除相应的 key 和 fs 对应的映射。

方法 synchronized void closeAll(boolean onlyAutomatic) throws IOException 用来删除所有的映射，并调用文件系统的 close()方法。当 onlyAutomatic 为 true 时，仅删除在集合 toAutoClose 中含有的键值对。

5.2.2.1 Statistics

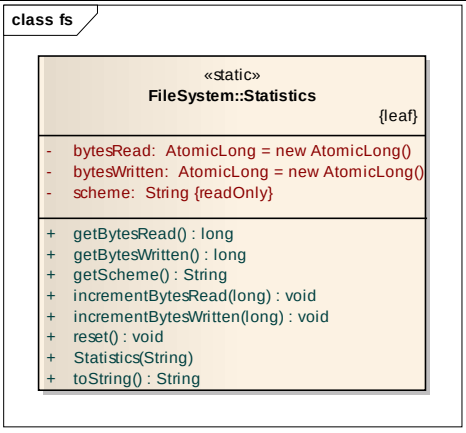

图 4-8. Statistics 类

内部类 Statistics 用来保存与一个文件系统相关的统计信息，主要包括从该文件系统读取和写入的总的字节数。每一个文件系统都有一个与之相关的 protected Statistics statistics 属性。

5.2.2.2 GlobFilter

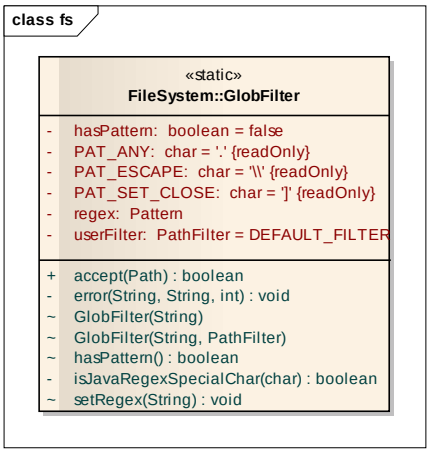

图 4-9. GlobFilter 类

内部类 GlobFilter 实现了接口 PathFilter，该类判断一个路径是否匹配指定模式。该模式通过构造函数 GlobFilter(String filePattern)和 GlobFilter(String filePattern, PathFilter filter)中的 filePattern 指定，经过检查是否有效之后，再通过 Pattern.compile 编译，并将结果存于属性 regex 中。

5.2.3 文件系统的获取

FileSystem 是一个普通的文件系统 API，所以首要任务是检索我们要用的文件系统实例。取得 FileSystem 实例有三种静态工厂方法 get()。这些 get()方法是 Hadoop 0.21 版本之前就存在的，但在 0.21 版本中又添加了与之功能相同的三个方法 newInstance()。

get()方法如下：

```
/**
 * Get a filesystem instance based on the uri, the passed
 * configuration and the user
 * @param uri
 * @param conf
 * @param user
 * @return the filesystem instance
 * @throws IOException
 * @throws InterruptedException
 */
public static FileSystem get(final URI uri, final Configuration conf,
        final String user) throws IOException, InterruptedException {
   UserGroupInformation ugi;
   if (user == null) {
      ugi = UserGroupInformation.getCurrentUser();
   } else {
      ugi = UserGroupInformation.createRemoteUser(user);
   }
   return ugi.doAs(new PrivilegedExceptionAction<FileSystem>() {
      public FileSystem run() throws IOException {
        return get(uri, conf);
      }
   });
}
/** Returns the configured filesystem implementation.*/
public static FileSystem get(Configuration conf) throws IOException {
   return get(getDefaultUri(conf), conf);
}
```

图 4-10 get 方法代码片段 1

```
/** Returns the FileSystem for this URI's scheme and authority.  The scheme
 * of the URI determines a configuration property name,
 * <tt>fs.<i>scheme</i>.class</tt> whose value names the FileSystem class.
 * The entire URI is passed to the FileSystem instance's initialize method.
 */
public static FileSystem get(URI uri, Configuration conf) throws IOException {
   String scheme = uri.getScheme();
   String authority = uri.getAuthority();

   if (scheme == null) {                          // no scheme: use default FS
      return get(conf);
   }

   if (authority == null) {                       // no authority
      URI defaultUri = getDefaultUri(conf);
      if (scheme.equals(defaultUri.getScheme()) // if scheme matches default
          && defaultUri.getAuthority() != null) {  // & default has authority
        return get(defaultUri, conf);                // return default
      }
   }

   String disableCacheName = String.format("fs.%s.impl.disable.cache", scheme);
   if (conf.getBoolean(disableCacheName, false)) {//  禁用了cache
      return createFileSystem(uri, conf); //  创建一个新的FileSystem
   }
   return CACHE.get(uri, conf);//  否则，从cache中查找
}
```

图 4-11 get 方法代码片段 2

各个 newInstance()方法分别对应一个 get()方法，完成的功能也是一样。

5.2.4 文件系统的关闭

有两个方法来关闭文件系统。closeAll()用来关闭所有的文件系统。而 close()只是关闭当前调用该方法的文件系统，如图 4-12 所示。

```
public void close() throws IOException {
   // delete all files that were marked as delete-on-exit.
   processDeleteOnExit();
   CACHE.remove(this.key, this);
}
public static void closeAll() throws IOException {
   CACHE.closeAll();
}
```

图 4-12 closeAll 代码片段

5.2.5 读取数据

数据的读取就像 Java 中的 io 一样，首先要获得一个输入流。输入流是通过 open() 方法获得的。抽象方法 open() 将由各个不同的文件系统重写，如图 4-13 所示。

```
/**
 * Opens an FSDataInputStream at the indicated Path.
 * @param f the file name to open
 * @param bufferSize the size of the buffer to be used.
 */
public abstract FSDataInputStream open(Path f, int bufferSize)
   throws IOException;
/**
 * Opens an FSDataInputStream at the indicated Path.
 * @param f the file to open
 */
public FSDataInputStream open(Path f) throws IOException {
   return open(f, getConf().getInt("io.file.buffer.size", 4096));
}
```

图 4-13 open 代码片段

图 4-14 为 KosmosFileSystem 的 open 实现：

```
    @Override
    public FSDataInputStream open(Path path, int bufferSize) throws IOException
{
        if (!exists(path))
            throw new IOException("File does not exist: " + path);
        Path absolute = makeAbsolute(path);
        String srep = absolute.toUri().getPath();
        return kfsImpl.open(srep, bufferSize);
}
```

图 4-14 KosmosFileSystem open 代码片段

其中 kfsImpl 为 KFSImpl 类型的。KFS 文件系统实现了自己的输入流 KFSInputStream 和输出流 KFSOutputStream。下面为 KFSImpl.open 的实现。

```
public FSDataInputStream open(String path, int bufferSize) throws IOException {
        return new FSDataInputStream(new KFSInputStream(kfsAccess, path,
statistics));
}
```

图 4-15 KFSImpl.open 代码片段

其他文件系统的实现类似。

5.2.6 写入数据

向文件系统中写入数据，或新建一个文件，需要获得一个用来写的输出流，这可以通过以下的多个 create()方法来实现。这些重载的方法允许我们指定是否强制覆盖已有的文件、文件副本数量、写入文件时的缓冲大小、文件块大小以及文件许可等等。

```
public FSDataOutputStream create(Path f) throws IOException {
  return create(f, true);
}
public FSDataOutputStream create(Path f, boolean overwrite)//是否覆盖
  throws IOException {
  return create(f, overwrite,
                getConf().getInt("io.file.buffer.size", 4096),
                getDefaultReplication(),
                getDefaultBlockSize());
}
public FSDataOutputStream create(Path f, Progressable progress) throws IOException {
  return create(f, true,
                getConf().getInt("io.file.buffer.size", 4096),
                getDefaultReplication(),
                getDefaultBlockSize(), progress);
}
public FSDataOutputStream create(Path f, short replication)//文件副本数量
  throws IOException {
  return create(f, true,
                getConf().getInt("io.file.buffer.size", 4096),
                replication,
                getDefaultBlockSize());
}
public FSDataOutputStream create(Path f, short replication, Progressable progress)
  throws IOException {
  return create(f, true,
                getConf().getInt("io.file.buffer.size", 4096),
                replication,
                getDefaultBlockSize(), progress);
}
public FSDataOutputStream create(Path f,
                                 boolean overwrite,
                                 int bufferSize
                                 ) throws IOException {
  return create(f, overwrite, bufferSize,
                getDefaultReplication(),
                getDefaultBlockSize());
}
```

图 4-16 create 代码片段 1

```
  public FSDataOutputStream create(Path f,
                                   boolean overwrite,
                                   int bufferSize,
                                   Progressable progress
                                   ) throws IOException {
    return create(f, overwrite, bufferSize,
                  getDefaultReplication(),
                  getDefaultBlockSize(), progress);
  }
  public FSDataOutputStream create(Path f,
                                   boolean overwrite,
                                   int bufferSize,
                                   short replication,
                                   long blockSize
                                   ) throws IOException {
    return create(f, overwrite, bufferSize, replication, blockSize, null);
  }
  public FSDataOutputStream create(Path f,
                                   boolean overwrite,
                                   int bufferSize,
                                   short replication,
                                   long blockSize,
                                   Progressable progress
                                   ) throws IOException {
    return this.create(f, FsPermission.getDefault(), overwrite, bufferSize,
        replication, blockSize, progress);
  }
  public abstract FSDataOutputStream create(Path f,
      FsPermission permission,
      boolean overwrite,
      int bufferSize,
      short replication,
      long blockSize,
      Progressable progress) throws IOException;
    /**
   * Creates the given Path as a brand-new zero-length file.   If
   * create fails, or if it already existed, return false.*/
  public boolean createNewFile(Path f) throws IOException {
    if (exists(f)) { return false;
    } else {
      create(f, false, getConf().getInt("io.file.buffer.size", 4096)).close();
      return true;
    }
}
```

图 4-17 create 代码片段 2

由此可见，各个 create 方法最终都是调用了抽象方法 create。

```
  /**
     * Opens an FSDataOutputStream at the indicated Path with
write-progress
     * reporting.
     * @param f the file name to open
     * @param permission
     * @param overwrite if a file with this name already exists, then if true,
     *     the file will be overwritten, and if false an error will be thrown.
     * @param bufferSize the size of the buffer to be used.
     * @param replication required block replication for the file.
     * @param blockSize
     * @param progress
     * @throws IOException
     * @see #setPermission(Path, FsPermission)
     */
    public abstract FSDataOutputStream create(Path f,
          FsPermission permission,
          boolean overwrite,
          int bufferSize,
          short replication,
          long blockSize,
          Progressable progress) throws IOException;
```

图 4-18 抽象方法 create 代码片段

还有一个用于传递回调接口的重载方法 Progressable，如此一来，我们所写的应用就会被告知数据写入数据节点的进度，Progressable 接口的代码片段如图 4-19 所示：

```
public interface Progressable {
  /**
   * Report progress to the Hadoop framework.
   */
  public void progress();
}
```

图 4-19 接口 Progressable 代码片段

新建文件的另一种方法是使用 append()在一个已有文件中追加(也有一些其他重载版本)。由下图可见，重要的是抽象方法 append()。同抽象方法 create 一样，各个派生类需要重写。

```
  public FSDataOutputStream append(Path f) throws IOException {
    return append(f, getConf().getInt("io.file.buffer.size", 4096), null);
  }
  public FSDataOutputStream append(Path f, int bufferSize) throws IOException {
    return append(f, bufferSize, null);
  }
  /**
   * Append to an existing file (optional operation).
   * @param f the existing file to be appended.
   * @param bufferSize the size of the buffer to be used.
   * @param progress for reporting progress if it is not null.
   * @throws IOException
   */
  public abstract FSDataOutputStream append(Path f, int bufferSize,
      Progressable progress) throws IOException;
```

图 4-20 append 函数代码片段

同 open 一样，不同的文件系统的实现会重新改函数的实现。

5.2.7 文件操作

5.2.7.1 重命名

```
/**
 * Renames Path src to Path dst.    Can take place on local fs
 * or remote DFS.
 * @throws IOException on failure
 * @return true if rename is successful
 */
public abstract boolean rename(Path src, Path dst) throws IOException;
@Deprecated
protected void rename(final Path src, final Path dst,
      final Rename... options) throws IOException
```

图 4-21 rename 函数代码片段

文件重命名使用抽象方法 rename()实现，其代码如图 4-21 所示。0.21 版本以前使用非抽象方法 rename()方法实现。但现在该方法已经不再推荐使用。

5.2.7.2　文件删除

```
/** Delete a file.
 *
 * @param f the path to delete.
 * @param recursive if path is a directory and set to
 * true, the directory is deleted else throws an exception. In
 * case of a file the recursive can be set to either true or false.
 * @return    true if delete is successful else false.
 * @throws IOException
 */
public abstract boolean delete(Path f, boolean recursive) throws IOException;
/**
 * Delete a file
 * @deprecated Use {@link #delete(Path, boolean)} instead.
 */
@Deprecated
public boolean delete(Path f) throws IOException
```

图 4-22 delete 函数代码片段

文件删除使用抽象方法 delete()实现，其代码如图 4-22 所示，0.21 版本以前使用非抽象方法 delete()方法实现。但现在该方法已经不再推荐使用。

5.2.7.3　文件或路径测试

```
/** Check if exists.
 * @param f source file
 */
public boolean exists(Path f) throws IOException {
  try {
    return getFileStatus(f) != null;
  } catch (FileNotFoundException e) {
    return false;
  }
}
/** True iff the named path is a directory.
 * Note: Avoid using this method. Instead reuse the FileStatus
 * returned by getFileStatus() or listStatus() methods.
 */
public boolean isDirectory(Path f) throws IOException {
  try {
    return getFileStatus(f).isDirectory();
  } catch (FileNotFoundException e) {
    return false;                // f does not exist
  }
}

/** True iff the named path is a regular file.
 * Note: Avoid using this method. Instead reuse the FileStatus
 * returned by getFileStatus() or listStatus() methods.
 */
public boolean isFile(Path f) throws IOException {
  try {
    return getFileStatus(f).isFile();
  } catch (FileNotFoundException e) {
    return false;                // f does not exist
  }
}
```

图 4-23 文件判断代码片段

5.2.7.4 文件复制

copyFromLocalFile 来实现将文件从本地复制到 HDFS，FileSystm 中有多个重载的 copyFromLocalFile

public void copyFromLocalFile(Path src, Path dst)

public void copyFromLocalFile(boolean delSrc, Path src, Path dst)

public void copyFromLocalFile(boolean delSrc, boolean overwrite, Path[] srcs, Path dst)

public void copyFromLocalFile(boolean delSrc, boolean overwrite, Path src, Path dst)

copyToLocalFile 来实现将文件从 HDFS 复制到本地。

public void copyToLocalFile(Path src, Path dst) throws IOException

public void copyToLocalFile(boolean delSrc, Path src, Path dst)

moveFromLocalFile 来实现将文件从本地移动到 HDFS。

public void moveFromLocalFile(Path[] srcs, Path dst)

public void moveFromLocalFile(Path src, Path dst)

moveToLocalFile 来实现将文件从 HDFS 移动到本地。

public void moveToLocalFile(Path src, Path dst) throws IOException

这些方法最终都是调用 FileUtil.copy()实现的。

5.2.8 查询文件系统

5.2.8.1 文件元数据：FileStatus

任何文件系统的一个重要特征是定位其目录结构及检索其存储的文件和目录信息的能力。FileStatus 是一个简单的类，封装了文件系统中文件和目录的元数据，包括文件长度、块大小、副本、修改时间、所有者以及许可信息。FileStatus 实现了 Writeable 接口，可以序列化。

图 4-24 FileStatus 类

FileSystem 的 getFileStatus()提供了获取一个文件或目录的状态对象的方法。

5.2.8.2 列出文件

查找一个文件或目录的信息很实用，但有时我们还需要能够列出目录的内容。这就是 listStatus()方法的功能：

1. public abstract FileStatus[] listStatus(Path f) throws IOException
2. public FileStatus[] listStatus(Path f, PathFilter filter) throws IOException
3. public FileStatus[] listStatus(Path[] files) throws IOException
4. public FileStatus[] listStatus(Path[] files, PathFilter filter) throws IOException

传入参数是一个文件时，它会简单地返回长度为 1 的 FileStatus 对象的一个数组。当传入参数是一个目录时，它会返回 0 或者多个 FileStatus 对象，代表着此目录所包含的文件和目录。

重载方法允许我们使用 PathFilter 来限制匹配的文件和目录。如果把路径数组作为参数来调用 listStatus 方法，其结果是与依次对每个路径调用此方法，再将 FileStatus 对象数组收集在一个单一数组中的结果是相同的，但是前者更为方便。这在建立从文件系统树的不同部分执行的输入文件的列表时很有用。

5.2.8.3　文件格式

在一步操作中处理批量文件，这个要求很常见。举例来说，处理日志的 MapReduce 作业可能会分析一个月的文件，这些文件被包含在大量目录中。Hadoop 有一个通配的操作，可以方便地使用通配符在一个表达式中核对多个文件，不需要列举每个文件和目录来指定输入。Hadoop 为执行通配提供了两个 FileSystem 方法：

1. public FileStatus[] globStatus(Path pathPattern) throws IOException

2. public FileStatus[] globStatus(Path pathPattern,PathFilter filter) throws IOException

globStatus()返回了其路径匹配于所供格式的 FileStatus 对象数组，按路径排序。可选的 PathFilter 命令可以进一步指定限制匹配

通配符	名称	匹配
*	星号	匹配 0 或多个字符
?	问号	匹配单一字符
[ab]	字符类别	匹配{a,b}中的一个字符

图 4-25 通配符 1

通配符	名称	匹配
[^ab]	非字符类别	匹配不是{a,b}中的一个字符
[a-b]	字符范围	匹配一个在{a,b}范围内的字符(包括 ab)，a 在字典顺序上要小于或等于 b
[^a-b]	非字符范围	匹配一个不在{a,b}范围内的字符(包括 ab)，a 在字典顺序上要小于或等于 b
{a,b}	或选择	匹配包含 a 或 b 中的一个的语句
\c	转义字符	匹配元字符 c

图 4-26 通配符 2

PathFilter 对象

通配格式不是总能够精确地描述我们想要访问的文件集合。比如，使用通配格式排除一个特定的文件就不太可能。FileSystem 中的 listStatus()和 globStatus()方法提供了可选的 PathFilter 对象，使我们能够通过编程方式控制匹配：

```
public interface PathFilter {
  boolean accept(Path path);
}
```

图 4-27 PathFilter 代码片段

5.2.9　其它方法

1. 默认的文件系统 uri 通过 public static URI getDefaultUri(Configuration conf)来获取；通过 public static void setDefaultUri(Configuration conf, URI uri) 和 public static void setDefaultUri(Configuration conf, String uri)来设置。

2. 当一个文件系统实例被创建后，就需要调用 initialize 来初始化。其默认实现仅仅是设置了 statistics 属性。
3. public abstract URI getUri() 用来获取该文件系统对应的 Uri。
4. public static LocalFileSystem getLocal(Configuration conf) 获得本地文件系统。在 0.21 版本中出现了一个新的方法 newInstanceLocal()，该方法与 getLocal 完成完全一样的功能。
5. public BlockLocation[] getFileBlockLocations(Path p, long start, long len) throws IOException
6. public BlockLocation[] getFileBlockLocations(FileStatus file, long start, long len) throws IOException 返回 file 中从 start 开始 len 长度的数据所在的块
7. public FsServerDefaults getServerDefaults() throws IOException 得到服务器的默认配置。
8. public boolean setReplication(Path src, short replication)
9. public Path getHomeDirectory()

等等。

5.3 FilterFileSystem

FilterFileSystem 的类图如下：

图 4-28 FilterFileSystem 类

FilterFileSystem 类包含了一个其它的文件系统的实例 fs，并将其作为基本的文件系统。FilterFileSystem 类几乎将所有重写的方法交给了其内部保存的 fs 来处理。但在交给 fs 处理之前，自己可以做一些处理，以此来实现过滤。如图 4-29 所示：

```
public void initialize(URI name, Configuration conf) throws IOException {
    fs.initialize(name, conf); //交给fs处理
  }
  public URI getUri() {
    return fs.getUri(); //交给fs处理
  }
// 其余的方法类似
```

图 4-29 代码分析

5.4 ChecksumFileSystem

ChecksumFileSystem 类为文件系统提供了校验和的功能。

图 4-30 ChecksumFileSystem 类

HDFS 以透明方式校验所有写入它的数据，并在默认设置下，会在读取数据时验证校验和。针对数据的每个 io.bytes.per.checksum 字节，都会创建一个单独的校验和。默认值为 512 字节，使用 CRC-32 校验和，存储开销为 1%。

每一个原始文件都有一个校验和文件，比如文件 /a/b/c.txt 对应的校验和文件为 /a/b/.c.txt.crc。校验和文件默认是隐藏的文件，即在文件名“c.txt”前加上一个“.”，并在文件名“c.txt”后面加上后缀“.crc”。

ChecksumFileSystem 类中的属性 CHECKSUM_VERSION 为写入校验和文件的文件头中的版本号。verifyChecksum 表示是否需要检测校验和。

```
/** get the raw file system */
public FileSystem getRawFileSystem() {
  return fs; // 返回基类FilterFileSystem的fs属性
}
/** Return the name of the checksum file associated with a file.*/
public Path getChecksumFile(Path file) {
  return new Path(file.getParent(), "." + file.getName() + ".crc");
}
```

图 4-31 代码分析

ChecksumFileSystem 中有两个内部类 ChecksumFSInputChecker 和 ChecksumFSOutputSummer。详见后面分析。

ChecksumFileSystem 类 open 和 create 方法的实现如下，仅是简单的创建一个 FSDataInputStream 和 FSDataOutputStream。

```
public FSDataInputStream open(Path f, int bufferSize) throws IOException {
  return new FSDataInputStream(
      new ChecksumFSInputChecker(this, f, bufferSize));
}
public FSDataOutputStream create(Path f, FsPermission permission,
    boolean overwrite, int bufferSize, short replication, long blockSize,
    Progressable progress) throws IOException {
  Path parent = f.getParent();
  if (parent != null && !mkdirs(parent)) {
    throw new IOException("Mkdirs failed to create " + parent);
  }
  final FSDataOutputStream out = new FSDataOutputStream(
      new ChecksumFSOutputSummer(this, f, overwrite, bufferSize, replication,
          blockSize, progress), null);
  if (permission != null) {
    setPermission(f, permission);
  }
  return out;
}
```

图 4-32 代码分析

5.5 LocalFileSystem

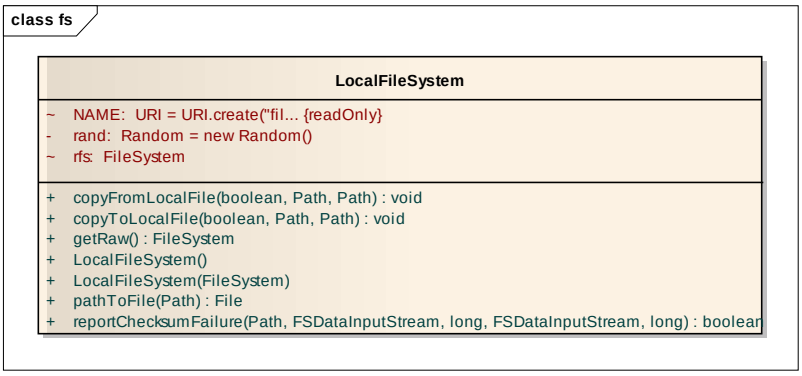

图 4-33 LocalFileSystem 类

LocalFileSystem 从 ChecksumFileSystem 类派生，主要实现其它一些支持校验和文件系统的 API。LocalFileSystem 有一个属性 rfs，用来表原生的文件系统。LocalFileSystem 重写了一些方法，如 copyFromLocalFile 和 copyToLocalFile 等。

6 迥凇迥剜涣割秘

6.1 FSInputStream 挟賃糁

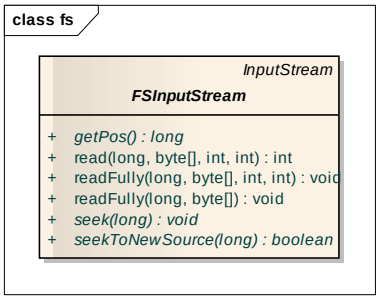

图 5-1 FSInputStream 类

FSInputStream 实现了接口 Seekable 和 PositionedReadable。提供了基本的读取一个输入流的操作，比如定位 seek 和读取到特定的 buffer 的操作 read 和 readFully。FSInputStream 在原有 InputStream 基础之上添加了 getPos 方法，同时可以通过 seek 方法定位指定的偏移量处。新添加的 getPos 和 seek 方法在 FSDataInputStream 类中被使用。如图 5-2 所示：

```
public interface Seekable {
  /**
   * Seek to the given offset from the start of the file.
   * The next read() will be from that location.  Can't
   * seek past the end of the file.
   */
  void seek(long pos) throws IOException;

  /**
   * Return the current offset from the start of the file
   */
  long getPos() throws IOException;

  /**
   * Seeks a different copy of the data.  Returns true if
   * found a new source, false otherwise.
   */
  @InterfaceAudience.Private
  boolean seekToNewSource(long targetPos) throws IOException;
}

public interface PositionedReadable {
  /**
   * Read upto the specified number of bytes, from a given
   * position within a file, and return the number of bytes read. This does not
   * change the current offset of a file, and is thread-safe.
   */
  public int read(long position, byte[] buffer, int offset, int length)
    throws IOException;

  /**
   * Read the specified number of bytes, from a given
   * position within a file. This does not
   * change the current offset of a file, and is thread-safe.
   */
  public void readFully(long position, byte[] buffer, int offset, int length)
    throws IOException;

  /**
   * Read number of bytes equalt to the length of the buffer, from a given
   * position within a file. This does not
   * change the current offset of a file, and is thread-safe.
   */
  public void readFully(long position, byte[] buffer) throws IOException;
}
```

图 5-2 代码分析

各个不同的文件系统会从该类派生，重写相应的抽象方法。如：

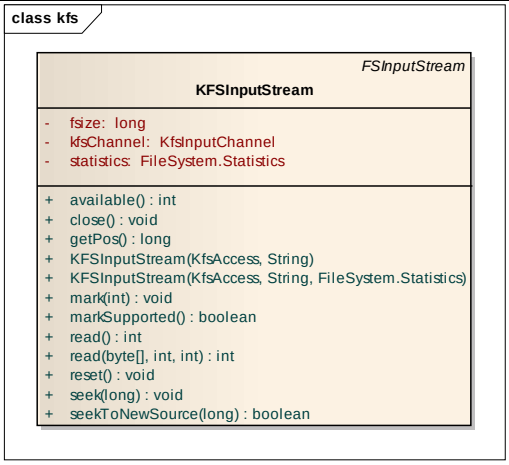

图 5-3 KFSInputStream 类

KFSInputStream 类重写了 InputStream 接口中的各个方法。实现使用了外部的 kfs-0.3.jar，该 jar 提供了访问 KFS 文件系统的 API。在 KFSOutputStream 中，有属性

org.kosmix.kosmosfs.access.KfsOutputChannel kfsChannel;

在方法的实现中，KFSOutputStream 仅是简单的将有关操作转交给了 kfsChannel 来处理。例如：

```
public long getPos() throws IOException {
    if (kfsChannel == null) {
        throw new IOException("File closed");
     }
    return kfsChannel.tell();
}
```

图 5-4 代码分析

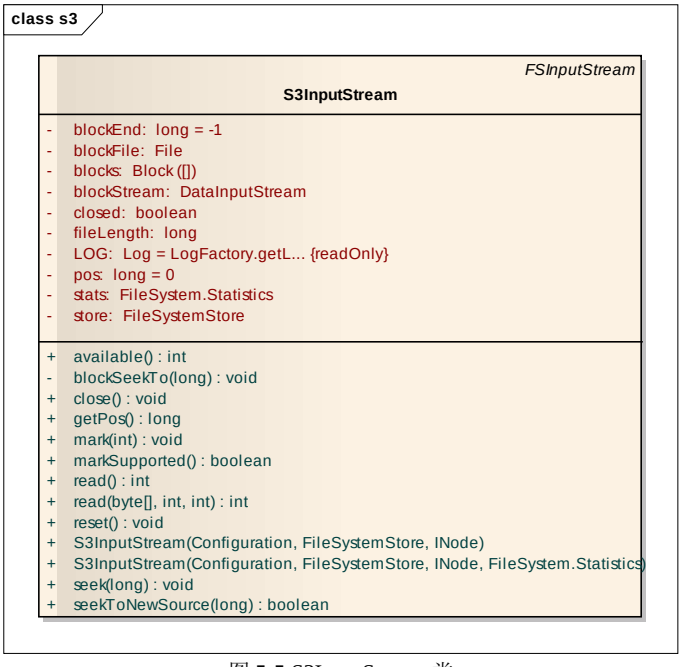

图 5-5 S3InputStream 类

S3InputStream 类重写了 InputStream 接口中的各个方法。

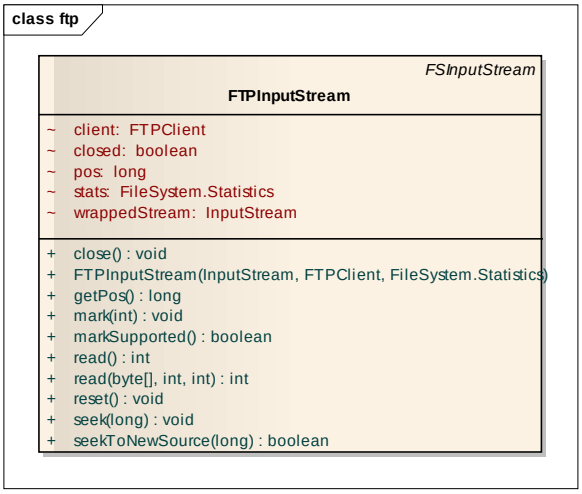

图 5-6 FTPInputStream 类

FTPInputStream 类重写了 InputStream 接口中的各个方法。

图 5-7 DFSInputStream 类

DFSInputStream 类重写了 InputStream 接口中的各个方法。

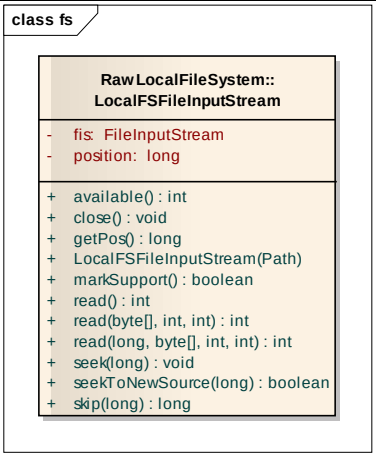

图 5-8 LocalFSFileInputStream 类

LocalFSFileInputStream 类重写了 InputStream 接口中的各个方法。

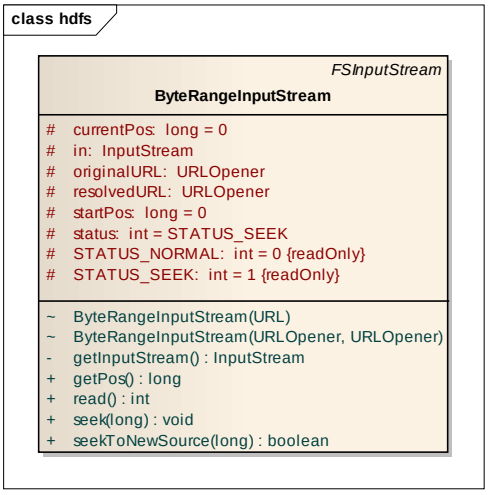

图 5-9 ByteRangeInputStream 类

ByteRangeInputStream 类重写了 InputStream 接口中的各个方法。

6.2 迥剜涣

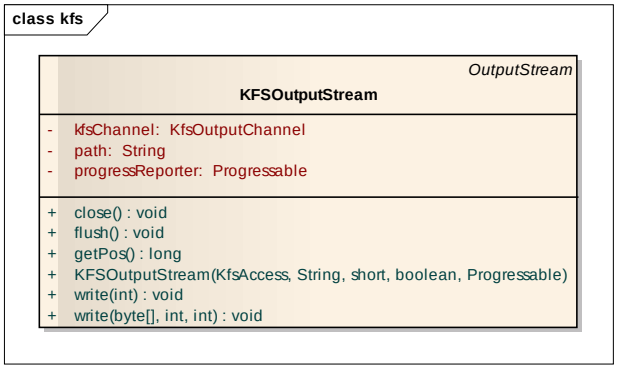

图 5-10 KFSOutputStream 类

KFSOutputStream 的实现使用了外部的 kfs-0.3.jar，该 jar 提供了访问 KFS 文件系统的 API。在 KFSOutputStream 中，有属性

org.kosmix.kosmosfs.access.KfsOutputChannel kfsChannel;

org.kosmix.kosmosfs.access.Progressable progressReporter;

在方法的实现中，KFSOutputStream 仅是简单的将有关操作转交给了 kfsChannel 和 progressReporter 来处理。例如：

```
public long getPos() throws IOException {
    if (kfsChannel == null) {
        throw new IOException("File closed");
    }
    return kfsChannel.tell();
}
public void write(byte b[], int off, int len) throws IOException {
    if (kfsChannel == null) {
        throw new IOException("File closed");
    }
    // touch the progress before going into KFS since the call can
block
    progressReporter.progress();
    kfsChannel.write(ByteBuffer.wrap(b, off, len));
    }
```

图 5-11 代码分析

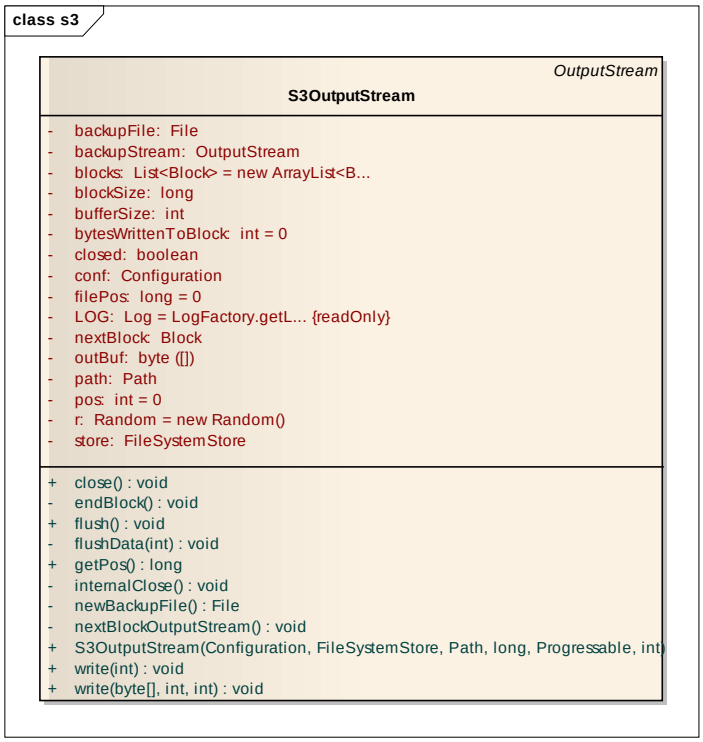

图 5-12 S3OutputStream 类

S3OutputStream 类提供了缓冲的功能，在写数据时，首先将数据写到自身的缓冲区 outBuf 中。待到缓冲区满或显式调用 flush，才将数据写入到一个备份文件 backupStream 中。当写入到 backupStream 的数据量为一个 Block 块的大小时，调用 FileSystemStore.storeBlock，将该 backupStream 中的数据作为一个块，存储到 S3 分布式文件系统。FileSystemStore 仅仅是一个接口，实际调用的是 Jets3tFileSystemStore 的方法。Jets3tFileSystemStore 利用 Jets3t-0.7.1.jar 包实现了对 S3 文件系统的封装。

6.3 FSInputChecker

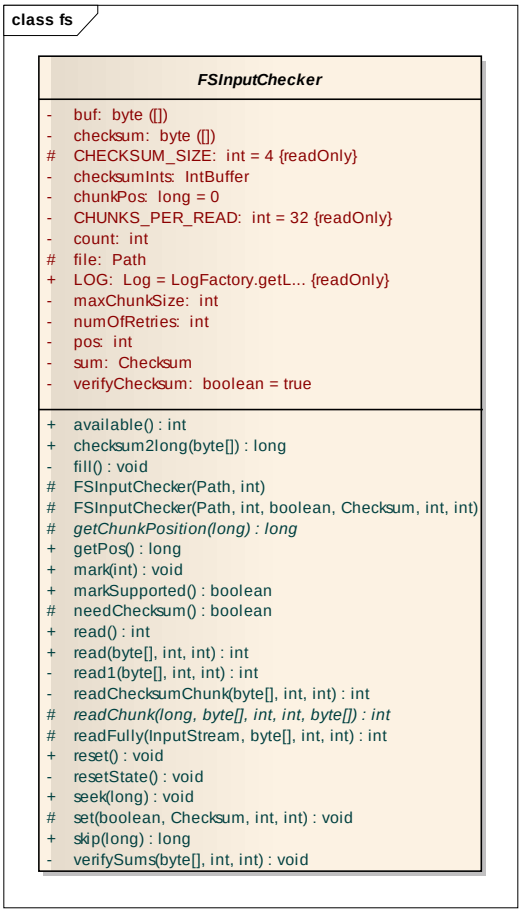

图 5-13 FSInputChecker 类

FSInputChecker 提供了检测校验和的功能，主要实现了 read 方法。在 read 方法中调用了 read1 方法。每当读取的数据字节数大于 maxChunkSize 时，就会调用 readChecksumChunk 方法。在 readChecksumChunk 方法中，进一步调用 readChunk 抽象方法来实现真正的数据读取。读取完数据后，然后判断是否需要检验校验和，如果需要，则调用 private void verifySums(final byte b[], final int off, int read)throws ChecksumException 进行检验，如果有错，则抛出异常。

abstract protected int readChunk(long pos, byte[] buf, int offset, int len,
byte[] checksum) throws IOException;

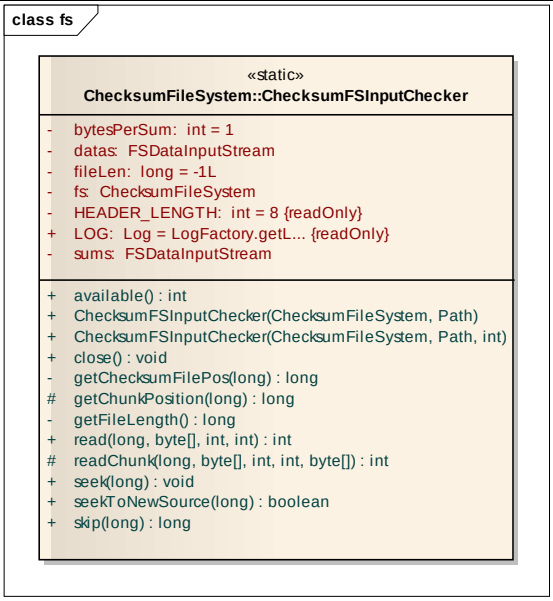

图 5-14 ChecksumFSInputChecker 类

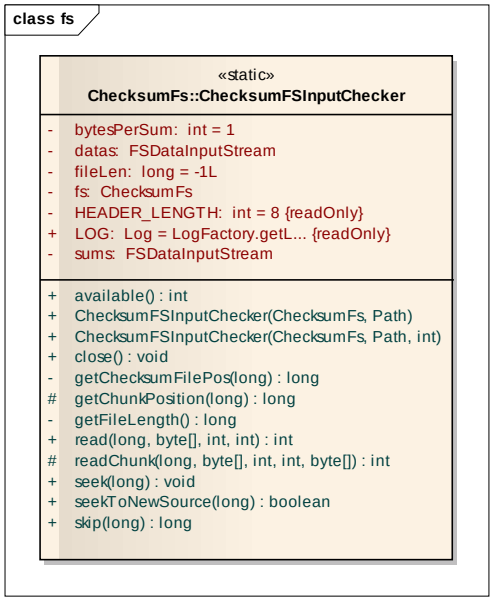

图 5-15 ChecksumFSInputChecker 类

ChecksumFs. ChecksumFSInputChecker 和 ChecksumFileSystem.ChecksumFSInputChecker 的实现非常类似。似乎前者是 0.21 版本中的新 api，是用来取代以前版本中的后者的。两者都有 FSDataInputStream 类型的属性 datas 和 sums，datas 代表一般文件的输入流，而 sums 则代表该文件对应的校验和文件的输入流。

private FSDataInputStream datas;

private FSDataInputStream sums;

在 ChecksumFileSystem.ChecksumFSInputChecker 中的 private ChecksumFileSystem fs;属性对应 ChecksumFs. ChecksumFSInputChecker 中的 private ChecksumFs fs;

这两个类都实现了 readChunk 方法。方法的实现很简单，从 datas 中读出数据，然后检查是否 needChecksum()，如果需要，则读取相应的校验和。

6.4 FSOutputSummer

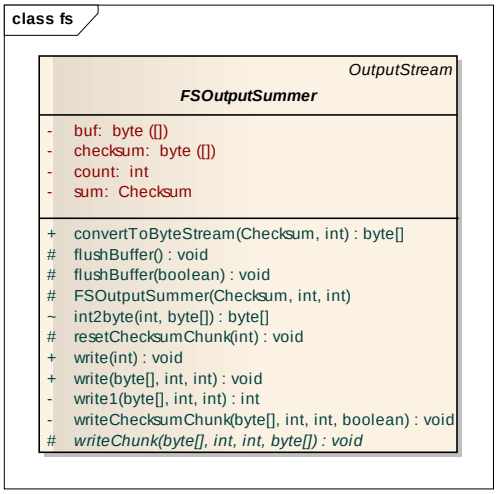

图 5-16 FSOutputSummer 类

KFSOutputStream、S3OutputStream 和 FSOutputSummer 类都是从抽象类 OutputStream 派生的。KFSOutputStream 和 S3OutputStream 都是具体类，他们重写了 OutputStream 中的抽象方法。

FSOutputSummer 也是抽象类，它实现了 OutputStream 中的 write()方法，并提供了计算校验和的功能。

FSOutputSummer 类中的属性 buf 数组用来缓存写入的数据，当数据超过 buf.length 个数时，才会调用 flushBuffer()实现真正的写入数据。

属性 sum 用来保存计算的校验和。使用类型应为 org.apache.hadoop.util.PureJavaCrc32，可以看成是一个 4 字节的整数。

一次写入一个字节时：

```
/** Write one byte */
public synchronized void write(int b) throws IOException {
  sum.update(b); //更新校验和
  buf[count++] = (byte)b;//缓存数据
  if(count == buf.length) {
    flushBuffer();// 写入数据
  }
}
```

图 5-17 代码分析

一次写入多个字节时使用 public synchronized void write(byte b[], int off, int len)方法。该方法仅是连续调用多次 private int write1(byte b[], int off, int len) throws IOException 方法。

在 write1()中，会更新校验和 sum，最终都会调用以下函数

```
 /** Generate checksum for the data chunk and output data chunk &
checksum
    * to the underlying output stream. If keep is true then keep the
    * current checksum intact, do not reset it.
    */
private void writeChecksumChunk(byte b[], int off, int len, boolean keep)
   throws IOException {
      int tempChecksum = (int)sum.getValue();
      if (!keep) { //  对一个数据块计算完校验和（实际为BytesPerSum个数
据），然后重置校验和
         sum.reset();
      }
      int2byte(tempChecksum, checksum);//将校验和写入到字节数组
checksum中
      writeChunk(b, off, len, checksum);//调用该抽象方法完成最终的写入
   }
```

图 5-18 代码分析

FSOutputSummer 类的派生类 ChecksumFs.ChecksumFSOutputSummer、ChecksumFileSystem. ChecksumFSOutputSummer 和 DFSOutputStream 都重写了 close()和 writeChunk()方法。

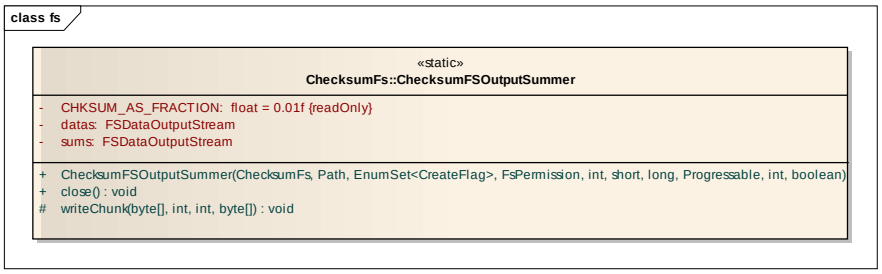

图 5-19 ChecksumFSOutputSummer 类

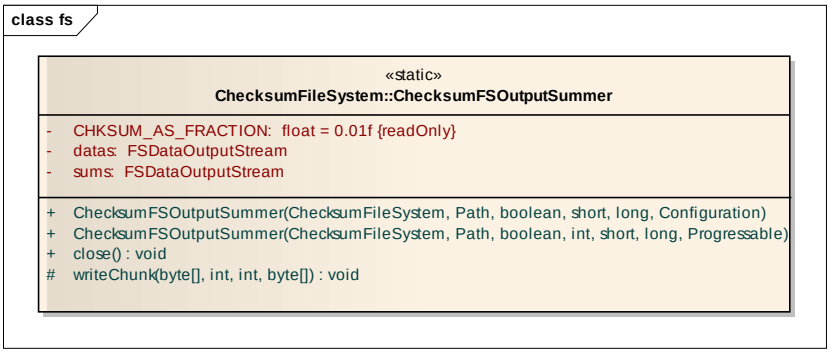

图 5-20 ChecksumFSOutputSummer 类

ChecksumFs.ChecksumFSOutputSummer 和 ChecksumFileSystem. ChecksumFSOutputSummer 类几乎完全一样。似乎前者是 0.21 版本中的新 api，是用来取代以前版本中的后者的。两者都有 FSDataOutputStream 类型的属性 datas 和 sums，datas 代表一般文件的输出流，而 sums 则代表该文件对应的校验和文件的输出流。

两个类的 close()和 writeChunk()方法的实现完全一样

```
protected void writeChunk(byte[] b, int offset, int len, byte[] checksum)
      throws IOException {
         datas.write(b, offset, len);// 写入数据
         sums.write(checksum);// 写入校验和
      }
public void close() throws IOException {
         flushBuffer();
         sums.close();
         datas.close();
      }
```

图 5-21 代码分析

6.5 FSDataInputStream

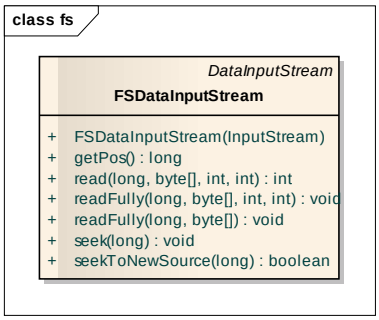

图 5-22 FSDataInputStream 类

FSDataInputStram 继承于 DataInputStream，并实现了接口 Seekable 和 PositionedReadable，提供了随机访问的功能。在构造函数 FSDataInputStream(InputStream in)中，将参数 in 赋值给 FilterInputStream.in。实际的参数 in 应为 FSInputStream 类型的。为实现接口 Seekable 和 PositionedReadable，FSDataInputStram 类只是简单的调用了属性 in 对这两个接口的实现（FSInputStream 实现了接口 Seekable 和 PositionedReadable）。如：

```
   public int read(long position, byte[] buffer, int offset, int length)
      throws IOException {
      return ((PositionedReadable)in).read(position, buffer, offset, length);
}
```

图 5-23 代码分析

BufferedFSInputStream 的实现与 FSDataInputStream 的实现极其相似。它也实现了接口 Seekable 和 PositionedReadable，提供了随机访问的功能。在构造函数中，将参数 in 赋值给 FilterInputStream.in。实际的参数 in 应为 FSInputStream 类型的。为实现接口 Seekable 和 PositionedReadable，FSDataInputStram 类只是简单的调用了属性 in 对这两个接口的实现（FSInputStream 实现了接口 Seekable 和 PositionedReadable）。如：

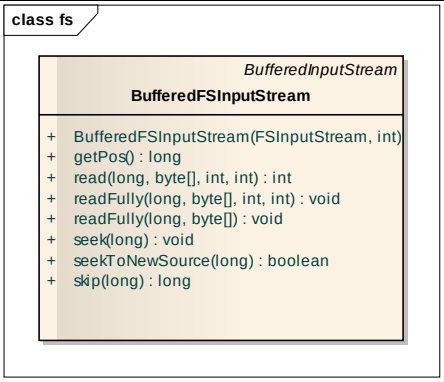

图 5-24 BufferedFSInputStream 类

6.6 FSDataOutputStream

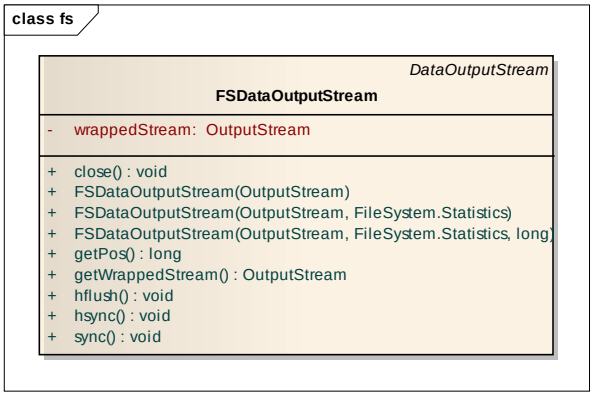

图 5-25 FSDataOutputStream 类

FSDataOutputStream 并没有实现接口 Seekable 和 PositionedReadable，而是实现了 Syncable 接口。HDFS 只允许对一个打开的文件顺序写入，后向一个已有的文件添加（append）。换句话说，它不允许除文件尾部的其他位置的写入，因此也就不需要定位。

```
public interface Syncable {
  /**
   * @deprecated As of HADOOP 0.21.0, replaced by hflush
   * @see #hflush()
   */
  @Deprecated   public void sync() throws IOException;

  /** Flush out the data in client's user buffer. After the return of
   * this call, new readers will see the data.
   * @throws IOException if any error occurs
   */
  public void hflush() throws IOException;

  /** Similar to posix fsync, flush out the data in client's user buffer
   * all the way to the disk device (but the disk may have it in its cache).
   * @throws IOException if error occurs
   */
  public void hsync() throws IOException;
}
```

图 5-26 代码分析

FSDataOutputStream 中有一个内部类 PositionCache，其从 FilterOutputStream 派生，提供了缓存文件指针 position 的功能，并且有一个 statistics 属性，用来统计。

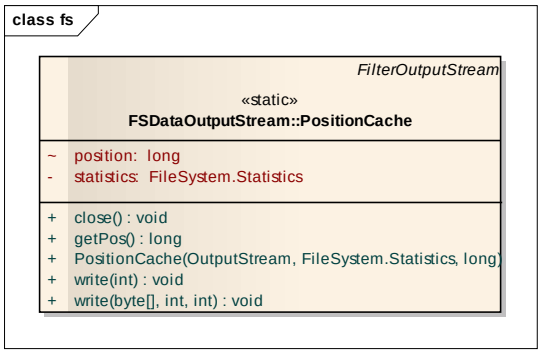

图 5-27 PositionCache 类

FSDataOutputStream 的一个属性 wrappedStream 用来保存 PositionCache 所过滤的输出流，对接口 Syncable 的实现则是通过 wrappedStream 对应的方法来完成的。如：

```
  public void hsync() throws IOException {
    if (wrappedStream instanceof Syncable) {
((Syncable)wrappedStream).hsync();
    } else {
      wrappedStream.flush();
    }
```

图 5-28 代码分析

7 AbstractFileSystem 剖析

在分析该类层次结构时，可以将 AbstractFileSystem 与 FileSystem 对应，FilterFs 与 FilerFileSystem 对应，ChecksumFs 与 ChecksumFileSystem 对应，LocalFs 与 LocalFileSystem 对应，RawLocalFs 与 RawLocalFileSystem 对应。他们完成的功能及其相似。

7.1 AbstractFileSystem 挟赁糁

AbstractFileSystem 是 0.21 版本新出现的 API，应该是用来替代 FileSystem 的。该类为 Hadoop 文件系统的实现提供了一个接口。AbstractFileSystem 是一个抽象类。 它与 FileSystem 有很多同名的方法。这些同名的方法完成相同的功能。比如 get，create 等等。其它的方法大都是抽象方法。

图 7-1 AbstractFileSystem 类

get 方法调用了 createFileSystm 方法实现。而 createFileSystm 方法实现与 FileSystm 的 createFileSystm 可以说是完全一样。

create 方法首先解析参数，然后调用 createInternal 抽象方法。

7.2 FilterFs 挟赁糁

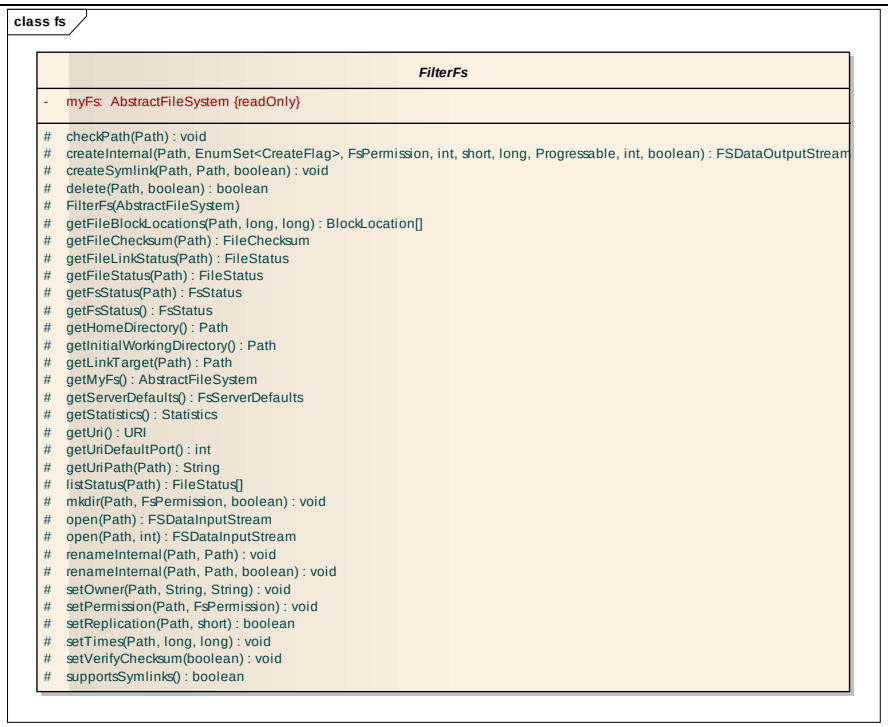

图 6-2 FilterFs 类

FilterFs 也是一个抽象类，它同 FilterFileSystem 完全一样，只是拥有一个 AbstractFileSystem 类型的属性 myFs。FilterFs 将其作为基本的文件系统。FilterFs 类几乎将所有重写的方法交给了其内部保存的 myFs 来处理。但在交给 myFs 处理之前，自己可以做一些处理，以此来实现过滤。如：

```
  protected FSDataOutputStream createInternal(Path f,
      EnumSet<CreateFlag> flag, FsPermission absolutePermission, int
bufferSize,
      short replication, long blockSize, Progressable progress,
      int bytesPerChecksum, boolean createParent)
        throws IOException, UnresolvedLinkException {
      checkPath(f);
      return myFs.createInternal(f, flag, absolutePermission, bufferSize,
          replication, blockSize, progress, bytesPerChecksum,
createParent);
    }
```

图 7-3 代码分析

7.3 ChecksumFs 挟赁糁

ChecksumFs 抽象类的实现与 ChecksumFileSystem 的实现完全一样。只是 ChecksumFs 使用的是 ChecksumFs.ChecksumFSInputStream ， 而 ChecksumFileSystem 使 用 的 是 ChecksumFileSystem.ChecksumFSInputStream。

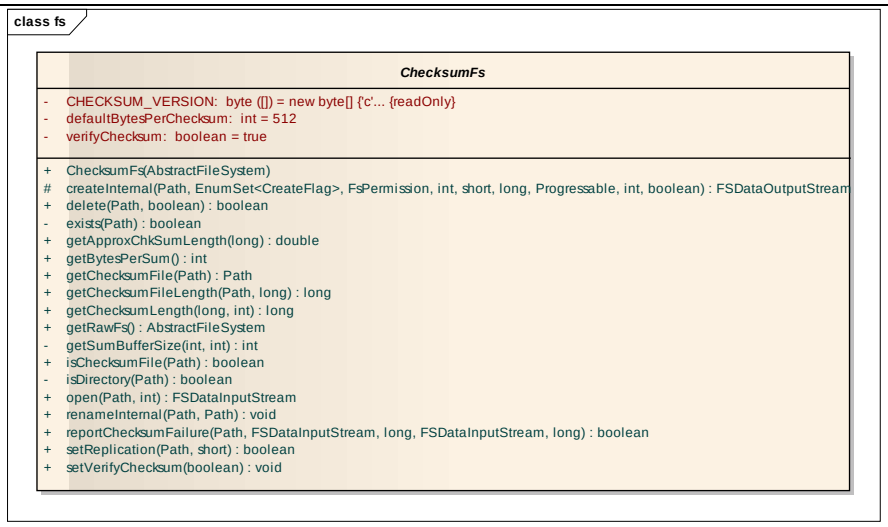

图 6-4 ChecksumFs 类

7.4 LocalFs 类

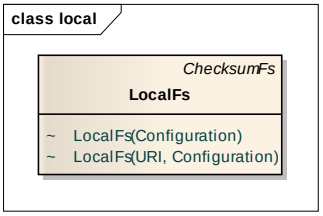

图 6-5 LocalFs 类

LocalFs 仅仅有两个构造函数。

7.5 DelegateToFileSystem 类

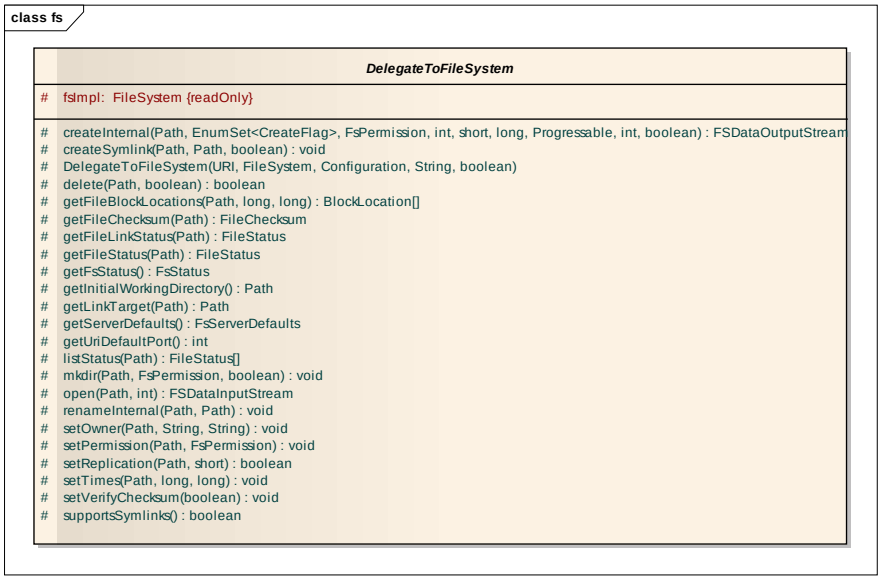

图 6-6 DelegateToFileSystem 类

顾名思义，DelegateToFileSystm 是一个代理类。它简单的将所有的操作交给 FileSystm 类型的属性 fsImpl 来处理。

目前 Hadoop 源码中只有 FtpFs 和 RawLocalFs 是从其派生的。

7.6 FileContext 糝

FileContext 类是 0.21 版中提供的新的 API，它为应用程序编写者提供了访问 Hadoop 文件系统的接口，例如 create open list 等方法。

图 6-7 FileContext 类

7.6.1 Hadoop 中的路径

Hadoop 中使用 Path 来表示一个文件路径。Path 中含有文件所在的 URI。

Hadoop 支持 URI 名字空间和 URI 名字。Hadoop 中 URI 名字非常灵活，使用者可以在不知道服务器地址或名字的情况下，访问默认的文件系统。默认的文件系统通过配置文件来制定。

Hadoop 中路径名有三种形式：

- 完全限定的 URI：scheme://authority/path
- 以斜杠开头的路径：/path 表示相对于默认的文件系统，即相当于 default-scheme:// default-authority/path
- 相对路径：path 相对于工作目录

完全限定的 URI 和以斜杠开头的路径称为绝对路径。

但如下形式的路径则是非法的：scheme:foo/bar

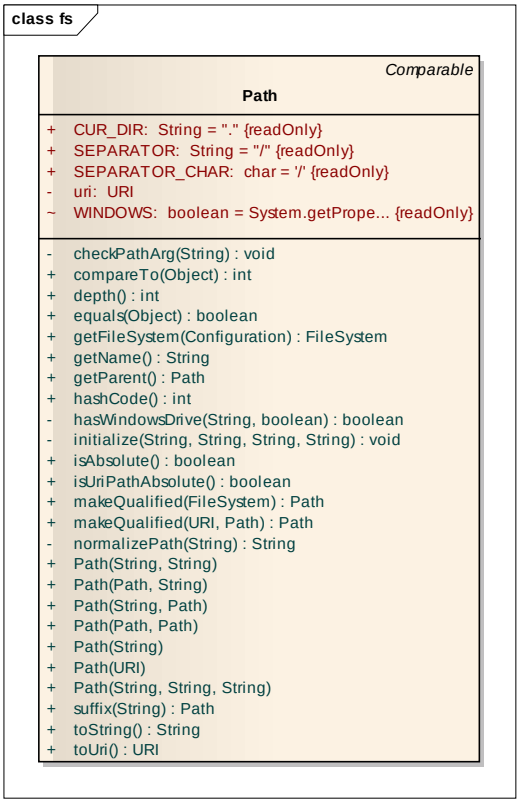

图 6-8 Path 类

Hadoop 中文件和目录的路径用 Path 类表示。Path 的表示与 Unix 路径相似，使用’/’来分隔目录，而不是’\’。

7.6.2 server side 属性

所有的文件系统实例（例如文件系统的部署）都有默认属性。这些属性叫做 server side (SS) defaults。像 create 这样的操作允许选择多种属性：要么当做参数传给它，要么使用 SS 属性。

SS 属性由 FsServerDefaults 表示。

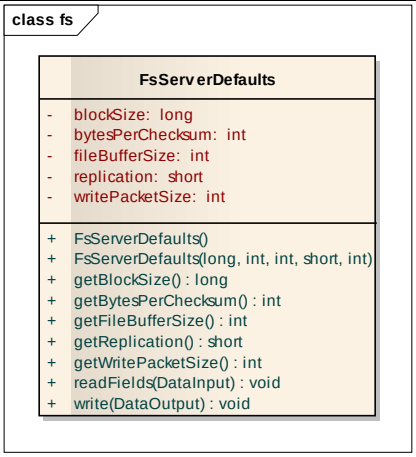

图 6-9 FsServerDefaults 类

与文件系统有关的 SS 属性有

- the home directory (default is "/user/userName")
- the initial wd (only for local fs)
- replication factor 分片因子
- block size
- buffer size
- bytesPerChecksum (if used).

7.6.3 FileContext

FileContext 由默认文件系统、工作目录和 umask 定义。

获得一个 FileContext 可以使用 FileContext 的静态方法：getFileContext();getLocalFSFileContext()

getFileContext 有多重重载的形式。但最终都是调用了如下函数：

```
/**
     * Create a FileContext for specified default URI using the specified
config.
     *
     * @param defaultFsUri
     * @param aConf
     * @return new FileContext for specified uri
     * @throws UnsupportedFileSystemException If the file system with
specified is
     *                 not supported
     */
   public static FileContext getFileContext(final URI defaultFsUri,
         final Configuration aConf) throws
UnsupportedFileSystemException {
      return getFileContext(AbstractFileSystem.get(defaultFsUri,    aConf),
aConf);
   }

/**
     * Create a FileContext with specified FS as default using the specified
     * config.
     *
     * @param defFS
     * @param aConf
     * @return new FileContext with specifed FS as default.
     */
   protected static FileContext getFileContext(final AbstractFileSystem
defFS,
                                final Configuration aConf) {
      return new FileContext(defFS, FsPermission.getUMask(aConf),
aConf);
   }
```

图 6-10 代码分析

public void setWorkingDirectory(final Path newWDir) throws IOException 用来设置工作目录。

FileContext 中的很多方法是用来取代 FileSystem 中的方法的。包括 create，mkdir，delete，open，setReplication，rename，setPermission，setOwner，setTimes，getFileStatus，

getFileLinkStatus，getLinkTarget，getFileBlockLocations 等方法在 AbstractFileSystem 中都有对应的方法。这些方法的实现具有相同的形式。比如 open 方法的代码如下：

```
public FSDataInputStream open(final Path f, final int bufferSize)
      throws AccessControlException, FileNotFoundException,
      UnsupportedFileSystemException, IOException {
    final Path absF = fixRelativePart(f); //  转换为绝对路径
    return new FSLinkResolver<FSDataInputStream>() {
      public FSDataInputStream next(final AbstractFileSystem fs, final
Path p)
          throws IOException, UnresolvedLinkException {
          return fs.open(p, bufferSize);
        }
    }.resolve(this, absF);//如无异常，则调用AbstractFileSystem.open方法
  }
```

图6-11 代码分析

内部 FSLinkResolver<T>抽象模板使用来解析路径中的符号链接的。public abstract T next(final AbstractFileSystem fs, final Path p)为其抽象方法。resolve 方法如下：

```
public T resolve(final FileContext fc, Path p) throws IOException {
    int count = 0;
    T in = null;
    Path first = p;
    // NB: More than one AbstractFileSystem can match a scheme, eg
    // "file" resolves to LocalFs but could have come by RawLocalFs.
    AbstractFileSystem fs = fc.getFSofPath(p);

    // Loop until all symlinks are resolved or the limit is reached
    for (boolean isLink = true; isLink;) {
      try {
        in = next(fs, p); //  调用next
        isLink = false;
      } catch (UnresolvedLinkException e) {
        if (count++ > MAX_PATH_LINKS) {
          throw new IOException("Possible cyclic loop while " +
                                "following symbolic link " + first);
        }
        // Resolve the first unresolved path component
        p = qualifySymlinkTarget(fs, p, fs.getLinkTarget(p));
        fs = fc.getFSofPath(p);
      }
    }
    return in;
  }
```

图6-12 代码分析

FileContext 的 getFSofPath 方法用来获得支持指定路径 path 的文件系统。该方法首先检查路径 path 是不是属于默认文件系统支持的路径，如果是则返回 FileContext.defaultFS 属性；否则 AbstractFileSystem.get 来获取对应的文件系统。

在获得文件系统 fs 之后，resolve 会调用 next 方法，并返回其结果。如果出现异常，则解析路径中的符号链接，再次调用 next。

总的来说，FileContext.open 是通过调用 AbstractFileSystem.open 来实现的。

8 渐寥糁

8.1 Hadoop She l l 哟但

调用文件系统(FS)Shell 命令应使用 bin/hadoop fs <args>的形式。 所有的的 FS shell 命令使用 URI 路径作为参数。URI 格式是 scheme://authority/path。对 HDFS 文件系统，scheme 是 hdfs，对本地文件系统，scheme 是 file。其中 scheme 和 authority 参数都是可选的，如果未加指定，就会使用配置中指定的默认 scheme。一个 HDFS 文件或目录比如/parent/child 可以表示成 hdfs://namenode:namenodeport/parent/child，或者更简单的/parent/child（假设你配置文件中的默认值是 namenode:namenodeport）。大多数 FS Shell 命令的行为和对应的 Unix Shell 命令类似，不同之处会在下面介绍各命令使用详情时指出。出错信息会输出到 stderr，其他信息输出到 stdout。

- cat

  使用方法：hadoop fs -cat URI [URI …]

  将路径指定文件的内容输出到 stdout。

  示例：

  hadoop fs -cat hdfs://host1:port1/file1 hdfs://host2:port2/file2

hadoop fs -cat file:///file3 /user/hadoop/file4

返回值：

成功返回 0，失败返回-1。

- chgrp

使用方法：hadoop fs -chgrp [-R] GROUP URI [URI …]

Change group association of files. With -R, make the change recursively through the directory structure. The user must be the owner of files, or else a super-user. Additional information is in the Permissions User Guide. -->

改变文件所属的组。使用-R 将使改变在目录结构下递归进行。命令的使用者必须是文件的所有者或者超级用户。更多的信息请参见 HDFS 权限用户指南。

- chmod

使用方法：hadoop fs -chmod [-R] <MODE[,MODE]... | OCTALMODE> URI [URI …]

改变文件的权限。使用-R 将使改变在目录结构下递归进行。命令的使用者必须是文件的所有者或者超级用户。更多的信息请参见 HDFS 权限用户指南。

- chown

使用方法：hadoop fs -chown [-R] [OWNER][:[GROUP]] URI [URI]

改变文件的拥有者。使用-R 将使改变在目录结构下递归进行。命令的使用者必须是超级用户。更多的信息请参见 HDFS 权限用户指南。

- copyFromLocal

使用方法：hadoop fs -copyFromLocal <localsrc> URI

除了限定源路径是一个本地文件外，和 **put** 命令相似。

- copyToLocal

使用方法：hadoop fs -copyToLocal [-ignorecrc] [-crc] URI <localdst>

除了限定目标路径是一个本地文件外，和 **get** 命令类似。

- cp

使用方法：hadoop fs -cp URI [URI …] <dest>

将文件从源路径复制到目标路径。这个命令允许有多个源路径，此时目标路径必须是一个目录。

示例：

hadoop fs -cp /user/hadoop/file1 /user/hadoop/file2

hadoop fs -cp /user/hadoop/file1 /user/hadoop/file2 /user/hadoop/dir

返回值：

成功返回 0，失败返回-1。

- du

使用方法：hadoop fs -du URI [URI …]

显示目录中所有文件的大小，或者当只指定一个文件时，显示此文件的大小。

示例：

hadoop fs -du /user/hadoop/dir1 /user/hadoop/file1 hdfs://host:port/user/hadoop/dir1

返回值：

成功返回 0，失败返回-1。

- dus

使用方法：hadoop fs -dus <args>

显示文件的大小。

- expunge

使用方法：hadoop fs -expunge

清空回收站。请参考 HDFS 设计文档以获取更多关于回收站特性的信息。

- get

使用方法：hadoop fs -get [-ignorecrc] [-crc] <src> <localdst>

复制文件到本地文件系统。可用-ignorecrc 选项复制 CRC 校验失败的文件。使用-crc 选项复制文件以及 CRC 信息。

示例：

hadoop fs -get /user/hadoop/file localfile

hadoop fs -get hdfs://host:port/user/hadoop/file localfile

返回值：

成功返回 0，失败返回-1。

- getmerge

使用方法：hadoop fs -getmerge <src> <localdst> [addnl]

接受一个源目录和一个目标文件作为输入，并且将源目录中所有的文件连接成本地目标文件。addnl 是可选的，用于指定在每个文件结尾添加一个换行符。

- ls

使用方法：hadoop fs -ls <args>

如果是文件，则按照如下格式返回文件信息：

文件名 <副本数> 文件大小 修改日期 修改时间 权限 用户 ID 组 ID

如果是目录，则返回它直接子文件的一个列表，就像在 Unix 中一样。目录返回列表的信息如下：

目录名 <dir> 修改日期 修改时间 权限 用户 ID 组 ID

示例：

hadoop fs -ls /user/hadoop/file1 /user/hadoop/file2 hdfs://host:port/user/hadoop/dir1 /nonexistentfile

返回值：

成功返回 0，失败返回-1。

- lsr

使用方法：hadoop fs -lsr <args>

ls 命令的递归版本。类似于 Unix 中的 ls -R。

- mkdir

使用方法：hadoop fs -mkdir <paths>

接受路径指定的 uri 作为参数，创建这些目录。其行为类似于 Unix 的 mkdir -p，它会创建路径中的各级父目录。

示例：

hadoop fs -mkdir /user/hadoop/dir1 /user/hadoop/dir2

hadoop fs -mkdir hdfs://host1:port1/user/hadoop/dir hdfs://host2:port2/user/hadoop/dir

返回值：

成功返回 0，失败返回-1。

- movefromLocal

使用方法：dfs -moveFromLocal <src> <dst>

输出一个”not implemented“信息。

- mv

使用方法：hadoop fs -mv URI [URI …] <dest>

将文件从源路径移动到目标路径。这个命令允许有多个源路径，此时目标路径必须是一个目录。不允许在不同的文件系统间移动文件。

示例：

hadoop fs -mv /user/hadoop/file1 /user/hadoop/file2

hadoop fs -mv hdfs://host:port/file1 hdfs://host:port/file2 hdfs://host:port/file3 hdfs://host:port/dir1

返回值：

成功返回 0，失败返回-1。

- put

使用方法：hadoop fs -put <localsrc> ... <dst>

从本地文件系统中复制单个或多个源路径到目标文件系统。也支持从标准输入中读取输入写入目标文件系统。

hadoop fs -put localfile /user/hadoop/hadoopfile

hadoop fs -put localfile1 localfile2 /user/hadoop/hadoopdir

hadoop fs -put localfile hdfs://host:port/hadoop/hadoopfile

hadoop fs -put - hdfs://host:port/hadoop/hadoopfile

从标准输入中读取输入。

返回值：

成功返回 0，失败返回-1。

- rm

使用方法：hadoop fs -rm URI [URI …]

删除指定的文件。只删除非空目录和文件。请参考 rmr 命令了解递归删除。

示例：

hadoop fs -rm hdfs://host:port/file /user/hadoop/emptydir

返回值：

成功返回 0，失败返回-1。

- rmr

使用方法：hadoop fs -rmr URI [URI …]

delete 的递归版本。

示例：

hadoop fs -rmr /user/hadoop/dir

hadoop fs -rmr hdfs://host:port/user/hadoop/dir

返回值：

成功返回 0，失败返回-1。

- setrep

使用方法：hadoop fs -setrep [-R] <path>

改变一个文件的副本系数。-R 选项用于递归改变目录下所有文件的副本系数。

示例：

hadoop fs -setrep -w 3 -R /user/hadoop/dir1

返回值：

成功返回 0，失败返回-1。

- stat

使用方法：hadoop fs -stat URI [URI …]

返回指定路径的统计信息。

示例：

hadoop fs -stat path

返回值：

成功返回 0，失败返回-1。

- tail

使用方法：hadoop fs -tail [-f] URI

将文件尾部 1K 字节的内容输出到 stdout。支持-f 选项，行为和 Unix 中一致。

示例：

hadoop fs -tail pathname

返回值：

成功返回 0，失败返回-1。

- test

使用方法：hadoop fs -test -[ezd] URI

选项：

-e 检查文件是否存在。如果存在则返回 0。

-z 检查文件是否是 0 字节。如果是则返回 0。

-d 如果路径是个目录，则返回 1，否则返回 0。

示例：

hadoop fs -test -e filename

text

使用方法：hadoop fs -text <src>

将源文件输出为文本格式。允许的格式是 zip 和 TextRecordInputStream。

- touchz

使用方法：hadoop fs -touchz URI [URI …]

创建一个 0 字节的空文件。

示例：

hadoop -touchz pathname

返回值：

成功返回 0，失败返回-1。

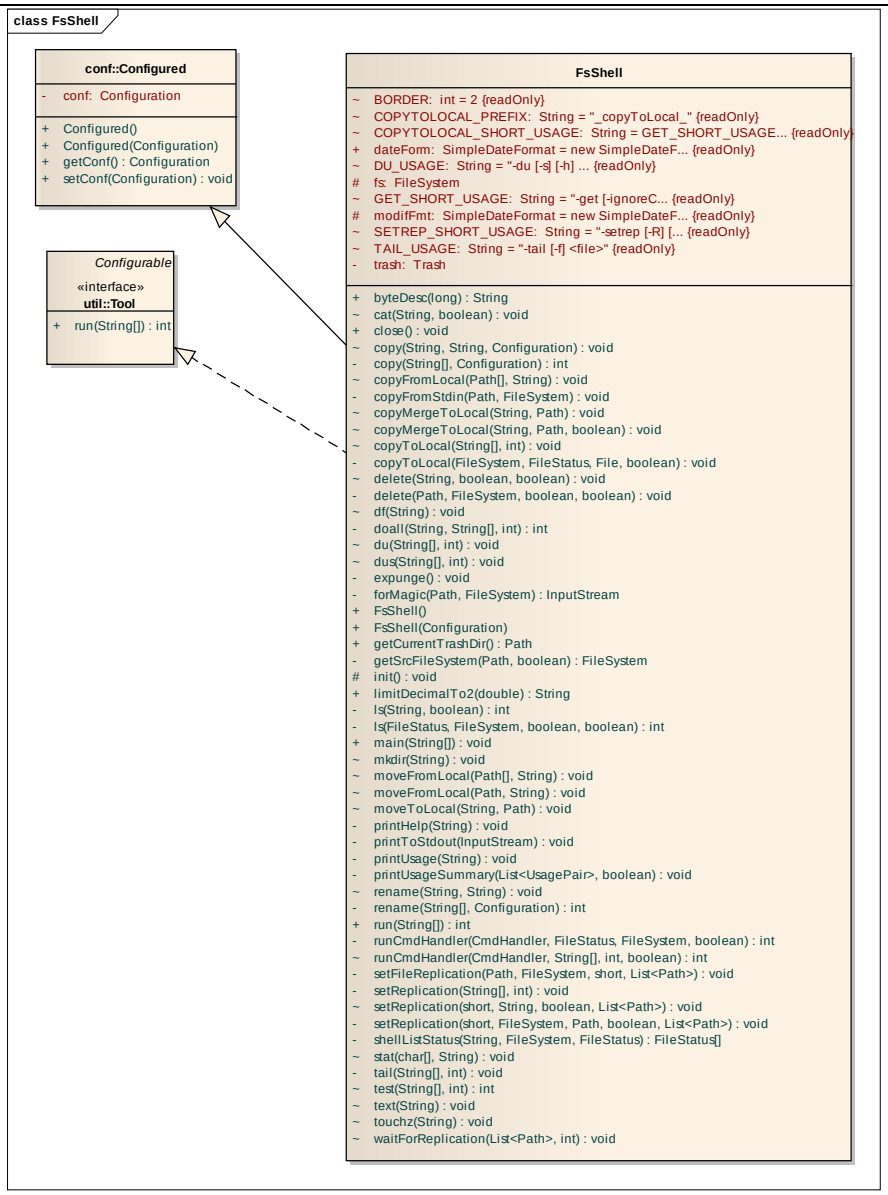

图 7-1 FsShell 类

FsShell 提供了通过命令行来访问文件系统的能力。大部分的 shell 命令是在 FsShell 中实现的。

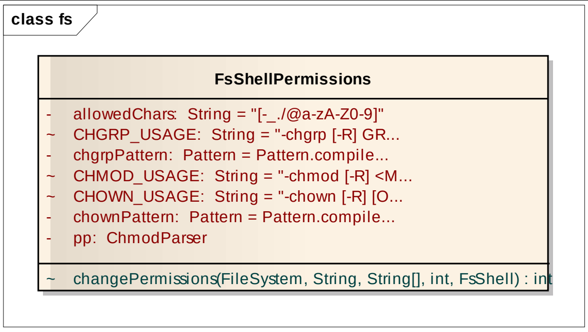

图 8-2 FsShellpPermissions 类

FsShellpPermissions 是从 FsShell 中分离出来的，实现了 chmod、chown 和 chgrp 三个命令。它有三个内部类，从 FsShell::CmdHandler 派生，具体实现了运行 Shell 命令。如下图所示。

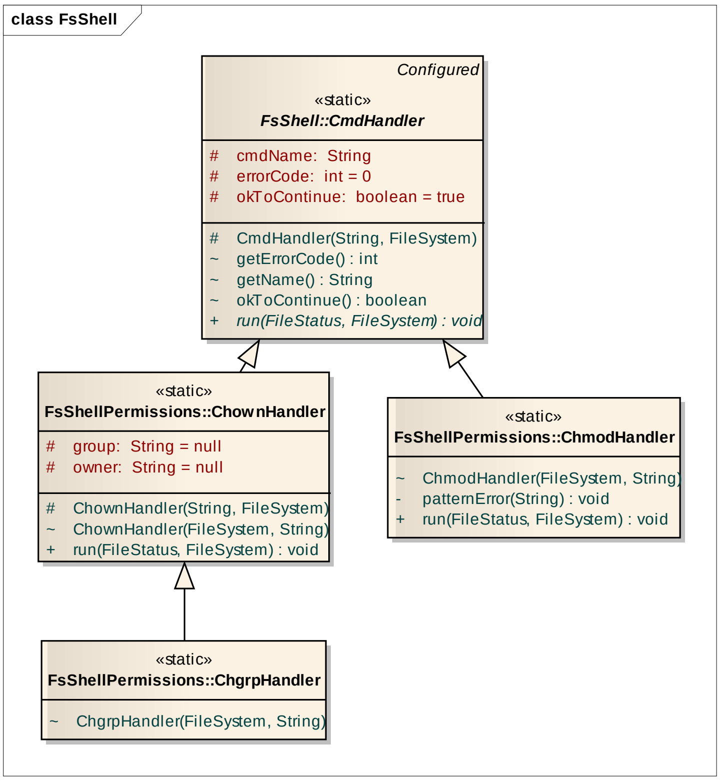

图 8-3 FsShell::CmdHandler 的派生类

这三个内部类通过重写 run 方法，可以在单独的线程内运行。

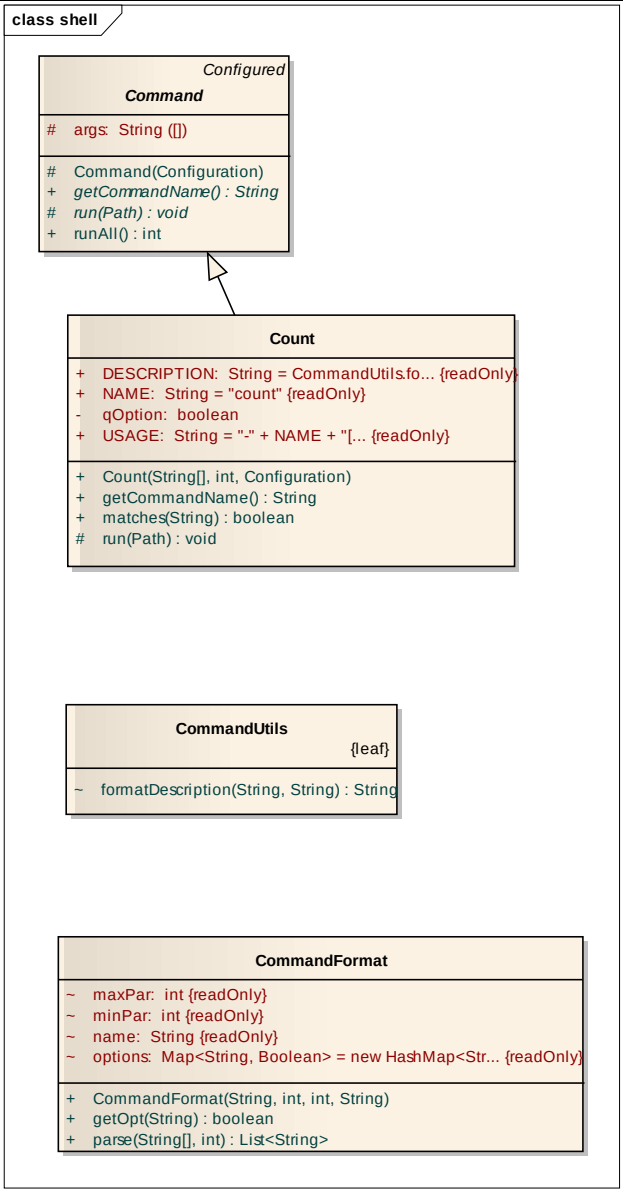

图 7-4 shell 包

这是 shell 子包的类图。Command 代表一个要执行的系统命令。它有一个 run 抽象方法，具体的命令要从该类派生，并重写 run 方法。Count 类用来统计文件夹个数、文件个数和文件的字节数等。

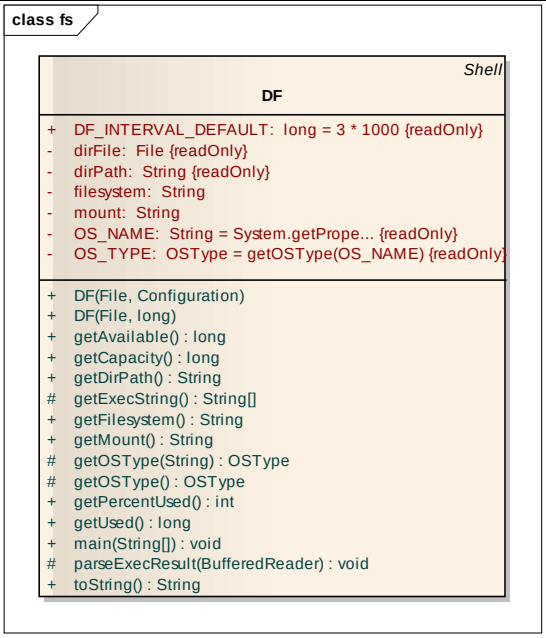

图 7-5 DF 类

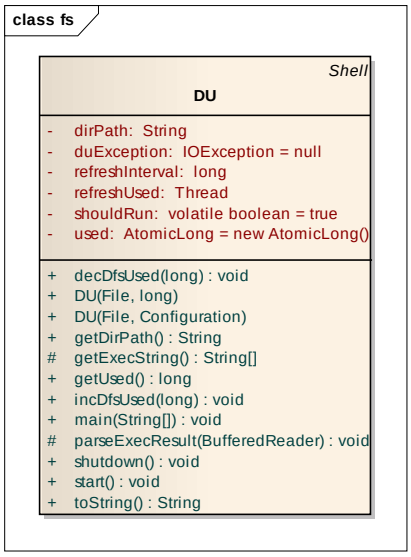

图 7-6 DU 类

DF 和 DU 通过 shell 来调用 Unix 命令 df 和 du。DF 和 DU 都有 main 方法，可以生成独立的 Java 程序。

8.2 栻糝

8.2.1 BlockLocation

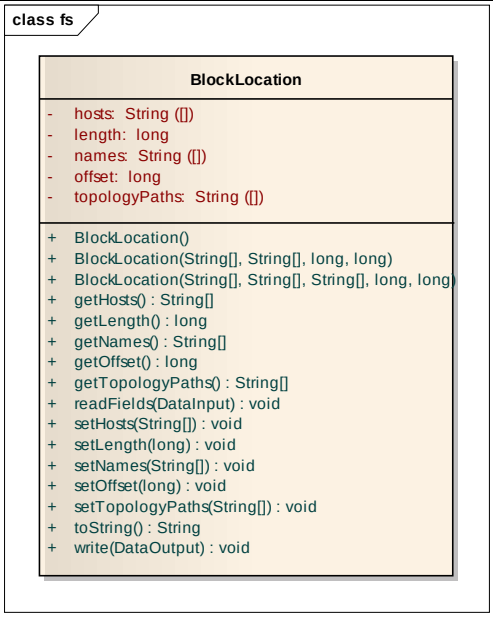

图 7-7 BlockLocation 类

BlockLocation 类记录了一个 Block 的位置信息，包括：该块所在的数据节点的主机名，端口号，拓扑路径，该块在文件中的偏移量和块的长度。

8.2.2 异常与错误

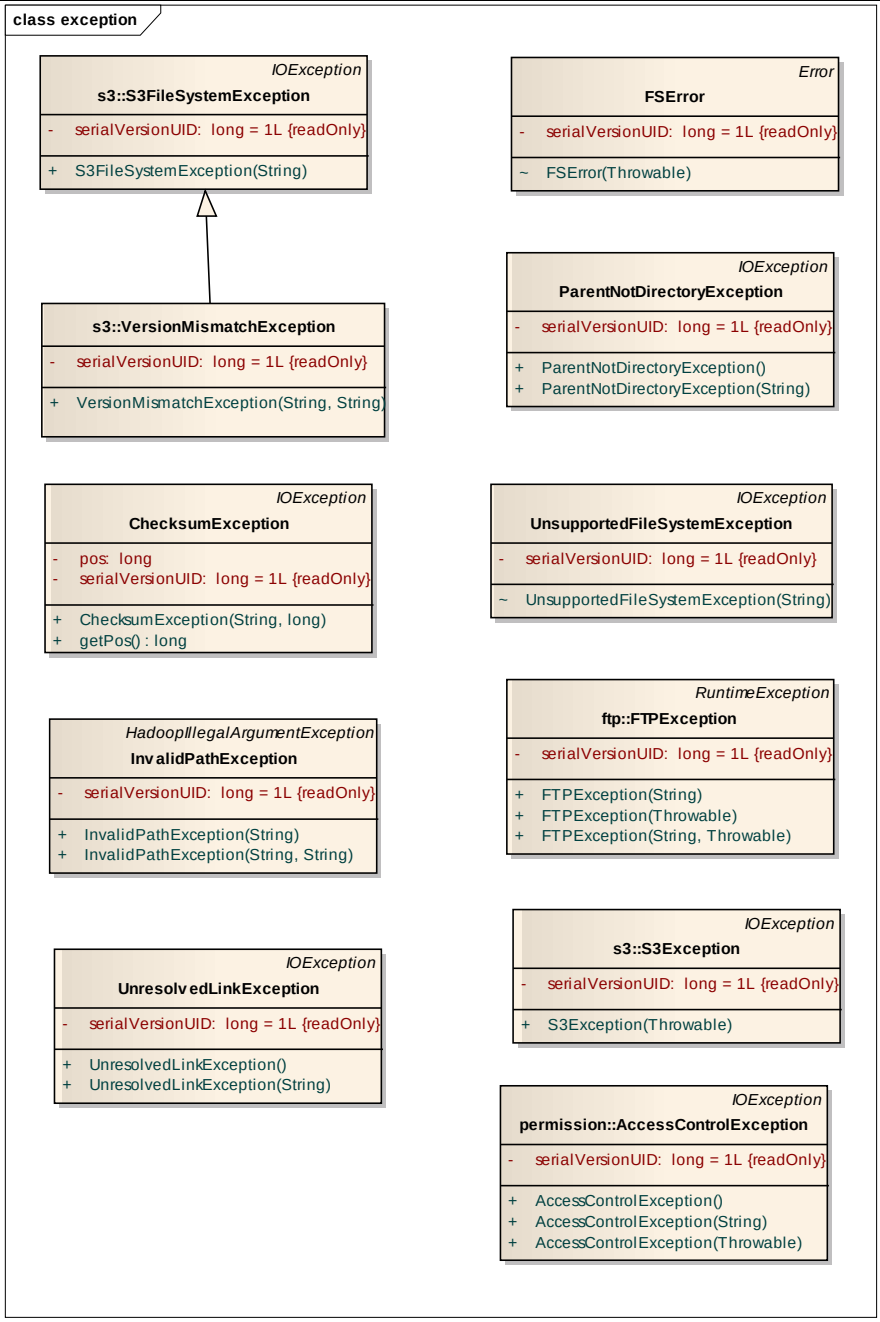

图 7-8 异常与错误

8.2.3 配置

图 8-9 FsConfig 类

FsConfig 类提供了从一些静态方法，用来方便从配置文件中获取或设置以下几项的值：

fs.default.name

fs.homeDir

dfs.replication

dfs.block.size

io.file.buffer.size

class config

CommonConfigurationKeys

+ FS_AUTOMATIC_CLOSE_DEFAULT: boolean = true {readOnly}
+ FS_AUTOMATIC_CLOSE_KEY: String = "fs.automatic.close" {readOnly}
+ FS_CLIENT_BUFFER_DIR_KEY: String = "fs.client.buff... {readOnly}
+ FS_DEFAULT_NAME_DEFAULT: String = "file:///" {readOnly}
+ FS_DEFAULT_NAME_KEY: String = "fs.defaultFS" {readOnly}
+ FS_DF_INTERVAL_DEFAULT: long = 60000 {readOnly}
+ FS_DF_INTERVAL_KEY: String = "fs.df.interval" {readOnly}
+ FS_FILE_IMPL_KEY: String = "fs.file.impl" {readOnly}
+ FS_FTP_HOST_KEY: String = "fs.ftp.host" {readOnly}
+ FS_FTP_HOST_PORT_KEY: String = "fs.ftp.host.port" {readOnly}
+ FS_HOME_DIR_DEFAULT: String = "/user" {readOnly}
+ FS_HOME_DIR_KEY: String = "fs.homeDir" {readOnly}
+ FS_LOCAL_BLOCK_SIZE_DEFAULT: long = 32*1024*1024 {readOnly}
+ FS_PERMISSIONS_UMASK_DEFAULT: int = 0022 {readOnly}
+ FS_PERMISSIONS_UMASK_KEY: String = "fs.permissions... {readOnly}
+ FS_TRASH_INTERVAL_DEFAULT: long = 0 {readOnly}
+ FS_TRASH_INTERVAL_KEY: String = "fs.trash.interval" {readOnly}
+ HADOOP_JOB_UGI_KEY: String = "hadoop.job.ugi" {readOnly}
+ HADOOP_RPC_SOCKET_FACTORY_CLASS_DEFAULT_KEY: String = "hadoop.rpc.soc... {readOnly}
+ HADOOP_SECURITY_AUTHENTICATION: String = "hadoop.securit... {readOnly}
+ HADOOP_SECURITY_AUTHORIZATION: String = "hadoop.securit... {readOnly}
+ HADOOP_SECURITY_GROUP_MAPPING: String = "hadoop.securit... {readOnly}
+ HADOOP_SECURITY_GROUPS_CACHE_SECS: String = "hadoop.securit... {readOnly}
+ HADOOP_SOCKS_SERVER_KEY: String = "hadoop.socks.s... {readOnly}
+ HADOOP_UTIL_HASH_TYPE_DEFAULT: String = "murmur" {readOnly}
+ HADOOP_UTIL_HASH_TYPE_KEY: String = "hadoop.util.ha... {readOnly}
+ IO_COMPRESSION_CODEC_LZO_BUFFERSIZE_DEFAULT: int = 64*1024 {readOnly}
+ IO_COMPRESSION_CODEC_LZO_BUFFERSIZE_KEY: String = "io.compression... {readOnly}
+ IO_COMPRESSION_CODEC_LZO_CLASS_KEY: String = "io.compression... {readOnly}
+ IO_MAP_INDEX_INTERVAL_DEFAULT: int = 128 {readOnly}
+ IO_MAP_INDEX_INTERVAL_KEY: String = "io.map.index.i... {readOnly}
+ IO_MAP_INDEX_SKIP_DEFAULT: int = 0 {readOnly}
+ IO_MAP_INDEX_SKIP_KEY: String = "io.map.index.skip" {readOnly}
+ IO_MAPFILE_BLOOM_ERROR_RATE_DEFAULT: float = 0.005f {readOnly}
+ IO_MAPFILE_BLOOM_ERROR_RATE_KEY: String = "io.mapfile.blo... {readOnly}
+ IO_MAPFILE_BLOOM_SIZE_DEFAULT: int = 1024*1024 {readOnly}
+ IO_MAPFILE_BLOOM_SIZE_KEY: String = "io.mapfile.blo... {readOnly}
+ IO_NATIVE_LIB_AVAILABLE_DEFAULT: boolean = true {readOnly}
+ IO_NATIVE_LIB_AVAILABLE_KEY: String = "io.native.lib.... {readOnly}
+ IO_SEQFILE_COMPRESS_BLOCKSIZE_DEFAULT: int = 1000000 {readOnly}
+ IO_SEQFILE_COMPRESS_BLOCKSIZE_KEY: String = "io.seqfile.com... {readOnly}
+ IO_SERIALIZATIONS_KEY: String = "io.serializations" {readOnly}
+ IO_SKIP_CHECKSUM_ERRORS_DEFAULT: boolean = false {readOnly}
+ IO_SKIP_CHECKSUM_ERRORS_KEY: String = "io.skip.checks... {readOnly}
+ IO_SORT_FACTOR_DEFAULT: int = 100 {readOnly}
+ IO_SORT_FACTOR_KEY: String = "io.sort.factor" {readOnly}
+ IO_SORT_MB_DEFAULT: int = 100 {readOnly}
+ IO_SORT_MB_KEY: String = "io.sort.mb" {readOnly}
+ IPC_CLIENT_CONNECT_MAX_RETRIES_DEFAULT: int = 10 {readOnly}
+ IPC_CLIENT_CONNECT_MAX_RETRIES_KEY: String = "ipc.client.con... {readOnly}
+ IPC_CLIENT_CONNECTION_MAXIDLETIME_DEFAULT: int = 10000 {readOnly}
+ IPC_CLIENT_CONNECTION_MAXIDLETIME_KEY: String = "ipc.client.con... {readOnly}
+ IPC_CLIENT_IDLETHRESHOLD_DEFAULT: int = 4000 {readOnly}
+ IPC_CLIENT_IDLETHRESHOLD_KEY: String = "ipc.client.idl... {readOnly}
+ IPC_CLIENT_KILL_MAX_DEFAULT: int = 10 {readOnly}
+ IPC_CLIENT_KILL_MAX_KEY: String = "ipc.client.kil... {readOnly}
+ IPC_CLIENT_PING_DEFAULT: boolean = true {readOnly}
+ IPC_CLIENT_PING_KEY: String = "ipc.client.ping" {readOnly}
+ IPC_CLIENT_TCPNODELAY_DEFAULT: boolean = false {readOnly}
+ IPC_CLIENT_TCPNODELAY_KEY: String = "ipc.client.tcp... {readOnly}
+ IPC_PING_INTERVAL_DEFAULT: int = 60000 {readOnly}
+ IPC_PING_INTERVAL_KEY: String = "ipc.ping.interval" {readOnly}
+ IPC_SERVER_HANDLER_QUEUE_SIZE_DEFAULT: int = 100 {readOnly}
+ IPC_SERVER_HANDLER_QUEUE_SIZE_KEY: String = "ipc.server.han... {readOnly}
+ IPC_SERVER_LISTEN_QUEUE_SIZE_DEFAULT: int = 128 {readOnly}
+ IPC_SERVER_LISTEN_QUEUE_SIZE_KEY: String = "ipc.server.lis... {readOnly}
+ IPC_SERVER_RPC_MAX_RESPONSE_SIZE_DEFAULT: int = 1024*1024 {readOnly}
+ IPC_SERVER_RPC_MAX_RESPONSE_SIZE_KEY: String = "ipc.server.max... {readOnly}
+ IPC_SERVER_RPC_READ_THREADS_DEFAULT: int = 1 {readOnly}
+ IPC_SERVER_RPC_READ_THREADS_KEY: String = "ipc.server.rea... {readOnly}
+ IPC_SERVER_TCPNODELAY_DEFAULT: boolean = false {readOnly}
+ IPC_SERVER_TCPNODELAY_KEY: String = "ipc.server.tcp... {readOnly}
+ NET_TOPOLOGY_CONFIGURED_NODE_MAPPING_KEY: String = "net.topology.c... {readOnly}
+ NET_TOPOLOGY_NODE_SWITCH_MAPPING_IMPL_KEY: String = "net.topology.n... {readOnly}
+ NET_TOPOLOGY_SCRIPT_FILE_NAME_KEY: String = "net.topology.s... {readOnly}
+ NET_TOPOLOGY_SCRIPT_NUMBER_ARGS_DEFAULT: int = 100 {readOnly}
+ NET_TOPOLOGY_SCRIPT_NUMBER_ARGS_KEY: String = "net.topology.s... {readOnly}
+ TFILE_FS_INPUT_BUFFER_SIZE_DEFAULT: int = 256*1024 {readOnly}
+ TFILE_FS_INPUT_BUFFER_SIZE_KEY: String = "tfile.fs.input... {readOnly}
+ TFILE_FS_OUTPUT_BUFFER_SIZE_DEFAULT: int = 256*1024 {readOnly}
+ TFILE_FS_OUTPUT_BUFFER_SIZE_KEY: String = "tfile.fs.outpu... {readOnly}
+ TFILE_IO_CHUNK_SIZE_DEFAULT: int = 1024*1024 {readOnly}
+ TFILE_IO_CHUNK_SIZE_KEY: String = "tfile.io.chunk... {readOnly}

图 7-10 CommonConfigurationKeys 类

8.2.4 ContentSummary

图 7-11 ContentSummary 类

ContentSummary 类存储有关文件或目录的一些信息，包括文件长度，文件数量，目录数量，磁盘配额，已用空间大小，剩余空间大小。

8.2.5 CreateFlag

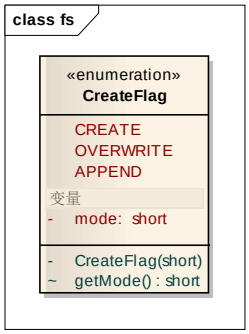

图 7-12 CreateFlag 类

CreateFlag 枚举用来指定在创建文件时的标志，如 FileSystem #create(Path f, FsPermission permission, * EnumSet flag, int bufferSize, short replication, long blockSize, Progressable progress)。常量 CREATE、OVERWRITE 和 APPEND 可以通过或运算相互组合。

8.2.6 FileChecksum

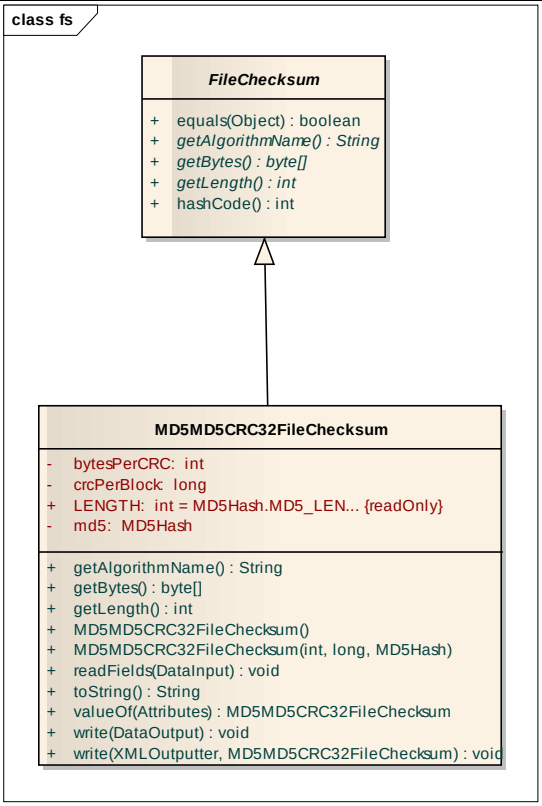

图 7-13 FileChecksum 类

FileChecksum 是一抽象类，从 Writable 接口派生，来表示一些文件的校验和。MD5MD5CRC32FileChecksum 类从 FileChecksum，实现了序列化和反序列化的功能。

8.2.7 FileUtil

图 7-14 FileUtil 类

FileUtil 类提供了一些有关文件处理的实用方法，包括文件夹的递归删除，跨文件系统的文件的复制（使用 IOUtils.copyBytes()方法实现）取得当前 shell 的路径，解压文件，创建硬链接和符号链接，chmod，创建临时文件和替换文件等方法。

8.2.8 FsStatus

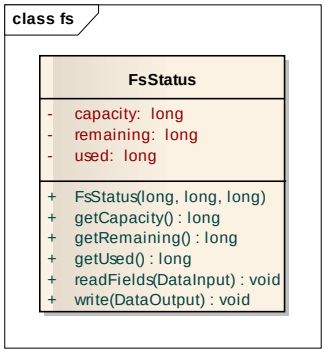

图 7-15 FsStatus 类

FsStatus 类表示了一个文件系统的容量、已使用空间大小和剩余空间大小。

8.2.9 FsUrlConnection

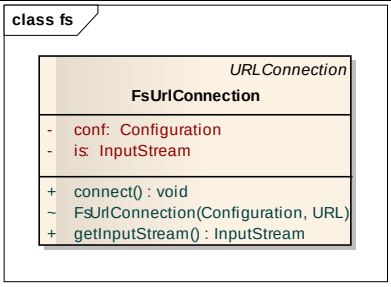

图 7-16 FsUrlConnection 类

FsUrlConnection 代表一个 URL 连接，用来打开一个文件系统的输入流。

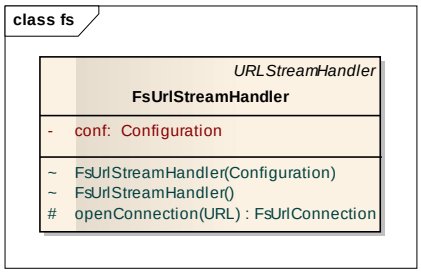

图 7-17 FsUrlStreamHandler 类

FsUrlStreamHandler 用来处理文件系统有关的 URL 协议，实现了 openConnection 抽象方法，该方法会创建一个 FsUrlConnection 实例。

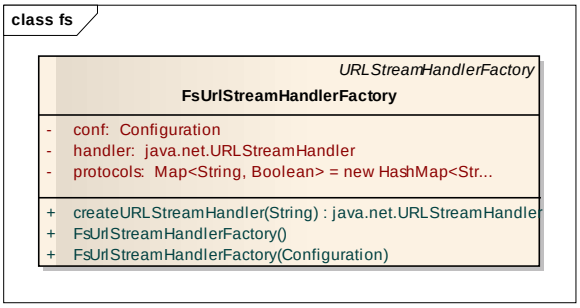

图 7-18 FsUrlStreamHandlerFactory 类

FsUrlStreamHandlerFactory 工厂类会根据 URL 协议创建相应的 FsUrlStreamHandler 类的实例。

8.2.10 GlobExpander

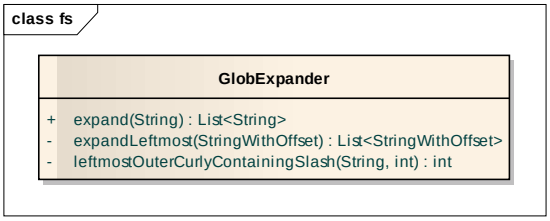

图 7-19 GlobExpander 类

GlobExpander 用来对文件通配符中的模式进行转换。

8.2.11 LocalDirAllocator

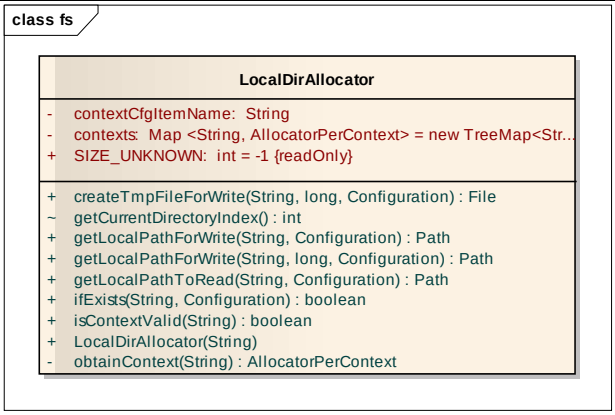

图 7-20 LocalDirAllocator 类

对磁盘使用 round-robin 算法来分配文件。

8.2.12 Options

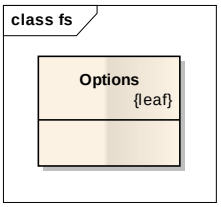

图 7-21 Options 类

Options 类对 FileSystem.create 方法中的可变参数提供支持。

8.2.13 Trash

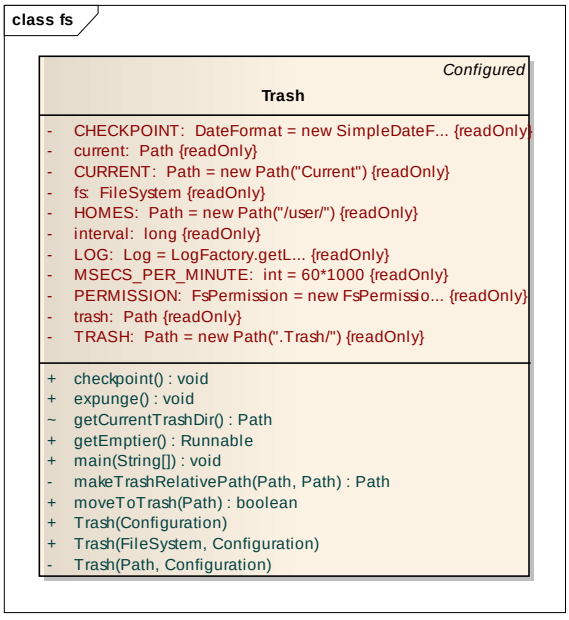

图 7-22 Trash 类

该类提供了垃圾箱的功能。垃圾箱所在的文件夹是用户主目录的一个子目录.Trash。方法 moveToTrash 可以将文件移到到垃圾箱中。

9 缵谜买递屺毅盍帮俘

对报告解决的问题做一总结与讨论，并说明下一步的主要工作。