Prof R. Madana Mohana

BIG DATA ANALYTICS

HDFS

(Hadoop Distributed Files System)

The Java Interface

https://www.youtube.com/c/RASINENIMADANAMOHANA

HDFS: Outline

The Java Interface :

- Reading Data from a Hadoop URL
- Reading Data Using the File System API

The Java Interface :

- Writing Data
- Directories

The Java Interface:

- Querying the File system
- Deleting Data

The Java Interface

HDFS: The Java Interface

The Java Interface :

- Reading Data from a Hadoop URL
- Reading Data Using the File System API
- Writing Data
- Directories
- Querying the File system
- Deleting Data

HDFS: The Java Interface

- **Hadoop FileSystem class:** the **API** for interacting with one of Hadoop's filesystems.
- **HDFS** implementation, DistributedFileSystem, in general we should strive to write our code against the **FileSystem** abstract class, to retain portability across filesystems.

The Java Interface: Reading Data from a Hadoop URL

- One of the simplest ways to read a file from a **Hadoop** filesystem is by using a **java.net.URL** object to open a stream to read the data from.
- The general idiom is:

```
InputStream in = null;
try {
in = new URL("hdfs://host/path").openStream();
// process in
} finally {
IOUtils.closeStream(in);
}
```

The Java Interface: Reading Data from a Hadoop URL

- There's a little bit more work required to make **Java** recognize **Hadoop's hdfs URL** scheme.
- This is achieved by calling the **setURLStreamHandlerFactory()** method on **URL** with an instance of **FsUrlStreamHandlerFactory**.
- This method can be called only once per **JVM**, so it is typically executed in a static block.
- This limitation means that if some other part of your program-perhaps a third-party component outside your control-sets a **URLStreamHandlerFactory**, you won't be able to use this approach for reading data from **Hadoop**.

The Java Interface: Reading Data from a Hadoop URL

Example: *A program for Displaying files from a Hadoop filesystem on standard output using a* ***URLStreamHandler***

```
public class URLCat {
static {
URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
}
public static void main(String[] args) throws Exception {
InputStream in = null;
try {
in = new URL(args[0]).openStream();
IOUtils.copyBytes(in, System.out, 4096, false);
} finally {
IOUtils.closeStream(in);
} } }
```

The Java Interface: Reading Data from a Hadoop URL

Example: *A program for Displaying files from a Hadoop filesystem on standard output using a* ***URLStreamHandler***

- We make use of the handy **IOUtils** class that comes with **Hadoop** for closing the stream in the **finally** clause, and also for copying bytes between the input stream and the output stream (**System.out**, in this case).
- The last two arguments to the **copyBytes()** method are the buffer size used for copying and whether to close the streams when the copy is complete.
- We close the input stream ourselves, and **System.out** doesn't need to be closed.

The Java Interface: Reading Data from a Hadoop URL

Example: *A program for Displaying files from a Hadoop filesystem on standard output using a* ***URLStreamHandler***

Sample Output:

% export HADOOP_CLASSPATH=hadoop-examples.jar

% hadoop URLCat hdfs://localhost/user/tom/quangle.txt

On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.

The Java Interface: Reading Data Using the File System API

- Sometimes it is impossible to set a **URLStreamHandlerFactory** for our application. In this case, we will need to use the **FileSystem API** to open an input stream for a file.
- A file in a **Hadoop filesystem** is represented by a **Hadoop Path object** (and not a **java.io.File** object, since its semantics are too closely tied to the local filesystem).

The Java Interface: Reading Data Using the File System API

- We can think of a **Path** as a **Hadoop filesystem URI**, such as **hdfs://localhost/user/tom/quangle.txt.**
- **FileSystem** is a general filesystem API, so the first step is to retrieve an instance for the filesystem we want to use-**HDFS**, in this case.

The Java Interface: Reading Data Using the File System API

There are several static factory methods for getting a ***FileSystem*** *instance:*

```
public static FileSystem get(Configuration conf)
throws IOException

public static FileSystem get(URI uri, Configuration
conf) throws IOException

public static FileSystem get(URI uri, Configuration
conf, String user) throws IOException
```

The Java Interface: Reading Data Using the File System API

- A **Configuration** object encapsulates a client or server's configuration, which is set using configuration files read from the classpath, such as ***etc/hadoop/core-site.xml***.
- The ***first method*** returns the default filesystem (as specified in ***core-site.xml***, or the default local filesystem if not specified there).
- The ***second*** uses the given **URI's** scheme and authority to determine the filesystem to use, falling back to the default filesystem if no scheme is specified in the given **URI**.
- The ***third*** retrieves the filesystem as the given user, which is important in the context of security.

The Java Interface: Reading Data Using the File System API

- In some cases, we may want to retrieve a local filesystem instance. For this, we can use the convenience method **getLocal()**:

```
public static LocalFileSystem getLocal(Configuration conf)
throws IOException
```

- With a **FileSystem** instance in hand, we invoke an **open()** method to get the input stream for a file:

```
public FSDataInputStream open(Path f) throws IOException
public abstract FSDataInputStream open(Path f, int
bufferSize) throws IOException
```

- The **first method** uses a default buffer size of **4 KB**.

The Java Interface: Reading Data Using the File System API

Example-1. *Displaying files from a Hadoop filesystem on standard output by using the FileSystem directly*

```
public class FileSystemCat {
public static void main(String[] args) throws Exception {
String uri = args[0];
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(URI.create(uri), conf);
InputStream in = null;
try {
in = fs.open(new Path(uri));
IOUtils.copyBytes(in, System.out, 4096, false);
} finally {
IOUtils.closeStream(in);
} } }
```

The Java Interface: Reading Data Using the File System API

Example-1. *Displaying files from a Hadoop filesystem on standard output by using the FileSystem directly*

The program runs as follows:

```
% hadoop FileSystemCat hdfs://localhost/user/tom/quangle.txt
```

On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.

The Java Interface: Reading Data Using the File System API

FSDataInputStream:

- The **open()** method on **FileSystem** actually returns an **FSDataInputStream** rather than a standard **java.io class**.
- This class is a specialization of **java.io.DataInputStream** with support for random access, so we can read from any part of the stream:

```
package org.apache.hadoop.fs;
public class FSDataInputStream extends DataInputStream
implements Seekable, PositionedReadable {
// implementation elided (omitted)
}
```

The Java Interface: Reading Data Using the File System API

FSDataInputStream:

- The **Seekable** interface permits seeking to a position in the file and provides a query method for the current offset from the start of the file (**getPos()**):

```
public interface Seekable {
void seek(long pos) throws IOException;
long getPos() throws IOException;
}
```

The Java Interface: Reading Data Using the File System API

FSDataInputStream:

- Calling **seek()** with a position that is greater than the length of the file will result in an **IOException**.
- Unlike the **skip()** method of **java.io.InputStream**, which positions the stream at a point later than the current position, **seek()** can move to an arbitrary, absolute position in the file.

The Java Interface: Reading Data Using the File System API

FSDataInputStream:

Example-2. *Displaying files from a Hadoop filesystem on standard output twice, by using seek()*

```
public class FileSystemDoubleCat {
public static void main(String[] args) throws Exception {
String uri = args[0];
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(URI.create(uri), conf);
FSDataInputStream in = null;
```

The Java Interface: Reading Data Using the File System API

FSDataInputStream:

Example-2. *Displaying files from a Hadoop filesystem on standard output twice, by using seek()*

```
try {
in = fs.open(new Path(uri));
IOUtils.copyBytes(in, System.out, 4096, false);
in.seek(0); // go back to the start of the file
IOUtils.copyBytes(in, System.out, 4096, false);
} finally {
IOUtils.closeStream(in);
} } }
```

The Java Interface: Reading Data Using the File System API

FSDataInputStream:

Example-2. *Displaying files from a Hadoop filesystem on standard output twice, by using seek()*

Here's the result of running it on a small file:

```
% hadoop FileSystemDoubleCat hdfs://localhost/user/tom/quangle.txt
```

On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.

The Java Interface: Reading Data Using the File System API

FSDataInputStream:

FSDataInputStream also implements the **PositionedReadable** interface for reading parts of a file at a given offset:

```
public interface PositionedReadable {
public int read(long position, byte[] buffer, int
offset, int length) throws IOException;
public void readFully(long position, byte[] buffer,
int offset, int length) throws IOException;
public void readFully(long position, byte[] buffer)
throws IOException;
}
```

The Java Interface: Reading Data Using the File System API

FSDataInputStream:

- The **read()** method reads up to length bytes from the given position in the file into the buffer at the given offset in the buffer.
- The return value is the number of bytes actually read; callers should check this value, as it may be less than length.
- The **readFully()** methods will read length bytes into the buffer (or **buffer.length** bytes for the version that just takes a byte array buffer), unless the end of the file is reached, in which case an **EOFException** is thrown.

The Java Interface: Reading Data Using the File System API

FSDataInputStream:

- All of these methods preserve the current offset in the file and are thread safe (although **FSDataInputStream** is not designed for concurrent access; therefore, it's better to create multiple instances), so they provide a convenient way to access another part of the file-metadata, perhaps-while reading the main body of the file.
- Finally, calling **seek()** is a relatively expensive operation and should be done sparingly. You should structure your application access patterns to rely on streaming data (by using **MapReduce**, for example) rather than performing a large number of seeks.

The Java Interface: Writing Data

- The **FileSystem** class has a number of methods for creating a file.
- The simplest is the method that takes a Path object for the file to be created and returns an output stream to **write** to:

```
public FSDataOutputStream create(Path f) throws IOException
```

- There are overloaded versions of this method that allow you to specify whether to forcibly overwrite existing files, the replication factor of the file, the buffer size to use when writing the file, the block size for the file, and file permissions.

The Java Interface: Writing Data

- The **create()** methods create any parent directories of the file to be written that don't already exist.
- Though convenient, this behavior may be unexpected.
- If you want the write to fail when the parent directory doesn't exist, you should check for the existence of the parent directory first by calling the **exists()** method.
- Alternatively, use **FileContext**, which allows you to control whether parent directories are created or not.

The Java Interface: Writing Data

- There's also an overloaded method for passing a callback interface, **Progressable**, so our application can be notified of the progress of the data being written to the datanodes:

```
package org.apache.hadoop.util;
public interface Progressable {
  public void progress();
}
```

The Java Interface: Writing Data

- As an alternative to creating a new file, we can append to an existing file using the **append()** method (there are also some other overloaded versions):

```
public FSDataOutputStream append(Path f) throws IOException
```

- The **append** operation allows a single writer to modify an already written file by opening it and writing data from the final offset in the file.
- With this **API**, applications that produce unbounded files, such as logfiles, can write to an existing file after having closed it.

The Java Interface: Writing Data

- The **append** operation is optional and not implemented by all **Hadoop filesystems**.
- For example, **HDFS** supports **append**, but **S3** filesystems don't.

The Java Interface: Writing Data

Example. ***Copying a local file to a Hadoop filesystem***

```
public class FileCopyWithProgress {
  public static void main(String[] args) throws
Exception{
    String localSrc = args[0];
    String dst = args[1];

    InputStream in = new BufferedInputStream(new
FileInputStream(localSrc));
```

The Java Interface: Writing Data

Example. ***Copying a local file to a Hadoop filesystem***

```
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(dst), conf);
    OutputStream out = fs.create(new Path(dst), new
Progressable() {
        public void progress() {
            System.out.print(".");
        }
});
IOUtils.copyBytes(in, out, 4096, true);
}}
```

The Java Interface: Writing Data

Example. ***Copying a local file to a Hadoop filesystem***

Typical usage:

```
% hadoop FileCopyWithProgress input/docs/1400-8.txt
hdfs://localhost/user/tom/1400-8.txt

.....................
```

- Currently, none of the other **Hadoop filesystems** call **progress()** during writes.
- **Progress** is important in **MapReduce** applications.

The Java Interface: Writing Data

FSDataOutputStream:

The **create()** method on **FileSystem** returns an **FSDataOutputStream**, which, like **FSDataInputStream**, has a method for querying the current position in the file:

```
package org.apache.hadoop.fs;
public class FSDataOutputStream extends
DataOutputStream implements Syncable {
  public long getPos() throws IOException {
    // implementation elided
  } // implementation elided
}
```

The Java Interface: Writing Data

FSDataOutputStream:

- However, unlike **FSDataInputStream**, **FSDataOutputStream** does not permit seeking.
- This is because **HDFS** allows only sequential writes to an open file or appends to an already written file.
- In other words, there is no support for writing to anywhere other than the end of the file, so there is no value in being able to seek while writing.

The Java Interface: Directories

- **FileSystem** provides a method to create a directory:

```
public boolean mkdirs(Path f) throws IOException
```

- This method creates all of the necessary parent directories if they don't already exist, just like the **java.io.File's mkdirs()** method. It returns **true** if the directory (and all parent directories) was (were) successfully created.
- Often, you don't need to explicitly create a directory, because writing a file by calling **create()** will automatically create any parent directories.

The Java Interface: Querying the Filesystem

File metadata: FileStatus

- An important feature of any filesystem is the ability to navigate its directory structure and retrieve information about the files and directories that it stores.
- The **FileStatus** class encapsulates filesystem metadata for files and directories, including file length, block size, replication, modification time, ownership, and permission information.
- The method **getFileStatus()** on **FileSystem** provides a way of getting a **FileStatus** object for a single file or directory.

The Java Interface: Querying the Filesystem

File metadata: FileStatus

Example. ***Demonstrating file status information***

```
public class ShowFileStatusTest {
  private MiniDFSCluster cluster; // use an in-
process HDFS cluster for testing
  private FileSystem fs;
```

The Java Interface: Querying the Filesystem

File metadata: FileStatus

Example. ***Demonstrating file status information***

```
@Before
public void setUp() throws IOException {
   Configuration conf = new Configuration();
   if (System.getProperty("test.build.data") ==
null) {
      System.setProperty("test.build.data", "/tmp");
   }
```

The Java Interface: Querying the Filesystem

File metadata: FileStatus

Example. ***Demonstrating file status information***

```
 cluster = new
 MiniDFSCluster.Builder(conf).build();
 fs = cluster.getFileSystem();
 OutputStream out = fs.create(new
 Path("/dir/file"));
 out.write("content".getBytes("UTF-8"));
 out.close();
}
```

The Java Interface: Querying the Filesystem

File metadata: FileStatus

Example. ***Demonstrating file status information***

```
@After
public void tearDown() throws IOException {
   if (fs != null) { fs.close(); }
   if (cluster != null) { cluster.shutdown(); }
}
```

The Java Interface: Querying the Filesystem

File metadata: FileStatus

Example. ***Demonstrating file status information***

```
@Test(expected = FileNotFoundException.class)
public void throwsFileNotFoundForNonExistentFile()
throws IOException {
  fs.getFileStatus(new Path("no-such-file"));
}
```

The Java Interface: Querying the Filesystem

File metadata: FileStatus

Example. ***Demonstrating file status information***

```
@Test
public void fileStatusForFile() throws IOException {
Path file = new Path("/dir/file");
FileStatus stat = fs.getFileStatus(file);
assertThat(stat.getPath().toUri().getPath(),
is("/dir/file"));
assertThat(stat.isDirectory(), is(false));
assertThat(stat.getLen(), is(7L));
```

The Java Interface: Querying the Filesystem

File metadata: FileStatus

Example. ***Demonstrating file status information***

```
assertThat(stat.getModificationTime(),
is(lessThanOrEqualTo(System.currentTimeMillis())));
assertThat(stat.getReplication(), is((short) 1));
assertThat(stat.getBlockSize(), is(128 * 1024 * 1024L));
assertThat(stat.getOwner(),
is(System.getProperty("user.name")));
assertThat(stat.getGroup(), is("supergroup"));
assertThat(stat.getPermission().toString(),
is("rw-r--r--"));}
```

The Java Interface: Querying the Filesystem

File metadata: FileStatus

Example. ***Demonstrating file status information***

```
@Test
public void fileStatusForDirectory() throws
IOException {
   Path dir = new Path("/dir");
   FileStatus stat = fs.getFileStatus(dir);
   assertThat(stat.getPath().toUri().getPath(),
is("/dir"));
   assertThat(stat.isDirectory(), is(true));
```

The Java Interface: Querying the Filesystem

File metadata: FileStatus

Example. ***Demonstrating file status information***

```
assertThat(stat.getLen(), is(0L));
assertThat(stat.getModificationTime(),
is(lessThanOrEqualTo(System.currentTimeMillis())));
assertThat(stat.getReplication(), is((short) 0));
assertThat(stat.getBlockSize(), is(0L));
assertThat(stat.getOwner(),
is(System.getProperty("user.name")));
assertThat(stat.getGroup(), is("supergroup"));
assertThat(stat.getPermission().toString(), is("rwxr-xr-
x"));}}
```

The Java Interface: Querying the Filesystem

File metadata: FileStatus

Example. ***Demonstrating file status information***

- If no file or directory exists, a **FileNotFoundException** is thrown.
- However, if you are interested only in the existence of a file or directory, the **exists()** method on **FileSystem** is more convenient:

```
public boolean exists(Path f) throws IOException
```

The Java Interface: Querying the Filesystem

Listing files:

- Finding information on a single file or directory is useful, but you also often need to be able to list the contents of a directory.
- That's what **FileSystem's listStatus()** methods are for:

```
public FileStatus[] listStatus(Path f) throws IOException
public FileStatus[] listStatus(Path f, PathFilter filter)
throws IOException
public FileStatus[] listStatus(Path[] files) throws
IOException
public FileStatus[] listStatus(Path[] files, PathFilter
filter) throws IOException
```

The Java Interface: Querying the Filesystem

Listing files:

- When the argument is a file, the simplest variant returns an array of **FileStatus** objects of length **1**.
- When the argument is a directory, it returns **zero** or more **FileStatus** objects representing the files and directories contained in the directory.
- Overloaded variants allow a **PathFilter** to be supplied to restrict the files and directories to match.

The Java Interface: Querying the Filesystem

Listing files:

Example. *Showing the file statuses for a collection of paths in a Hadoop filesystem*

```
public class ListStatus {
  public static void main(String[] args) throws
Exception {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri),
conf);
```

The Java Interface: Querying the Filesystem

Listing files:

Example. *Showing the file statuses for a collection of paths in a Hadoop filesystem*

```
Path[] paths = new Path[args.length];
for (int i = 0; i < paths.length; i++) {
  paths[i] = new Path(args[i]);
}
FileStatus[] status = fs.listStatus(paths);
Path[] listedPaths = FileUtil.stat2Paths(status);
for (Path p : listedPaths) {
   System.out.println(p);} } }
```

The Java Interface: Querying the Filesystem

Listing files:

Example. *Showing the file statuses for a collection of paths in a Hadoop filesystem*

We can use this program to find the union of directory listings for a collection of paths:

```
% hadoop ListStatus hdfs://localhost/
hdfs://localhost/user/tom
hdfs://localhost/user
hdfs://localhost/user/tom/books
hdfs://localhost/user/tom/quangle.txt
```

The Java Interface: Querying the Filesystem

File patterns:

- It is a common requirement to process sets of files in a single operation.
- For example, a **MapReduce** job for log processing might analyze a month's worth of files contained in a number of directories.
- Rather than having to enumerate each file and directory to specify the input, it is convenient to use wildcard characters to match multiple files with a single expression, an operation that is known as ***globbing***.

The Java Interface: Querying the Filesystem

File patterns:

- **Hadoop** provides two FileSystem methods for processing **globs**:

```
public FileStatus[] globStatus(Path pathPattern)
throws IOException

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

The Java Interface: Querying the Filesystem

File patterns:

- The **globStatus()** methods return an array of **FileStatus** objects whose paths match the supplied pattern, sorted by path.
- An optional **PathFilter** can be specified to restrict the matches further.

The Java Interface: Querying the Filesystem

File patterns:

Hadoop supports the same set of **glob characters** as the **Unix bash shell**

(*Shown in below table:*)

Glob	Name	Matches
*	*asterisk*	Matches zero or more characters.
?	*question mark*	Matches a single character.
[ab]	*character class*	Matches a single character in the set {a, b}.
[^ab]	*negated character class*	Matches a single character that is not in the set {a, b}.
[a-b]	*character range*	Matches a single character in the (closed) range [a, b], where a is lexicographically less than or equal to b.
[^a-b]	*negated character range*	Matches a single character that is not in the (closed) range [a, b], where a is lexicographically less than or equal to b.
{a,b}	*alternation*	Matches either expression a or b.
\c	*escaped character*	Matches character c when it is a metacharacter.

The Java Interface: Querying the Filesystem

File patterns:

- Imagine that logfiles are stored in a directory structure organized hierarchically by date. So, logfiles for the last day of **2021** would go in a directory named **/2021/12/31**, for example.
- ***Here are some file globs and their expansions:***

Glob	Expansion
/*	/2021 /2022
/*/*	/2021/12 /2022/01
/*/12/*	/2021/12/30 /2021/12/31
/20?	/2021 /2022
/20[2122]	/2021 /2022
/20[21-22]	/2021 /2022
/20[^01234569]	/2021 /2022
/*/*/{31,01}	/2021/12/31 /2022/01/01
/*/*/3{0,1}	/2021/12/30 /2021/12/31
/*/{12/31,01/01}	/2021/12/31 /2022/01/01

The Java Interface: Querying the Filesystem

PathFilter:

- **Glob patterns** are not always powerful enough to describe a set of files you want to access.
- For example, it is not generally possible to exclude a particular file using a **glob pattern**.

The Java Interface: Querying the Filesystem

PathFilter:

The **listStatus()** and **globStatus()** methods of **FileSystem** take an optional **PathFilter**, which allows programmatic control over matching:

```
package org.apache.hadoop.fs;
public interface PathFilter {
  boolean accept(Path path);
}
```

PathFilter is the equivalent of **java.io.FileFilter** for Path objects rather than **File** objects.

The Java Interface: Querying the Filesystem

PathFilter:

Example. *A PathFilter for excluding paths that match a regular expression*

```
public class RegexExcludePathFilter implements PathFilter
{
   private final String regex;
   public RegexExcludePathFilter(String regex) {
      this.regex = regex;
}
public boolean accept(Path path) {
   return !path.toString().matches(regex);
}}
```

The Java Interface: Querying the Filesystem

PathFilter:

- The filter passes only those files that *don't* match the regular expression.
- After the **glob** picks out an initial set of files to include, the filter is used to refine the results.

For example:

```
fs.globStatus(new Path("/2021/*/*"), new
RegexExcludeFilter("^.*/2021/12/31$"))
```

will expand to **/2021/12/30**.

The Java Interface: Querying the Filesystem

PathFilter:

- **Filters** can act only on a file's name, as represented by a Path.
- They can't use a file's properties, such as creation time, as their basis.
- However, they can perform matching that neither **glob patterns** nor **regular expressions** can achieve.
- For example, if you store files in a directory structure that is laid out by date, you can write a **PathFilter** to pick out files that fall in a given date range.

The Java Interface: Deleting Data

- Use the **delete()** method on **FileSystem** to permanently remove files or directories:

```
public boolean delete(Path f, boolean recursive)
throws IOException
```

- If **f** is a file or an empty directory, the value of recursive is ignored.
- A nonempty directory is deleted, along with its contents, only if recursive is **true** (otherwise, an **IOException** is thrown).