|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Page 19 – Example 2-2 | | | | | | |
| The AWK language is a data-driven scripting language for text processing and typically used as a data extraction and reporting tool. It is a standard feature of most Unix-like operating systems. It is included with GNU-based Linux packages. | | | | | | |
| #!/usr/bin/env bash | | **#!**---shebang or hashbang. It is followed by the full path to the interpreter such as /bin/bash. All scripts under UNIX and Linux execute using the interpreter specified on a first line.  Bash is a Unix shell and command language written by Brian Fox for the GNU Project as a free software replacement for the Bourne shell. Released in 1989, it has been distributed widely as a default shell on Linux and OS X. | | | | |
| for year in all/\* | | | | | **for…in** is a loop command; the **for** statement executes the commands enclosed in a body once for each year in the list. | |
| do | | | | | |  |
| echo -ne `basename $year .gz`"\t" | | | **Echo** --- Displays text or variables value on screen.  **-n** --- Do not output the trailing new line.  **-e** --- Enable interpretation of the following backslash escaped characters in the strings. For example, $ echo -e "An apple a day keeps away \t\tdoctor\n"  **\t** --- horizontal tab.  **\n** --- new line.  **basename** --- prints filename NAME with any leading directory components removed. It can optionally also remove any trailing suffix. For example, basename /usr/bin/sort Outputs the string "sort".  Dollar sign ($) allows users to have stored value of a variable. | | | |
| gunzip -c $year | \ | **Gunzip** --- to unzip a ".gz" compressed file; gzip is the counter part of gunzip.  **-c** --- Write output on standard output; keep original files unchanged. If there are several input files, the output consists of a sequence of independently compressed members.  | --- pipe output to input; pipeline symbol.  \ --- The backslash is an escape character, which is used to give characters to commands that would normally have a different meaning in the shell. | | | | | |
| awk '{ temp = substr($0, 88, 5) + 0; | | | | The **substr()** function can extract a portion of a string.  substr(string,position,length) where string is the string to search, position is the number of characters to start looking, and length is the number of characters to extract (default is 1).  **$0** --- get the value of the first string.  **+0** --- convert string to number. | | |
| q = substr($0, 93, 1); | | | | | |  |
| if (temp !=9999 && q ~ /[01459]/ && temp > max) max = temp } | | | | | | **temp !=9999** --- temperature value is not missing.  **q ~ /[01459]/** --- q equals to one of these five values. |
| END { print max }' | | | | | |  |
| done | | | | | |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Page 22 – Example 2-3 | | | | | |
| import java.io.IOException | | | | |  |
| import org.apache.hadoop.io.IntWritable; | | | | |  |
| import org.apache.hadoop.io.LongWritable; | | | | |  |
| import org.apache.hadoop.io.Text; | | | | |  |
| import org.apache.hadoop.mapreduce.Mapper; | | | | |  |
|  | | | | |  |
| public class MaxTemperatureMapper | | | | |  |
| extends Mapper<LongWritable, Text, Text, IntWritable> { | | | | |  |
|  | | | | |  |
| private static final int MISSING = 9999; | | | | Create an integer variable named MISSING | |
|  | | | | |  |
| @Override | The @Override annotation functions more or less as a new method modifier. Its presence indicates to the compiler that the annotated method must override an existing supertype method, either from an interface, or an abstract base class.  The main reason @Override was created was to deal with simple (but nasty) typographical errors. For example, a method mistakenly declared as  public int hashcode(){...}  is in fact not an override - the method name has all lower case letters, so it doesn't exactly match the name of the hashCode() method. It will however compile perfectly well. Such an error is easy to make, and difficult to catch, which is a dangerous combination. Using the @Override annotation prevents you from making such errors. | | | | |
| public void map(LongWritable key, Text value, Context context) | | **Void** --- no return value.  The **map()** method is passed a key and a value. The map() method also provides an instance of Context to write the output to. | | | |
| throws IOException, InterruptedException { | | |  | | |
|  | |  | | | |
| String line = value.toString(); | | Use String variable **line** to get the text **value.** | | | |
| String year = line.substring(15, 19); | | Use String variable **year** to get each year.  The **substring()** method extracts the characters in a string between "start" and "end", including "start" but not including "end" itself. | | | |
| int airTemperature; | |  | | | |
| if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs | | Test whether temperature is positive. | | | |
| airTemperature = Integer.parseInt(line.substring(88, 92)); | | The **Integer.parseInt()** function parses a string argument and returns an integer of the specified radix (the base in mathematical numeral systems); it is also known as java.lang.Integer.parseInt() method. Java.lang package is by default available for every Java program; no import statement is required to include this package.  Examples: parseInt("1111", 2), parseInt("15,123", 10); | | | |
| } else { | |  | | | |
| airTemperature = Integer.parseInt(line.substring(87, 92)); | |  | | | |
| } | |  | | | |
| String quality = line.substring(92, 93); | |  | | | |
| if (airTemperature != MISSING && quality.matches("[01459]")) { | | Character method --- **Matches**  [abc] a, b, or c (simple class);  [^abc] Any character except a, b, or c (negation). | | | |
| context.write(new Text(year), new IntWritable(airTemperature)); | | Write an output record only if the temperature is present and the quality code indicates the temperature reading is OK. | | | |
| } | |  | | | |
| } | |  | | | |
| } | |  | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Page 23 – Example 2-4 | | | |
| import java.io.IOException; | | |  |
| import org.apache.hadoop.io.IntWritable; | | |  |
| import org.apache.hadoop.io.Text; | | |  |
| import org.apache.hadoop.mapreduce.Reducer; | | |  |
|  | | |  |
| public class MaxTemperatureReducer | | |  |
| extends Reducer<Text, IntWritable, Text, IntWritable> { | | Again, four formal type parameters are used to specify the input and output types. The input types of the reduce function must match the output types of the map function: Text and IntWritable. And in this case, the output types of the reduce function are Text and IntWritable, for a year and its maximum temperature. | |
|  | | |  |
| @Override | | |  |
| public void reduce(Text key, Iterable<IntWritable> values, | | | The Iterable interface must be implemented by any class that can be used by Java’s enhanced **for** statement (commonly called “foreach”). |
| Context context) | | |  |
| throws IOException, InterruptedException { | | |  |
|  | | |  |
| int maxValue = Integer.MIN\_VALUE; | | | MIN\_VALUE --- A constant holding the minimum value an int can have, -231. |
| for (IntWritable value : values) { | Java does not support a "**foreach**" keyword but use the "**for**" keyword.  **For** is used to loop over a collection. We use a colon, not an index variable. This enumerates each element in the collection. | | |
| maxValue = Math.max(maxValue, value.get()); | | | Get the max temperature |
| } | | |  |
| context.write(key, new IntWritable(maxValue)); | | |  |
| } | | |  |
| } | | |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Page 24 – Example 2-5 | | | | | | | | |
| import org.apache.hadoop.fs.Path; | | | | | | | |  |
| import org.apache.hadoop.io.IntWritable; | | | | | | | |  |
| import org.apache.hadoop.io.Text; | | | | | | | |  |
| import org.apache.hadoop.mapreduce.Job; | | | | | | | |  |
| import org.apache.hadoop.mapreduce.lib.input.FileInputFormat; | | | | | | | |  |
| import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat; | | | | | | | |  |
|  | | | | | | | |  |
| public class MaxTemperature { | | | | | | | |  |
|  | | | | | | | |  |
| public static void main(String[] args) throws Exception { | | | | | | | |  |
| if (args.length != 2) { | | Requires 2 arguments: input path and output path. | | | | | | |
| System.err.println("Usage: MaxTemperature <input path> <output path>"); | | | | | | | |  |
| System.exit(-1); | | | | | | | |  |
| } | | | | | | | |  |
|  | | | | | | | |  |
| Job job = new Job(); | A **Job** object forms the specification of the job and gives you control over how the job is run. When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster). | | | | | | | |
| job.setJarByClass(MaxTemperature.class); | | | Rather than explicitly specify the name of the JAR file, we can pass a class in the Job’s setJarByClass() method, which  Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class. | | | | | |
| job.setJobName("Max temperature"); | | | | | | | |  |
|  | | | | | | | |  |
| FileInputFormat.addInputPath(job, new Path(args[0])); | | | | | | | Specify the **input** path by calling the static addInputPath() method on FileInputFormat. | |
| FileOutputFormat.setOutputPath(job, new Path(args[1])); | | | | | | | Specify the **output** path by calling the static addOutputPath() method on FileInputFormat. | |
|  | | | | | | | |  |
| job.setMapperClass(MaxTemperatureMapper.class); | | | | | Specify the map and reduce types to use via the setMapperClass() and setReducerClass() methods. | | | |
| job.setReducerClass(MaxTemperatureReducer.class); | | | | |
|  | | | | | | | |  |
| job.setOutputKeyClass(Text.class); | | | | The **setOutputKeyClass()** and **setOutputValueClass()** methods control the output types for the map and the reduce functions, which are often the same, as they are in our case. If they are different, the map output types can be set using the methods **setMapOutputKeyClass()** and **setMapOutputValueClass()**.  The **input** types are controlled via the input format, which we have not explicitly set because we are using the default **TextInputFormat**. | | | | |
| job.setOutputValueClass(IntWritable.class); | | | |
|  | | | |  | | | | |
| System.exit(job.waitForCompletion(true) ? 0 : 1); | | | | | | The waitForCompletion() method on Job submits the job and waits for it to finish. The return value of the waitForCompletion() method is a Boolean indicating success (true) or failure (false), which we translate into the program’s exit code of 0 or 1. | | |
| } | | | | | | | |  |
| } | | | | | | | |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Page 56 – Example 3-1 | | | |
| import java.io.InputStream; | | |  |
| import java.net.URL; | | |  |
|  | | |  |
| import org.apache.hadoop.fs.FsUrlStreamHandlerFactory; | | |  |
| import org.apache.hadoop.io.IOUtils; | | |  |
|  | | |  |
| // vv URLCat | | |  |
| 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); | **4096** --- is the buffer size used for copying;  **False** --- do not close the streams when the copy is complete. We close the input stream ourselves, and System.out doesn’t need to be closed. | | |
| } finally { | |  | |
| IOUtils.closeStream(in); | |  | |
| } | |  | |
| } | |  | |
| } | |  | |

|  |  |  |
| --- | --- | --- |
| Page 88 – Example 4-3 | | |
| import org.apache.hadoop.conf.Configuration; | |  |
| import org.apache.hadoop.io.IOUtils; | |  |
| import org.apache.hadoop.io.compress.\*; | |  |
| import org.apache.hadoop.util.ReflectionUtils; | |  |
|  | |  |
| // vv PooledStreamCompressor | |  |
| public class PooledStreamCompressor { | |  |
|  | |  |
| public static void main(String[] args) throws Exception { | |  |
| String codecClassname = args[0]; | |  |
| Class<?> codecClass = Class.forName(codecClassname); | | ? is a wildcard and means any type. So, Class<?> means any type of class. |
| Configuration conf = new Configuration(); | |  |
| CompressionCodec codec = (CompressionCodec)  ReflectionUtils.newInstance(codecClass, conf); | (CompressionCodec) is known as a **cast** (see Java book p.54). A cast is an instruction to the compiler to convert one type into another. Here,  (CompressionCodec) ReflectionUtils.newInstance(codecClass, conf) converts ReflectionUtils type to CompressionCodec type. | |
| /\*[\*/Compressor compressor = null; | |  |
| try { | |  |
| compressor = CodecPool.getCompressor(codec);/\*]\*/ | |  |
| CompressionOutputStream out = | |  |
| codec.createOutputStream(System.out, /\*[\*/compressor/\*]\*/); | |  |
| IOUtils.copyBytes(System.in, out, 4096, false); | |  |
| out.finish(); | |  |
| /\*[\*/} finally { | |  |
| CodecPool.returnCompressor(compressor); | |  |
| }/\*]\*/ | |  |
| } | |  |
| } | |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Page 100 – Example 4-6 | | | | | |
| import java.nio.ByteBuffer; | | |  | | |
| import org.apache.hadoop.io.Text; | | |  | | |
|  | | |  | | |
| // vv TextIterator | | |  | | |
| public class TextIterator { | | |  | | |
|  | | |  | | |
| public static void main(String[] args) { | | |  | | |
| Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00"); | | | | See Table 4-8. Unicode Characters | |
|  | | | | |  |
| ByteBuffer buf = ByteBuffer.wrap(t.getBytes(), 0, t.getLength()); | | **ByteBuffer.wrap(byte[] array, int offset, int length)**  Wraps a byte array into a buffer. The new buffer's capacity will be array.length, its position will be offset, its limit will be offset + length. Its backing array will be the given array, and its array offset will be zero. | | | |
| int cp; | | | | |  |
| while (buf.hasRemaining() && (cp = Text.bytesToCodePoint(buf)) != -1) { | | Let cp get the Unicode code point value.  bytesToCodePoint() returns -1 if the end of the string is detected. | | | |
| System.out.println(Integer.toHexString(cp)); | Convert the int cp into a hexadecimal string.  **Integer.toHexString**() – Returns a string representation of the integer argument as an unsigned integer in base 16. This value is converted to a string of ASCII digits in hexadecimal (base 16) with no extra leading 0s.  If the unsigned magnitude is zero, it is represented by a single zero character '0' ('\u0030'); otherwise, the first character of the representation of the unsigned magnitude will not be the zero character. The following characters are used as hexadecimal digits:  0123456789abcdef  These are the characters '\u0030' through '\u0039' (for the 10 numbers) and '\u0061' through '\u0066' (for the six letters). | | | | |
| } | | | | |  |
| } | | | | |  |
| } | | | | |  |

|  |  |  |
| --- | --- | --- |
| Page 131 – Example 4-14 | | |
| import java.io.IOException; | |  |
| import java.net.URI; | |  |
| import org.apache.hadoop.conf.Configuration; | |  |
| import org.apache.hadoop.fs.FileSystem; | |  |
| import org.apache.hadoop.fs.Path; | |  |
| import org.apache.hadoop.io.IOUtils; | |  |
| import org.apache.hadoop.io.IntWritable; | |  |
| import org.apache.hadoop.io.SequenceFile; | |  |
| import org.apache.hadoop.io.Text; | |  |
| // vv SequenceFileWriteDemo | |  |
| public class SequenceFileWriteDemo { | |  |
|  | |  |
| private static final String[] DATA = { | |  |
| "One, two, buckle my shoe", | |  |
| "Three, four, shut the door", | |  |
| "Five, six, pick up sticks", | |  |
| "Seven, eight, lay them straight", | |  |
| "Nine, ten, a big fat hen" | |  |
| }; | |  |
| public static void main(String[] args) throws IOException { | |  |
| String uri = args[0]; | |  |
| Configuration conf = new Configuration(); | |  |
| FileSystem fs = FileSystem.get(URI.create(uri), conf); | |  |
| Path path = new Path(uri); | |  |
|  | |  |
| IntWritable key = new IntWritable(); | |  |
| Text value = new Text(); | |  |
| SequenceFile.Writer writer = null; | |  |
| try { | |  |
| writer = SequenceFile.createWriter(fs, conf, path, | |  |
| key.getClass(), value.getClass()); | |  |
|  | |  |
| for (int i = 0; i < 100; i++) { | |  |
| key.set(100 - i); | |  |
| value.set(DATA[i % DATA.length]); | % (Modulus)  Divides left hand operand by right hand operand and returns remainder. Example: B % A will give 0. | |
| System.out.printf("[%s]\t%s\t%s\n", writer.getLength(), key, value); | printf format specifiers  **%s** --- a string of characters  **\t** --- horizontal tab  **\n** --- new line | |
| writer.append(key, value); | |  |
| } | |  |
| } finally { | |  |
| IOUtils.closeStream(writer); | |  |
| } | |  |
| } | |  |
| } | |  |