HadoopのファイルシステムAPI入門

2019/08/24
日本Hadoopユーザー会
岩崎 正剛

はじめに

Hadoopが提供するFileSystem APIを解説

　　JavaのAPIの話

普段ユーザがあまり意識しない部分かも

　　MapReduceやSpark経由で利用されている

より深くHadoopを使いこなすために

HDFS

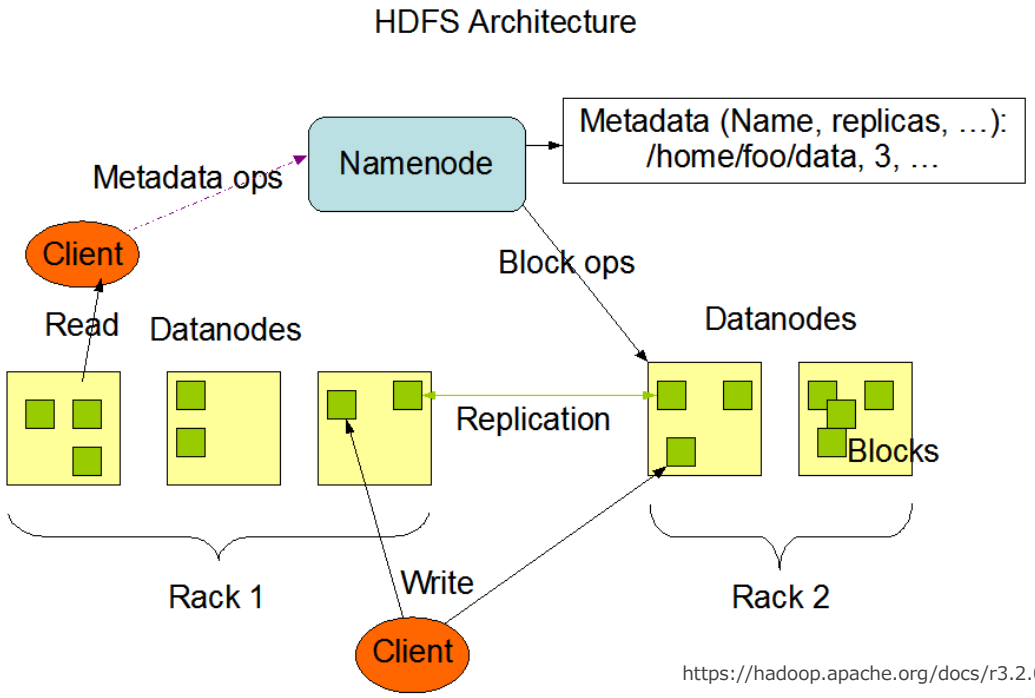
HDFS

Hadoop Distributed File System

Hadoop = 分散FS + 分散処理FW

Hadoopアプリケーション =

(HDFS上の)データを分散処理するもの



https://hadoop.apache.org/docs/r3.2.0/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

HDFSの概要

- ファイルシステムとしての機能を提供
  - 階層的な名前空間(ファイルとディレクトリ)
  - ファイルデータの高速な読み書き
  - パーミッションによるアクセス制御
  - quota
  - 透過的暗号化
  - extended file attribute, inotify
- xfsやext4などの上で動く
- POSIX準拠ではない
- 可用性、データ保全性が高い
- 大きなファイル(100+MB)の格納に最適化

HDFSのスケーラビリティ

マスターノード(NameNode)がボトルネック

1. NameNodeのヒープサイズ(<100GBくらい?)
2. 管理可能なスレーブノード数(<10000くらい?)
3. 処理可能なリクエスト数(<10万tpsくらい?)

ざっくりした目安

100万データブロックあたりヒープ1GB
ヒープサイズはGC的に100GB程度まで
1億ブロックで12.8PB (ブロックサイズ128MBで)

基本的なファイル操作

CLIによる基本的なファイル操作

Linuxのコマンドと似たような雰囲気

CLI(FsShell)はJava APIを利用して作られたもの

```
$ hdfs dfs -mkdir -p /foo/bar
$ hdfs dfs -chmod g+w /foo/bar
$ hdfs dfs -ls -R /
drwxr-xr-x - iwasakims supergroup 0 2019-08-21 15:11 /foo
drwxrwxr-x - iwasakims supergroup 0 2019-08-21 15:11 /foo/bar

$ echo baz > baz.txt
$ hdfs dfs -put baz.txt /foo/bar
$ hdfs dfs -head /foo/bar/baz.txt
baz

$ hdfs dfs -rm -r /foo
```

FileSystemインスタンスの取得

URIに対応するインスタンスを取得
設定上のデフォルトFSなら明示的な指定は不要

```
scala> import org.apache.hadoop.conf.Configuration
scala> import org.apache.hadoop.fs.FileSystem
scala> import org.apache.hadoop.fs.Path

scala> val conf = new Configuration()
scala> conf.get("fs.defaultFS")
res0: String = hdfs://localhost:8020/
scala> val fs = FileSystem.get(conf)

scala> val path = new Path("hdfs://localhost:8020/")
scala> val fs = p.getFileSystem(conf)
scala> val fs = FileSystem.get(path.toUri(), conf)
```

mkdirs

ディレクトリの作成
基本的に親がなければ作成 (mkdir -p)

```
scala> val path = new Path("/foo/bar")

scala> fs.mkdirs(path)
res1: Boolean = true

scala> fs.exists(new Path("/foo"))
res2: Boolean = true
```

setPermission

パーミッションの設定
値の指定は8進数やenumで
HDFSの場合ファイルのx(execute)に意味はない

```
scala> import org.apache.hadoop.fs.permission.FsPermission
scala> import org.apache.hadoop.fs.permission.FsAction

scala> val perm = new FsPermission("0775")
scala> fs.setPermission(path, perm)

scala> val perm = new FsPermission(FsAction.ALL,
                                   FsAction.ALL,
                                   FsAction.READ_EXECUTE)
perm: org.apache.hadoop.fs.permission.FsPermission = rwxrwxr-x
scala> fs.setPermission(path, perm)
```

listStatus

ファイル情報(FileStatus)の取得

```
scala> val listing = fs.listStatus(new Path("/"))
listing: Array[org.apache.hadoop.fs.FileStatus] =
Array(HdfsLocatedFileStatus{path=hdfs://localhost:8020/foo;
isDirectory=true; modification_time=1566384749729; access_time=0;
owner=iwasakims; group=supergroup; permission=rwxr-xr-x; isSymlink=false;
hasAcl=false; isEncrypted=false; isErasureCoded=false})

scala> listing(0).isDirectory
res1: Boolean = true

scala> fs.listStatus(listing(0).getPath())
res4: Array[org.apache.hadoop.fs.FileStatus] =
Array(HdfsLocatedFileStatus{path=hdfs://localhost:8020/foo/bar;
isDirectory=true; modification_time=1566384749729; access_time=0;
owner=iwasakims; group=supergroup; permission=rwxrwxr-x; isSymlink=false;
hasAcl=false; isEncrypted=false; isErasureCoded=false})
```

create

ファイルの新規作成&書き込みオープン
得られたOutputStreamにバイト列を書き込む
seek不可
create時点で他のクライアントにもvisible

```
scala> import java.nio.charset.Charset

scala> val os = fs.create(new Path("/foo/bar/baz.txt"))

scala> val buf ="baz".getBytes(Charset.forName("UTF-8"))
buf: Array[Byte] = Array(98, 97, 122)

scala> os.write(buf, 0, buf.length)

scala> os.close()
```

open

ファイルの読み込みオープン
任意の位置をreadできる(pread)

```
scala> import java.nio.ByteBuffer

scala> val is = fs.open(new Path("/foo/bar/baz.txt"))

scala> val buf = ByteBuffer.allocate(3)
scala> is.read(buf)
res1: Int = 3
scala> new String(buf.array(), Charset.forName("UTF-8"))
res2: String = baz

scala> val buf = ByteBuffer.allocate(2)
scala> is.read(1, buf)
res3: Int = 2
scala> new String(buf.array(), Charset.forName("UTF-8"))
res4: String = az
```

delete

ファイル/ディレクトリの削除
再帰的に削除するかどうかを引数で指定
引数なしのdelete(rm -r)はdeprecated

```
scala> fs.delete(new Path("/foo"), true)
res3: Boolean = true

scala> fs.delete(new Path("/foo"))
warning: there was one deprecation warning; for details, enable
`:setting -deprecation' or `:replay -deprecation'
res23: Boolean = false
```

HDFS特有の機能/仕様

データローカリティ

ブロックを保持するノード情報の取得
分散処理のタスクスケジューリングで利用
　ブロック保持ノードに処理させる

```
scala> val it = fs.listLocatedStatus(path)
it:
org.apache.hadoop.fs.RemoteIterator[org.apache.hadoop.fs.LocatedFi
leStatus] = org.apache.hadoop.hdfs.DistributedFileSystem
$DirListingIterator@68f68a1a

scala> while (it.hasNext()) {
     |   val locations = it.next().getBlockLocations()
     |   locations.foreach(println)
     | }
0,6,localhost
```

append

ファイルを書き込み再オープン
書き込みは末尾に追加(既存部分は上書きできない)

```
scala> val os = fs.append(new Path("/foo/bar/baz.txt"))
os: org.apache.hadoop.fs.FSDataOutputStream =
FSDataOutputStream{wrappedStream=DFSOutputStream:blk_1073741828_1004}

scala> val buf ="bazbaz".getBytes(Charset.forName("UTF-8"))

scala> os.write(buf, 0, buf.length)

scala> os.close()
```

```
$ bin/hdfs dfs -cat /foo/bar/baz.txt
bazbazbaz
```

hflush / hsync

書き込みを確定する(fsync)
ファイルシステムメタデータは更新されない
hsync: 各スレーブノードでfsyncしてリターン

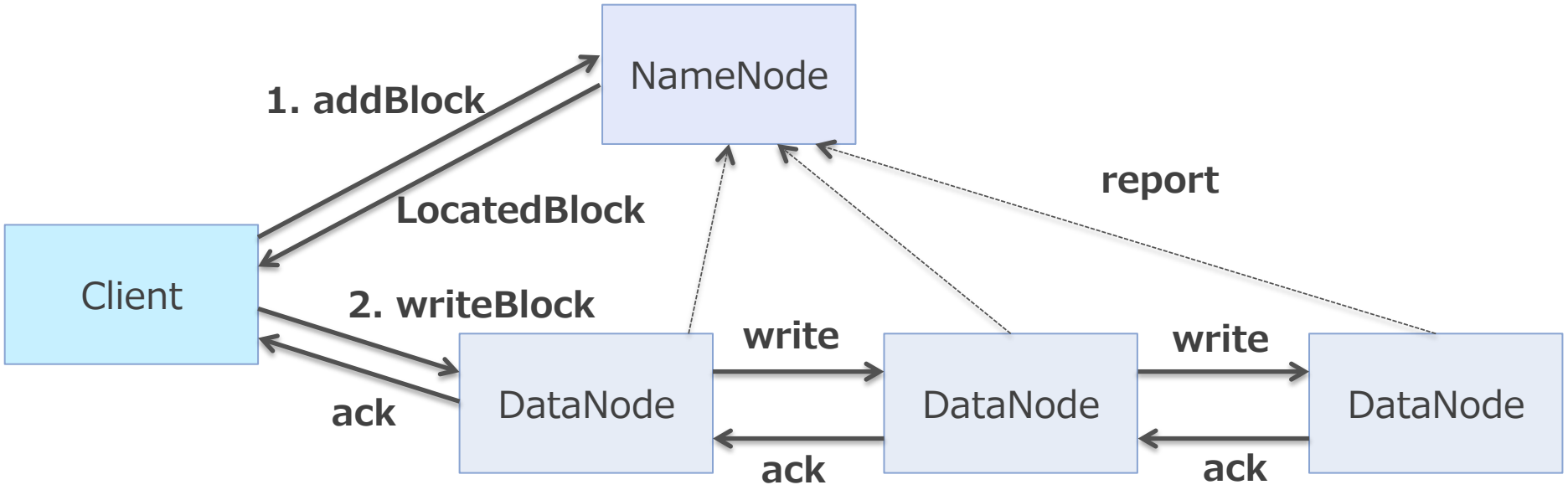
```
scala> val os = fs.create(new Path("/hflush.txt"))
scala> val buf ="sync".getBytes(Charset.forName("UTF-8"))
scala> os.write(buf, 0, buf.length)
scala> os.hflush()
scala> os.write(buf, 0, buf.length)
scala> os.hflush()
```

```
$ hdfs dfs -cat /hflush.txt
syncsync

$ hdfs dfs -ls /hflush.txt
-rw-r--r-- 1 iwasakims supergroup 4 2019-08-22 09:47 /hflush.txt
```

(参考)HDFSのデータ書き込みの流れ

NameNodeにブロック割り当てリクエスト
DataNodeに対して書き込みパイプライン構築
DataNodeからNameNodeにブロック情報を報告



Tailing

open時点でvisibleな部分までしか読めない

tail -f的に読むには「再openしてseek」を繰り返す

```
scala> val is = fs.open(new Path("/foo/bar/baz.txt"))
scala> is.read()
res1: Int = 98
...
scala> is.read()
res2: Int = -1
scala> val os = fs.append(new Path("/foo/bar/baz.txt"))
scala> os.write(buf, 0, buf.length)
scala> os.close()
scala> is.read()
res3: Int = -1

scala> val is = fs.open(new Path("/foo/bar/baz.txt"))
scala> fs.seek(3)
scala> is.read()
res4: Int = 98
```

セキュリティ

Hadoopのユーザ認証

Kerberosを利用

JAAS(Java Authentication and Authorization Service)

FileSystem APIで意識する場面は少ない

スレーブノード上で実行されるタスクは
エンドユーザ権限でHDFSにアクセスする必要がある

Delegation Token

ジョブ投入時にトークンを取得
タスクはトークンを利用して認証

```
scala> val ugi = UserGroupInformation.getCurrentUser()
ugi: org.apache.hadoop.security.UserGroupInformation =
iwasakims@EXAMPLE.COM (auth:KERBEROS)

scala> val token =
  fs.getDelegationToken("yarn/localhost@EXAMPLE.COM")
token: org.apache.hadoop.security.token.Token[_] = Kind: HDFS_DELEGATION_TOKEN,
Service: 127.0.0.1:8020, Ident: (token for iwasakims: HDFS_DELEGATION_TOKEN
owner=iwasakims@EXAMPLE.COM, renewer=iwasakims, realUser=, issueDate=1566394690379,
maxDate=1566999490379, sequenceNumber=1, masterKeyId=4)

scala> ugi.addToken(token)
res2: Boolean = true
```

Delegation Tokenの受け渡し

スレーブノード上のタスクにtokenを渡す

```
scala> import org.apache.hadoop.security.Credentials
scala> import org.apache.hadoop.io.DataOutputBuffer
scala> import org.apache.hadoop.io.DataInputByteBuffer
scala> import org.apache.hadoop.security.Credentials

scala> val creds = ugi.getCredentials()
scala> val ob = new DataOutputBuffer()
scala> creds.writeTokenStorageToStream(ob)
scala> val buf = ByteBuffer.wrap(ob.getData(), 0, ob.getLength())
```

```
scala> val creds = new Credentials()
scala> val ib = new DataInputByteBuffer()
scala> ib.reset(buf)
scala> creds.readTokenStorageStream(ib)
scala> val ugi = UserGroupInformation.getLoginUser()
scala> ugi.addCredentials(creds)
```

分散処理のための部品

Hadoopジョブによるデータ処理の流れ

入力ファイルを分割してタスクに対応づける
タスクごとにデータを処理する
    作業用ディレクトリを作る
    タスクの出力ファイルを作る
    入力ファイルからレコードを読み出す
    データを処理する
    出力ファイルにレコードを書き込む
出力ファイルを作業場所から移動する

入出力はフレームワークで抽象化されている

InputFormat

入力(ファイル)を抽象化するもの

- 入力をInputSplitに分割
- レコードを読み出す
- 例えばテキストファイルならレコードは行
- see TextOutputFormat

```
public abstract class InputFormat<K, V> {
  public abstract
    List<InputSplit> getSplits(JobContext context
                               ) throws ...

  public abstract
    RecordReader<K,V> createRecordReader(InputSplit split,
                                         TaskAttemptContext context
                                        ) throws ...
}
```

OutputFormat

出力(ファイル)を抽象化するもの

- レコードを書き出す
- 出力できるかを確認する
- 出力を確定する

```
public abstract class OutputFormat<K, V> {
  public abstract RecordWriter<K, V>
    getRecordWriter(TaskAttemptContext context
                    ) throws ...

  public abstract void checkOutputSpecs(JobContext context
                                        ) throws ...
  public abstract
  OutputCommitter getOutputCommitter(TaskAttemptContext context
                                     ) throws ...
}
```

OutputCommitter

ジョブ/タスク完了時に出力を確定

成功: ユーザ/後続処理にすべての出力が見える

失敗: ユーザ/後続処理にゴミが一切見えない

ジョブ/タスクの失敗/中止の後片付け

```
public abstract class OutputCommitter {
  public abstract void setupJob(JobContext jobContext)
  public void cleanupJob(JobContext jobContext)
  public void commitJob(JobContext jobContext)
  public void abortJob(JobContext jobContext, JobStatus.State state)
  public abstract void setupTask(TaskAttemptContext taskContext)
  public abstract boolean needsTaskCommit(TaskAttemptContext taskContext)
  public abstract void commitTask(TaskAttemptContext taskContext)
  public abstract void abortTask(TaskAttemptContext taskContext)
  public boolean isRecoverySupported()
  public boolean isCommitJobRepeatable(JobContext jobContext)
  public boolean isRecoverySupported(JobContext jobContext)
  public void recoverTask(TaskAttemptContext taskContext)
}
```

FileOutputCommitter

デフォルトのOutputCommitter

成功:

　出力ファイルを最終出力先にrename

　_SUCCESSというからファイルを作る

失敗:

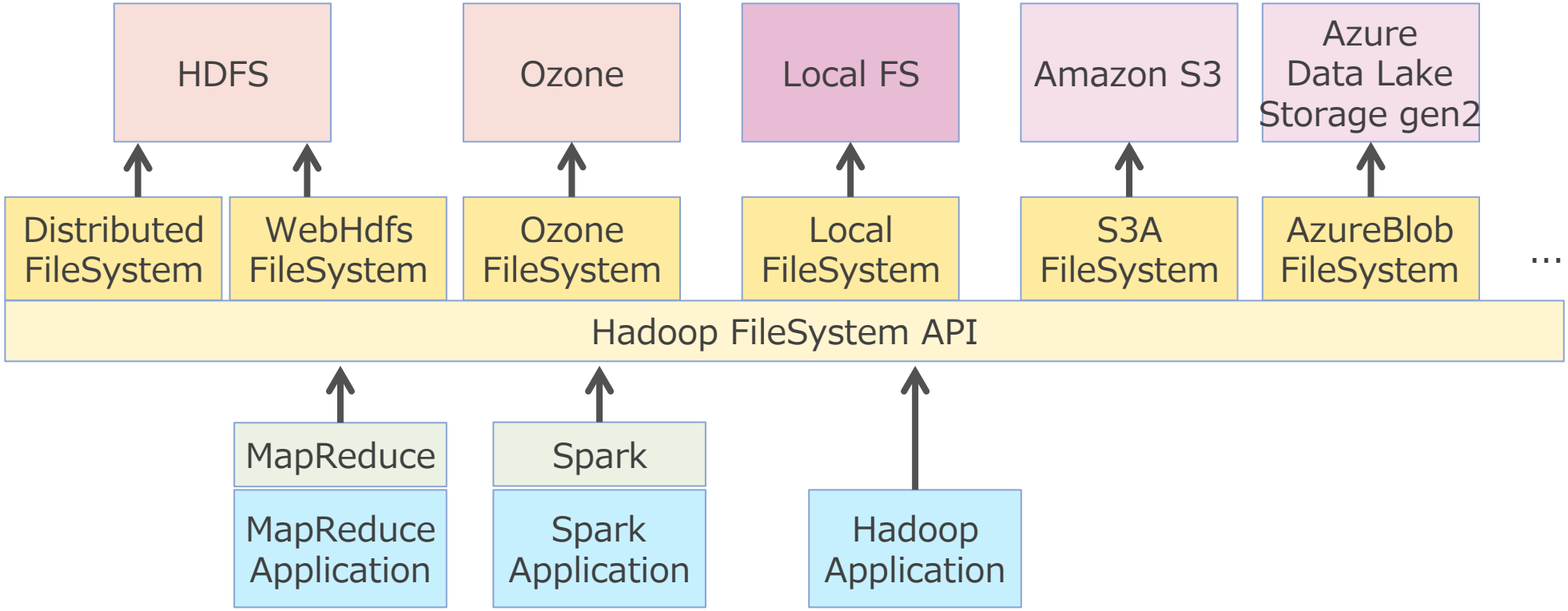
　作業ディレクトリをdelete

Why it matters

基本的なユースケースではあまり意識しなくて済む
既存の実装が要件にマッチしなければ改造できる
FileSystem APIの使い方の参考にもなる

データストアの抽象化

Hadoop Compatible File Systems



異なるデータストアへのアクセス

ファイルのpathをURI形式で指定
schemeに応じてよしなにデータを読み書き
裏でFileSystem実装をロードして使い分け

```
$ hadoop fs -cp file:///a/b hdfs://ns/c/d
$ hadoop fs -cp hdfs://ns/c/d s3a://bc/e/f
```

Amazon S3へのアクセス

オブジェクトをファイル風に読み書き

```
scala> val p = new Path("s3a://iwasakims-test/foo.txt")
scala> val s3 = p.getFileSystem(conf)
s3: org.apache.hadoop.fs.FileSystem = S3AFileSystem{uri=s3a://
iwasakims-test, workingDir=s3a://iwasakims-test/user/iwasakims,
inputPolicy=normal, ...

scala> val os = s3.create(p)
os: org.apache.hadoop.fs.FSDataOutputStream =
FSDataOutputStream{wrappedStream=S3ABlockOutputStream{WriteOperati
onHelper {bucket=iwasakims-test}, blockSize=67108864,
activeBlock=FileBlock{index=1, destFile=/tmp/hadoop-iwasakims/s3a/
s3ablock-0001-6278414896901011411.tmp, state=Writing, dataSize=0,
limit=67108864}}}

scala> val buf ="foo".getBytes(Charset.forName("UTF-8"))
scala> os.write(buf, 0, buf.length)
scala> os.close()
```

Limitations

対応していない(できない)機能もある
例外 or 何も起きない(noop)

```
scala> val os = s3.append(p)
java.lang.UnsupportedOperationException: Append is not supported
by S3AFileSystem
  ...

scala> s3.setPermission(p, new FsPermission("0775"))


scala> val is = s3.open(p)
scala> val buf = ByteBuffer.allocate(3)
scala> is.read(buf)
java.lang.UnsupportedOperationException: Byte-buffer read
unsupported by input stream
  ...
```

サードパーティ製のFileSystem実装

Google Cloud Storage

https://github.com/GoogleCloudPlatform/bigdata-interop/tree/master/gcs

Oracle Cloud Infrastructure

https://github.com/oracle/oci-hdfs-connector

Ignite File System

https://github.com/apache/ignite/tree/master/modules/hadoop

FileSystem実装の追加

.jarにclasspathを通す

Configurationでschemeとクラス名を対応づけ

```
<property>
  <name>fs.foobar.impl</name>
  <value>org.example.FooBarFileSystem</value>
</property>
```

もしくはjava.util.ServiceLoaderを使う

使ってなくてもロードされるのが難点

```
$ tail META-INF/services/org.apache.hadoop.fs.FileSystem
org.example.FooBarFileSystem
```

fs.defaultFS

デフォルト(pathのみ指定)時にどれを使うかは
設定ファイル(core-site.xml)上の指定で決まる

```
<property>
  <name>fs.defaultFS</name>
  <value>hdfs://mycluster/</value>
</property>
```

FileContext API (HADOOP-4952)

FileSystem APIをユーザ向けに整理する意図

意図通りに普及/移行していない...

- FileContextのドキュメントがない
- できることに(ほとんど)差はない
- Hadoopのコード自体が両方使っている

FileSystem実装を作るときにケアする必要あり

```
FileContext ctx = FileContext.getFileContext(uri);
FSDataInputStream is = ctx.create(path, ...);
ctx.setWorkingDir(path);
FSDataInputStream os = ctx.open(path, ...);
```

AbstractFileSystem?

FileContext APIのためのもの
ユーザに見せないバックエンド部分
FileSystemのベースクラスではない
FileSystem実装をwrapするパターンが多い
see o.a.h.fs.DelegateToFileSystem

```
<property>
  <name>fs.s3a.impl</name>
  <value>org.apache.hadoop.fs.s3a.S3AFileSystem</value>
</property>
<property>
  <name>fs.AbstractFileSystem.s3a.impl</name>
  <value>org.apache.hadoop.fs.s3a.S3A</value>
</property>
```

まとめ

まとめ

FileSystem APIでデータを読み書きできる
FileSystem APIはHDFSの機能を抽象化したもの
HDFS以外のデータストアにも透過的にアクセス
自分で実装を作ることもできる

おわり