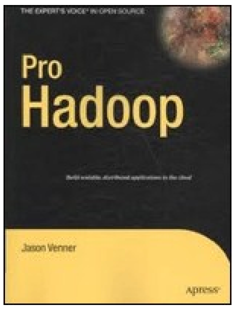

Pro Hadoop

by Jason Venner

Apress. (c) 2009. Copying Prohibited.

Appendix A: The JobConf Object in Detail

Overview

Everything in a job is controlled via the `JobConf` object; it is the center of the universe for a MapReduce job. The framework will take the `JobConf` object and render it to XML; then all the tasks will load that XML when they start. This section will cover all the relevant methods (as of Hadoop Core 0.19.0) and provide some basic usage examples.

The `JobConf` class inherits from the `Configuration` class. Because the `JobConf` object is the primary interface between the programmer and the framework, I'll detail all methods available to the user of a `JobConf` without distinguishing which methods come from the `Configuration` base class. I suggest that you create and use only `JobConf` objects. By default, a new `JobConf` object loads and merges the `hadoop-default.xml` and `hadoop-site.xml` files, as shown in Figure A-1.

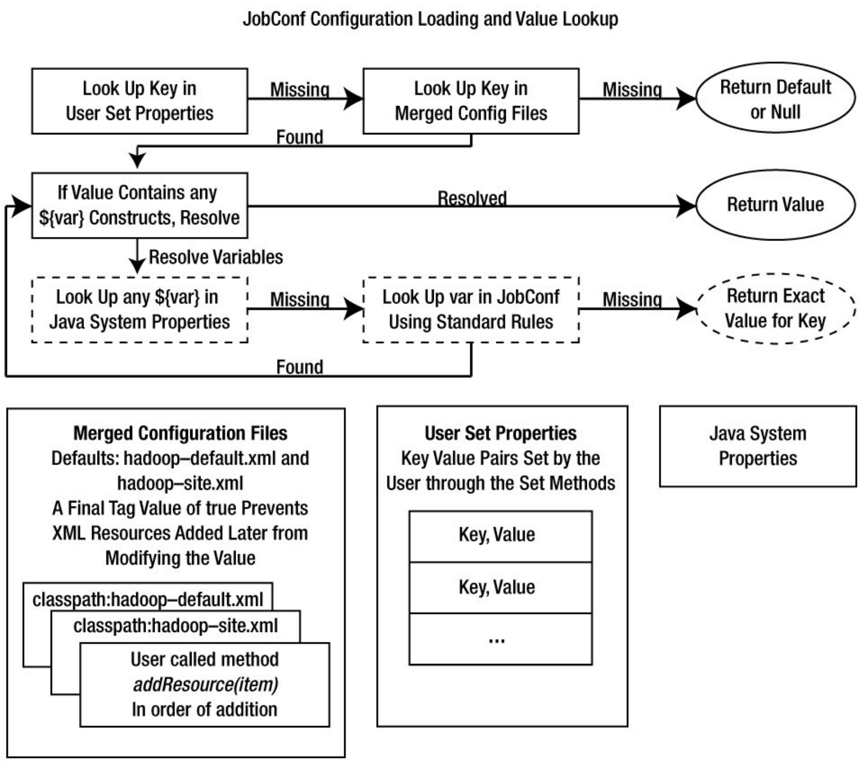
JobConf Configuration Loading and Value Lookup
Look Up Key in User Set Properties
Missing
Look Up Key in Merged Config Files
Missing
Return Default or Null
Found
If Value Contains any ${var} Constructs, Resolve
Resolved
Return Value
Resolve Variables
Look Up any ${var} in Java System Properties
Missing
Look Up var in JobConf Using Standard Rules
Missing
Return Exact Value for Key
Found
Merged Configuration Files
Defaults: hadoop–default.xml and hadoop–site.xml
A Final Tag Value of true Prevents XML Resources Added Later from Modifying the Value
classpath:hadoop–default.xml
classpath:hadoop–site.xml
User called method addResource(item) In order of addition
User Set Properties
Key Value Pairs Set by the User through the Set Methods
Key, Value
Key, Value
...
Java System Properties

Figure A-1: How configuration data is loaded into the JobConf object and resolved

The default files `hadoop-default.xml` and `hadoop-site.xml`, and any additional userspecified XML resources specified by the `AddResource()` method, are found in the Java Virtual Machine (JVM) classpath and merged into the configuration data in the order added. Configuration values that are loaded as resources are stored separately from the values that are set via setter calls. The values that were loaded via resources are removed by a call to `reloadConfiguration()`, whereas all the values are removed by a call to `clear()`. When looking for a value, a value set by a setter call takes precedence over a value loaded from a resource. The lookup process is described in Figure A-1.

Each configuration item is a name and value pair with an optional final parameter. These parameters tell the Hadoop framework code how to contact the cluster, are defaults for various attributes, and allow for passing arbitrary values to the tasks. The `conf/hadoop-default.xml` file has a list of most of the Hadoop Core framework parameters. Other parameters are found only by reading the source code.

You can set arbitrary names for value pairs in the configuration, and these name-value pairs are made available to MapReduce tasks. Values that are objects are serialized and then deserialized by each MapReduce task when tasks start.

The naming convention for configuration parameters is usually `area.subarea.specific` name. The parameters that configure the distributed file system start with `dfs`, and the parameters that configure the MapReduce framework start with `mapred`.

JobConf Object in the Driver and Tasks

The `JobConf` object has two roles. In the job driver, the `JobConf` object is constructed with all the parameters for the job. At job runtime, required data, the `JobConf` object, JAR files, archives, and other resources are stored in the Hadoop Distributed File System (HDFS) in a job-specific directory.

In the task, the `JobConf` object is reconstituted and localized, and it is given a set of directories between the paths defined in `mapred.local.dir`. Any items that must be referenced from the local file system, such as the job JAR file or other items passed via the `DistributedCache`, are unpacked into these local directories and the path references to items in the configuration are adjusted to be the task local path. The classpath for the JVM that the task will run in is also set up for the task to include the location on the local file system that the classpath resources were unpacked into.

JobConf Is a Properties Table

The `JobConf` instances maintain a table of key/value pairs for all the configuration parameters. The values are all stored as `String` objects and are serialized if they are objects. At the lowest level, operations get a value for a key or store a value for a key.

Variable Expansion

The `JobConf` object performs variable expansion on values when raw returned values have special text embedded in them. The syntax is `${key}`, which will be replaced by the value of `key`.

In the configuration files you will often see values in this form: `<value>${key}something</value>`. If `key` exists in the `System.properties` or in the current configuration, a get method will replace `${key}` with the value of `key`.

Note in Listing A-1 that values with `${key}` have the key resolved against `System.properties`. If there is no value found, the value is resolved against the configuration in the `JobConf` object. This expansion is recursive in that if the expansion contains another `${item}` reference, the `${item}` is expanded. This process continues until there are no items that are candidates for expansion or there are no items that can be expanded.

Listing A-1: XML File Used in the Variable Expansion Example: variable-expansion-example.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
   <property>
       <name>no.expansion</name>
       <value>no.expansion Value</value>
   </property>
   <property>
       <name>expansion.from.configuration</name>
       <value>The value of no.expansion is ${no.expansion}</value>
   </property>
   <property>
       <name>java.io.tmpdir</name>
       <value>failed attempt to override a System.properties value
        for variable expansion</value>
   </property>
   <property>
       <name>order.of.expansion</name>
       <value>The value of java.io.tmpdir
        from System.properties: ${java.io.tmpdir}</value>
   </property>

   <property>
       <name>expansion.from.JDK.properties</name>
       <value>The value of java.io.tmpdir from
          System.properties: ${java.io.tmpdir}</value>
   </property>
   <property>
       <name>nested.variable.expansion</name>
       <value>Will expansion.from.configuration's
  value have substition: [${expansion.from.configuration}]</value>
```

```
   </property>
</configuration>
```

The code example in Listing A-2 looks up keys defined in Listing A-1. The first key examined is `no.expansion`; in Listing A-1, the value is defined as `no.expansion Value`, which is the result printed. The value of `no.expansion` is `[no.expansion Value]`.

Listing A-2: Example of Variable Expansion: the Key is Defined in the JDK System Properties VariableExpansion.java

```
package com.apress.hadoopbook.examples.jobconf;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;

import org.apache.hadoop.mapred.JobConf;

/** Simple class to demonstrate variable expansion
 * within hadoop configuration values.
 * This relies on the hadoop-core jar, and the
 * hadoop-default.xml file being in the classpath.
 */
public class VariableExpansion {
    public static void main( String [] args ) throws IOException {
        /** Get a local file system object, so that we can construct a local Path
         * That will hold our demonstration configuration.
         */
        /** Construct a JobConf object with our configuration data. */
        JobConf conf = new JobConf( "variable-expansion-example.xml" );
        System.out.println( "The value of no.expansion is [" +
           conf.get( "no.expansion") + "]" );
        System.out.println( "The value of expansion.from.configuration is [" +
           conf.get("expansion.from.configuration") + "]");
        System.out.println( "The value of expansion.from.JDK.properties is ["
           + conf.get("expansion.from.JDK.properties") + "]");
        System.out.println( "The value of java.io.tmpdir is [" +
           conf.get("java.io.tmpdir") + "]" );
        System.out.println( "The value of order.of.expansion is [" +
           conf.get("order.of.expansion") + "]" );
        System.out.println( "Nested variable expansion for nested." +
          "variable.expansion is [" +
     conf.get("nested.variable.expansion") +"]");
    }
}
```

The next item demonstrating simple substitution is `expansion.from.configuration`, which is given the value of `The value of no.expansion is ${no.expansion}` in Listing A-1. The expanded result is `The value of no.expansion is no.expansion Value`, showing that the `${no.expansion}` was replaced by the value of `no.expansion` in the configuration.

The item for `expansion.from.JDK.properties` demonstrates that the key/value pairs in the `System.properties` are used for variable expansion. The value defined in Listing A-1 is `The value of java.io.tmpdir from System.properties: ${java.io.tmpdir}`, and the result of the expansion is `The value of java.io.tmpdir from System.properties: C:\DOCUME~1\Jason\LOCALS~1\Temp\]`. Note that the actual system property value for `java.io.tmpdir` is used, not the value stored in the configuration for `java.io.tmpdir`, `failed attempt to override a System.properties value for variable expansion`.

The final example demonstrates that the variable expansion results are candidates for further expansion. The key `nested.variable.expansion` has a value of `Will expansion.from.configuration's value have substition: [${expansion.from.configuration}]`, `expansion.from.configuration` has a value of `The value of no.expansion is ${no.expansion}`, and `no.expansion` has the value of `no.expansion Value`.

As expected in Listing A-2, the `conf.get("expansion.from.configuration")` returns `The value of no.expansion is no.expansion Value]`.

```
The value of no.expansion is [no.expansion Value]
The value of expansion.from.configuration is ➥
[The value of no.expansion is no.expansion Value]
The value of expansion.from.JDK.properties is ➥
[The value of java.io.tmpdir from System.properties: ➥
C:\DOCUME~1\Jason\LOCALS~1\Temp\]
The value of java.io.tmpdir is ➥
[failed attempt to override a System.properties value for variable expansion]
The value of order.of.expansion is ➥
[The value of java.io.tmpdir from System.properties: ➥
C:\DOCUME~1\Jason\LOCALS~1\Temp\]
Nested variable expansion for nested.variable.expansion is ➥
[Will expansion.from.configuration's value have substition: ➥
[The value of no.expansion is no.expansion Value]]
```

Final Values

The Hadoop Core framework gives you a way to mark some keys in a configuration file as final. The stanza `<final>true</final>` prevents later configuration files from overriding the value specified. The `<final>` tag does not prevent the user from overriding the value via the set method. The example in Listing A-3 creates several XML files in the temporary directory: the first file, `finalFirst`, contains the declaration of a configuration key, `final.first`, which has the value `first final value` declared final via `<final>true</final>`. The second file, `finalSecond`, also defines `final.first` with the value `This should not override the value of final.first`. After loading the two resource files via `JobConf conf = new JobConf( finalFirst.toURI().toString() );` and `conf.addResource( finalSecond.toURI().toString() );`, the value of the key `final.first` is gotten via `conf.get("final.first")` and found to be `first final value`. The next example calls `conf.set ("final.first", "This willoverride a final value, when applied by conf.set");` to demonstrate that the setter methods will override a value marked `final`.

Listing A-3: Sample Code Showing DemonstrationOfFinal.java

```java
package com.apress.hadoopbook.examples.jobconf;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;

import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.conf.Configuration;

/** Demonstrate how the final tag works for configuration files and is ignored
 *  by the {@link Configuration#set(java.lang.String,java.lang.String)} operator
 * This relies on the hadoop-core jar, and the
 * hadoop-default.xml file being in the classpath.
 */
public class DemonstrationOfFinal {
    /** Save xml configuration data to a temporary file,
     * that will be deleted on jvm exit
     *
     * @param configData The data to save to the file
     * @param baseName The base name to use for the file, may be null
     * @return The File object for the file with the data written to it.
     * @throws IOException
     */

    static File saveTemporaryConfigFile( final String configData,
              String baseName ) throws IOException {
        if (baseName==null) {
            baseName = "temporaryConfig";
        }
        /** Make a temporary file using the JVM file utilities */
        File tmpFile = File.createTempFile(baseName, ".xml");
        tmpFile.deleteOnExit(); /** Ensure the file is deleted
                                 * when this jvm exits. */
        Writer ow = null;
        /** Ensure that the output writer is closed even on errors. */
        try {
            ow = new OutputStreamWriter( new FileOutputStream( tmpFile ), "utf-8");
            ow.write( configData );
            ow.close();
            ow = null;
        } finally {
            if (ow!=null) {
                try {
                    ow.close();
                } catch (IOException e) {
                //  ignore, as we are already handling the real exception
                }
            }
        }
        return tmpFile;

    }
    public static void main( String [] args ) throws IOException {
        /** Get a local file system object, so that we can construct a local Path
         * That will hold our demonstration configuration.
         */

        File finalFirst = saveTemporaryConfigFile(
                "<?xml version=\"1.0\"?>\n" +
                "<?xml-stylesheet type=\"text/xsl\" ➥
  href=\"configuration.xsl\"?>\n" +
                "<configuration>\n" +
                "   <property>\n" +
                "       <name>final.first</name>\n" +
                "       <value>first final value.</value>\n" +
                "       <final>true</final>\n" +
                "   </property>\n" +
                "</configuration>\n",
                "finalFirst" );

        File finalSecond = saveTemporaryConfigFile(
                "<?xml version=\"1.0\"?>\n" +
                "<?xml-stylesheet type=\"text/xsl\" ➥
  href=\"configuration.xsl\"?>\n" +
                "<configuration>\n" +
                "   <property>\n" +
                "       <name>final.first</name>\n" +
                "       <value>This should not override the ➥
  value of final.first.</value>\n" +
                "   </property>\n" +
                "</configuration>\n",
                "finalSecond" );


        /** Construct a JobConf object with our configuration data. */
        JobConf conf = new JobConf( finalFirst.toURI().toString() );
        /** Add the additional file that will attempt to overwrite
          * the final value of final.first. */
        conf.addResource( finalSecond.toURI().toString());
        System.out.println( "The final tag in the first file will " +
          "prevent the final.first value in the second configuration file "
     +"from inserting into the configuration" );
        System.out.println( "The value of final.first in the " +
         "configuration is [" + conf.get("final.first") + "]" );
        /** Manually set the value of final.first to demonstrate
          * it can be overridden. */
        conf.set("final.first", "This will override a final value,➥
          when applied by conf.set");
        System.out.println( "The value of final.first in the configuration"
         + " is [" + conf.get("final.first") + "]" );

    }
}
```

```
The final tag in the first file will prevent the final ➥
first value in the second configuration file from inserting into the configuration
The value of final.first in the configuration is [first final value.]
The value of final.first in the configuration is ➥
[This will override a final value when applied by conf.set]
```

Constructors

All code that creates and launches a MapReduce job into a Hadoop cluster creates a `JobConf` object. The framework provides several methods for creating the object.

public JobConf()

This is the default constructor. This constructor should not be used because it doesn't provide the framework with information about the JAR file that this class was loaded from.

public JobConf(Class exampleClass)

This common use case constructor is the constructor you should use. The archive that the `exampleClass` was loaded from will be made available to the MapReduce tasks. The type of `exampleClass` is arbitrary; `exampleClass` is used only to find the classpath resource that the `exampleClass` was loaded from. The containing JAR file will be made available as a classpath item for the job tasks. The JAR is actually passed via the `DistributedCache` as a classpath archive. `exampleClass` is commonly the mapper or reducer class for the job, but it is not required to be so.

> **Tip** The task JVMs are run on different physical machines and do not have access to the classpath or the classpath items of the JVM that submits the job. The only way to set the classpath of the task JVMs is to either set the classpath in the `conf/hadoop-env.sh` script or pass the items via the `DistributedCache`.

public JobConf(Configuration conf)

This constructor is commonly used when your application already has constructed a `JobConf` object and wants a copy to use for an alternate job. The configuration in *`conf`* is copied into the new `JobConf` object.

It is very handy when unit testing because as the unit test can construct a standard `JobConf` object, and each individual test can use it as a reference and change specific values.

If your driver launches multiple MapReduce jobs, each job should have its own `JobConf` object, and the pattern described previously for unit tests is ideal to support this.

public JobConf(Configuration conf, Class exampleClass)

Construct a new `JobConf` object that inherits all the settings of the passed-in `Configuration` object `conf`, and make the archive that `exampleClass` was loaded from available to the MapReduce tasks.

Classes that launch jobs that may have unit tests or be called as part of a sequence of Hadoop jobs should provide a run method that accepts a `Configuration` object and calls this constructor to make the `JobConf` object for that class's job. This way, the unit test or calling code can preconfigure the configuration, and this class can customize its specific variables and launch the job.

Listing A-4: Sample Code Fragment of a Class run Method

```
/** Code Fragment to demonstrate a run method that can be called
 *from a unit test, or from a driver that launches multiple
```

```
     * Hadoop jobs.
     *
     * @param defaultConf The default configuration to use
     * @return The running job object for the completed job.
     * @throws IOException
     */
    public RunningJob run( Configuration defaultConf ) throws IOException
    {
       /** Construct the JobConf object from the passed-in object.
        * Ensure that the archive that contains this class will be
        * provided to the map and reduce tasks.
        */
        JobConf conf = new JobConf( defaultConf, this.getClass() );
        /**
         * Set job specific parameters on the conf object.
         *
         */
        conf.set( "our.parameter", "our.value" );

        RunningJob job = JobClient.runJob(conf);
        return job;
    }
```

public JobConf(String config)

Construct a `JobConf` object with configuration data loaded from the file that `config` is a path to.

public JobConf(Path config)

Construct a `JobConf` object and load configuration values from the XML data found in the file `config`. This constructor is used by the TaskTracker to construct the `JobConf` object from the job-specific configuration file that was written out by the Hadoop framework.

public JobConf(boolean loadDefaults)

This method is identical to the no-argument constructor unless the `loadDefaults` value is `false`. If `loadDefaults` is `false`, `hadoop-site.xml` and `hadoop-default.xml` are not loaded.

Methods for Loading Additional Configuration Resources

The methods described in this section load an XML configuration file resource and store it in the `JobConf` parameter set. The order in which these methods are called is important because the contents specified by the most recent call will override values supplied earlier.

If a specified resource cannot be loaded or parsed as valid configuration XML, a `RuntimeException` will be thrown unless quiet mode is enabled via a call to `setQuietMode(true)`.

Each call to one of these methods results in the complete destruction of the configuration data that resulted from the loading and merging of the XML resources. There are no changes made to the configuration parameters that have been created via the set methods. The entire set of XML resources is reparsed and merged on the next method call that reads or sets a configuration parameter.

These resource items follow the same rules as with the `hadoop-default.xml` and `hadoop-site.xml` files, and a parameter in a resource object can tag itself as `final`. In this case, resource objects loaded later may not change the value of the parameter.

Listing A-5: Sample Final Parameter Declaration

```
<property>
   <name>my.final.parameter</name>
   <value>unchanging</value>
   <final>true</final>
</property>
```

public void setQuietMode(boolean quietmode)

If `quietmode` is `true`, no log messages will be generated when loading the various resources into the configuration. If a resource cannot be parsed, no exception will be thrown.

If `quietmode` is `false`, a log message will be generated for each resource loaded. If a resource cannot be parsed, a `RuntimeException` will be thrown.

public void addResource(String name)

Load the contents of `name`. The parameter is loaded from the current classpath by the JDK `ClassLoader.getResource` method. `name` can be a simple string or a URL.

The default configuration has two `addResource( String name )` calls: one for `hadoop-default.xml` and the other for `hadoop-site.xml`.

> **Caution** The first `hadoop-default.xml` file and the first `hadoop-site.xml` file in your classpath are loaded. It is not uncommon for these files to accidentally be bundled into a JAR file and end up overriding the cluster-specific configuration data in the `conf` directory. A problem often happens with jobs that are not run through the `bin/hadoop` script and do not have a `hadoop-default.xml` or `hadoop-site.xml` file in their classpath.

public void addResource(URL url)

This method explicitly loads the contents of the passed-in URL, `url`, into the configuration.

public void addResource(Path file)

This method explicitly loads the contents of `file` into the configuration.

public void addResource(InputStream in)

Load the XML configuration data from the supplied `InputStream in` into the configuration.

public void reloadConfiguration()

Clear the current configuration, *excluding* any parameters set using the various `set` methods, and reload the configuration from the resources that have been specified. If the user has not specified any resources, the default pair of `hadoop-default.xml` and `hadoop-site.xml` will be used.

This method actually just clears the existing configuration, and the reload will happen on the next get or set.

Basic Getters and Setters

The methods in this section get and set basic types:

- In general, if the framework cannot convert the value stored under a key into the specific type required, a `RuntimeException` will be thrown.
- If the value being retrieved is to be a numeric type, and the value cannot be converted to the numeric type, a `NumberFormatException` will be thrown.
- For boolean types, a value of `true` is required for a `true` return. Any other value is considered `false`.
- For values that are class names, if the class cannot be instantiated, or the instantiated class is not of the correct type, a `RuntimeException` will be thrown.

The framework stores sets of things as comma-separated lists. There is no mechanism currently to escape a comma that must be a part of an individual item in a list.

Under the covers, all data is stored as a java `String` object. All items stored are serialized into a `String` object, and all values retrieved are deserialized from a `String` object. The user is required to convert objects into `String` representations to store arbitrary objects in the configuration and is responsible for re-creating the object from the stored `String` when retrieving the object.

public String get(String name)

This is the basic getter: it returns the `String` version of the value of `name` if `name` has a value or if the method returns `null`. Variable expansion is completed on the returned value. If the value is a serialized object, the results of the variable expansion may be incorrect.

public String getRaw(String name)

Returns the raw `String` value for `name` if `name` exists in the configuration; otherwise returns `null`. No variable expansion is done. This is the method to use to retrieve serialized objects.

public void set(String name, String value)

Stores the `value` under the key `name` in the configuration. Any prior value stored under `name` is discarded, even if the key was marked final.

public String get(String name, String defaultValue)

This method behaves as the `get()` method does: it returns `defaultValue` if `name` does not have a value in the configuration. This method is ideal to use for `get()` operations that must return a value and there is a sensible default value.

public int getInt(String name, int defaultValue)

Many properties stored in the configuration are simple integers, such as the number of reduces, `mapred.reduce.tasks`. If the underlying value for `name` is missing or not convertible to an `int`, the `defaultValue` is returned. If the value starts with a leading `0x` or `0X`, the value will be interpreted as a hexadecimal value.

public void setInt(String name, int value)

Stores the `String` representation of `value` in the configuration under the key `name`. Any prior value associated with `name` will be lost.

public long getLong(String name, long defaultValue)

Many properties stored in the configuration are simple long values, such as the file system block size `dfs.block.size`. If the underlying value for `name` is missing or not convertible to a long, the `defaultValue` is returned. If the value starts with a leading `0x` or `0X`, the value will be interpreted as a hexadecimal value.

public void setLong(String name, long value)

Stores the `String` representation of `value` in the configuration under the key `name`. Any prior value associated with `name` will be lost.

public float getFloat(String name, float defaultValue)

Some properties stored in the configuration are simple floating-point values. You might want to pass a float value to the mapper or reducer, which would use this method to get the float value. If the underlying value for `name` is missing or not convertible to a float, the `defaultValue` is returned.

public boolean getBoolean(String name, boolean defaultValue)

Many properties stored in the configuration are simple `boolean` values, such as the controlling speculative execution for map tasks, `mapred.map.tasks.speculative.execution`. If the underlying value for `name` is missing or not convertible to a `boolean` value, the `defaultValue` is returned. The only acceptable `boolean` values are `true` or false. The comparison is case sensitive, so a value of True will fail to convert, and the defaultValue will be returned.

public void setBoolean(String name, boolean value)

Convert the boolean value to the String true or the String false and store it in the configuration under the key name. Any prior value associated with name will be lost.

Ranges

The configuration supports storing basic types such as various numbers, boolean values, text, and class names. The configuration also supports a type called a Range. It is two integer values, in which the second integer is larger than the first. An individual range is specified by a String containing a -, a dash character that can also have a leading and trailing integer.

If the leading integer is absent, the first range value takes the value 0. If the trailing integer is absent, the second range value takes the value of Integer.MAX_VALUE.

The simplest range is -, which is the range 0 to Integer.MAX_VALUE. The range -35 parses as 0 to 35. The range 40parses as 40 to Integer.MAX_VALUE. The range 40–50 parses as 40 to 50. Multiple ranges may be separated by ,: a comma character such as 1–5,7–9,13–50.

public Configuration.IntegerRanges getRange(String name, String defaultValue)

getRange is a relatively unusual method and obtains a named range. It is currently not widely used. As of Hadoop 0.19.0 it is used only to determine which MapReduce tasks to profile. As of Hadoop 0.19.0 there is no corresponding set method, and the base set(String name, String value) is used to set a range The value has to be the valid String representation of a range or later calls to the getRange method for name will result in an exception being thrown.

The defaultValue must be passed in as a valid range. String null may not be passed as the default value, or else a NullPointerException will be thrown.

This method looks up the value of name in the configuration, and if there is no value, the defaultValue will be used. The resulting value will then be parsed as an IntegerRanges object and that result returned. If the parsing fails, an IllegalArgumentException will be thrown.

Note If there is a value for name in the configuration and it cannot be parsed as an IntegerRanges object, the defaultValue will be ignored, and an IllegalArgumentException will be thrown.

public Collection<String> getStringCollection(String name)

The JobConf and Configuration objects (at least through Hadoop 0.19.0) handle parameters that are sets of String objects by storing them internally as comma-separated lists in a single String. There is no provision for escaping the commas.

getStringCollection will get the value associated with name in the configuration and split the String on commas and return the resulting Collection.

Listing A-6: Sample Use of public Collection<String> getStringCollection(String name)

```
conf.set( "path.set", "path1,path2,path3,path4");
Collection<String> pathSet = conf.getStringCollection("path.set");
for( String path : pathSet ) {
    System.out.println( path );
}
```

```
path1
path2
path3
```

```
path4
```

public String[] getStrings(String name)

The `JobConf` and `Configuration` objects (at least through Hadoop 0.19.0) handle parameters that are sets of `String` objects by storing them internally as comma-separated lists in a single `String`. There is no provision for escaping the commas.

This method gets the value associated with `name` in the configuration, splits the `String` on commas, and returns the resulting array (see Listing A-7).

Listing A-7: Sample Use of public String[] getStrings(String name)

```
conf.set( "path.set", "path1,path2,path3,path4");
String[] pathSet = conf.getStrings("path.set");
for( String path : pathSet ) {
    System.out.println( path );
}
```

```
path1
path2
path3
path4
```

Java 1.5 and Beyond Varags Syntax

As of Java 1.5, variable argument lists are supported for method calls. The declaration of the last parameter may have an ellipsis between the type and the name, `type...name`. The caller can place an arbitrary number of objects of `type` in the method call, and the member method will receive an array of type with the elements from the caller's call. For the method `X(String .... strings)`, a call of the form `X("one","two","three")` would result in the variable `strings` being a three-element array of `String` objects containing `"one"`, `"two"`, `"three"`.

For more details, please visit `http://java.sun.com/j2se/1.5.0/docs/guide/language/varargs. html`.

public String[] getStrings(String name, String… defaultValue)

The `JobConf` and `Configuration` objects (at least through Hadoop 0.19.0) handle parameters that are sets of `String` objects by storing them internally as comma-separated lists in a single `String`. There is no provision for escaping the commas.

This method will get the value associated with `name` in the configuration and split the `String` on commas and return the resulting array (see Listing A-8). If there is no value stored in the configuration for `name`, the array built from the `defaultValue` parameters will be returned.

Listing A-8: Sample Use of public String[] getStrings(String name, String… defaultValue)

```
JobConf empty = new JobConf(false); /** Create an empty configuration to
 * ensure the default value is used in our getStrings example.*/
String[] pathSet = conf.getStrings("path.set", "path1", "path2", "path3", path4");
for( String path : pathSet ) {
    System.out.println( path );
}
```

```
path1
path2
path3
```

```
path4
```

public void setStrings(String name, String… values)

Stores the set of Strings provided in values under the key name in the configuration, deleting any prior value (see Listing A-9). The set of String objects defined by values is concatenated using the comma (,) character as a separator, and the resulting String is stored in the configuration under name.

Listing A-9: Sample Use of public void setStrings(String name, String… values)

```
conf.setStrings( "path.set", "path1", "path2, "path3", "path4");
String[] pathSet = conf.getStrings("path.set");
for( String path : pathSet ) {
    System.out.println( path );
}
```

```
path1
path2
path3
path4
```

public Class<?> getClassByName(String name) throws ClassNotFoundException

It attempts to load a class called name by using the JobConf customized class loader. If the class is not found, a ClassNotFoundException is thrown.

By default, the class loader used to load the class is the class loader for the thread that initialized the JobConf object. If that class loader is unavailable, the class loader used to load the Configuration.class is used.

> **Note** This method does not look up the value of name in the configuration; name is the value passed to the class loader.

public Class<?>[] getClasses(String name, Class<?>… defaultValue)

If name is present in the configuration, parses it as a comma-separated list of class names and construct a class object for each entry in the list. If a class cannot be loaded for any entry in the list, a RuntimeException is thrown. If name does not have a value in the configuration, this method returns the array of classes passed in as the defaultValue.

By default, the class loader used to load the class is the class loader for the thread that initialized the JobConf object. If that class loader is unavailable, the class loader used to load the Configuration.class is used.

public Class<?> getClass(String name, Class<?> defaultValue)

If name is present in the configuration, it attempts to load the value as a class using the configuration's class loader. If a value exists in the configuration and a class cannot be loaded for that value, a RuntimeException is thrown. If name does not have a value, the class defaultValue is returned.

By default, the class loader used to load the class is the class loader for the thread that initialized the JobConf object. If that class loader is unavailable, the class loader used to load the Configuration.class is used.

public <U> Class<? extends U> getClass(String name, Class<? extends U> defaultValue, Class<U> xface)

If name is present in the configuration, attempts to load the value as a class using the configuration's class loader. If a value exists in the configuration, and a class cannot be loaded for that value, a RuntimeException is thrown. The loaded class must derive from or implement xface, or else a RuntimeException will be thrown. If no value is present for name in the configuration, the defaultValue will be returned.

This getClass method returns the result of the call theClass.asSubclass(xface); where theClass is the class constructed from the class name stored under `name` or the `defaultValue` if there is no value stored under `name`.

By default, the class loader used to load the class is the class loader for the thread that initialized the `JobConf` object. If that class loader is unavailable, the class loader used to load the `Configuration.class` is used.

public void setClass(String name, Class<?> theClass, Class<?> xface)

The class name for `theClass` is stored in the configuration under the key `name`. If `theClass` does not derive from or implement `xface`, a `RuntimeException` is thrown.

Note The name of the class, `theClass.getName()`, is stored, not a serialized version of the class.

Getters for Localized and Load Balanced Paths

The framework provides the capability for multiple local directories to be specified for task temporary files. Multiple locations are allowed to load balance the I/O over multiple devices. The framework will attempt to select one location set either at random or in sequential order. The ordering used will be given in the method description.

Each of the methods described in this section is called with a trailing path component that includes a final file name. The return value will be the full path to the file; all the intermediate directory components will be constructed if needed.

The first found complete path that can be created or exists will be returned by the method. The intermediate directories may be constructed in locations that do not allow the complete construction of the path. If so, those intermediate directories that have been created will not be removed. If no path can be constructed, an `IOException` will be thrown.

These throwing IOException paths are explicitly constructed on the local (host) file system. See Listing A-10.

Listing A-10: Samples of public Path getLocalPath(String dirsProp, String path) Throwing IOException

```
conf.setStrings("path.set", "dira/a/a/a/","dirb/b/b/b","dirc/c/c/c");
Path random = conf.getLocalPath( "path.set", "trailing/path/file")
```

The path candidates are as follows:

- `dira/a/a/a/trailing/path`
- `dirb/b/b/trailing/path`
- `dirc/c/c/c/trailing/path`

A result in this example might be `dirc/c/c/c/trailing/path/file`.

Note This method leaves partial paths in place that were constructed during its operation. The pseudorandom method does not guarantee that all possible path candidates will be tried; only that no more than the count of path candidate elements will be tried (as of Hadoop 0.19.0). Also as of Hadoop 0.19.0, the method does not fail if the path candidate is a file, not a directory.

public Path getLocalPath(String dirsProp, String pathTrailer) throws IOException

Load balances access to a set of directories that reside on different devices. The goal is to return a resultant path composed of `pathTrailer` as the trailing component and one element out of the set of directories stored under `dirsProp` as the path leader. If `dirsProp` is unset, an `IOException` is thrown. This method uses the Hadoop `LocalFileSystem` object for all path operations. The paths defined by `dirsProp` are searched in a pseudo-random order.

public File getFile(String dirsProp, String pathTrailer) throws IOException

This method is used to load balance access to a set of directories that reside on different devices. The goal is to return a resultant path composed of `pathTrailer` as the trailing component and one element out of the set of directories stored under `dirsProp` as the path leader. If `dirsProp` is unset, an `IOException` is thrown.

This method uses the `java.io.File` methods to create directory paths and test for directory existence. The paths defined by `dirsProp` are searched in a pseudo-random order.

public String[] getLocalDirs() throws IOException

This method looks up the key `mapred.local.dir` in the configuration. The value is expected to be a set of file system paths separated by commas. If present, the value is split on comma characters and the resulting array of `String` objects is returned. If there is no value present, a `null` is returned. This is used by the TaskTracker to find the set of directories to use for per-task local storage. The TaskTracker uses a round robin strategy to allocate a task directory for a new task.

Note The value stored in the configuration under the key `mapred.local.dir` is the set of local file system locations to be used by MapReduce tasks for temporary file storage. This parameter is generally only used directly by the framework.

public void deleteLocalFiles() throws IOException

This method deletes all the directory trees stored in the configuration under the key `mapred.local.dir`. The value is parsed as a comma-separated list of paths. This is used by the framework to clean up the local machine temporary areas on TaskTracker start and TaskTracker exit.

public void deleteLocalFiles(String subdir)throws IOException

This method deletes `subdir` from all the directories that are stored in the configuration under the key `mapred.local.dir`. The value is parsed as a comma-separated list of paths. This is used by the framework to clean up the local machine temporary files for a particular task.

public Path getLocalPath(String pathString) throws IOException

This method looks up the key `mapred.local.dir` in the configuration and parses the value as a comma-separated list of local file system paths. For each directory in the resulting list, an attempt is made, in pseudo-random order, to create the path portion of `pathString`, including any leading directory elements. If after this creation attempt that directory exists, the file name portion of `pathString` is appended to the directory path and the resulting path is returned.

public String getJobLocalDir()

This method looks up the key `job.local.dir` in the configuration and returns the value. The value will be the task-specific shared directory for each job on each TaskTracker. This parameter is set only in the `JobConf` object passed to each task.

The returned value will be the path fragment `taskTracker/jobcache/JobId/work`, prefixed by one of the directories specified in the set of directories stored under the key `mapred.local.dir`.

This is used by the framework when setting up the per-job task environment on a Task-Tracker node. Tasks can use this method to find the path to the task-specific directory on the local file system, which may be used for temporary file storage.

Note In Hadoop 0.19.0, each task for a job on the same TaskTracker may get a distinct local `dir` if multiple directories are specified in the `mapred.local.dir` value.

Methods for Accessing Classpath Resources

The framework provides a way for tasks to access resources from the task-specific classpath objects.

public URL getResource(String name)

Returns the URL for the resource `name`, found by searching the configuration's class loader. If the resource is not found, `null` is returned.

By default, the class loader used to load the class is the class loader for the thread that initialized the `JobConf` object. If that class loader is unavailable, the class loader used to load the `Configuration.class` is used.

Note This method does not look up the value of `name` in the configuration; `name` is the value passed to the class loader.

public InputStream getConfResourceAsInputStream (String name)

Returns the `java.io.InputStream` resulting from opening the URL for the resource `name`, found by searching the configuration's class loader. If the resource is not found, `null` is returned.

By default, the class loader used to load the class is the class loader for the thread that initialized the `JobConf` object. If that class loader is unavailable, the class loader used to load the `Configuration.class` is used.

Caution This method does not look up the value of `name` in the configuration; `name` is the value passed to the class loader.

public Reader getConfResourceAsReader(String name)

Returns the `java.io.Reader` resulting from opening the URL for the resource `name`, found by searching the configuration's class loader. If the resource is not found, `null` is returned.

By default, the class loader used to load the class is the class loader for the thread that initialized the `JobConf` object. If that class loader is unavailable, the class loader used to load the `Configuration.class` is used.

This method does not look up the value of `name` in the configuration. The name is used directly as a Java resource name. This is approximately equivalent to `new InputStreamReader(System.getClassLoader().getResourceAsInputStream(name));` with error checking and using the `ClassLoader` member variable of the configuration.

Caution `getConfResourceAsReader` does not look up the value of `name` in the configuration; `name` is passed directly to the class loader.

Methods for Controlling the Task Classpath

These methods ensure that the objects referenced are distributed to the task nodes and made available in the classpath of the tasks.

public String getJar()

This method is a shortcut for the call `get("mapred.jar")`. The key `mapred.jar` is the JAR to use for the MapReduce job.

The `mapred.jar` key's value is set by the `setJar(String jar)` method, the `setJarByClass(Class cls)` method, and the `JobConf` constructors that take a `Class` value as a parameter.

If the `mapred.jar` key has been set in the configuration, the value will be returned.

public void setJar(String jar)

Stores the `String jar`, which should be the path to the JAR that contains the map and reduce classes for this job into the configuration under the `mapred.jar` key. Any prior value stored under the key `mapred.jar` will be discarded. This archive will be distributed to the task nodes and placed in the classpath for the map and reduce tasks.

public void setJarByClass(Class cls)

Looks for the first JAR file in the classpath that contains class `cls`. If found, stores the path to it under the `mapred.jar` key in the configuration. If no JAR file is found that contains `cls`, a `RuntimeException` is thrown.

Note This method will look only in JAR files, not in zip files or in directory trees.

Methods for Controlling the Task Execution Environment

These methods control the setup and cleanup of the individual task environment.

public String getUser()

This method returns the value stored in the configuration under the key `user.name`. This parameter is generally initialized to the name of the user that launched the job, but this is not enforced.

> **Caution** Usernames may be overwritten with a different username by any user. This is not a security feature, and Hadoop permissions are not a security feature. Through at least Hadoop 0.19 any user may claim to be any other Hadoop user and act fully as if they are that user, including the removal of files or the scheduling of jobs. There is way to prevent this; you have to trust the users who have access to your cluster because any user can override any Hadoop level permission restrictions placed on that user.

public void setUser(String user)

The value of `user` is stored in the configuration under the key `user.name`. This value is used for HDFS permission checking.

public void setKeepFailedTaskFiles(boolean keep)

Stores the value of `keep` in the configuration under the key `keep.failed.task.files`. This value configures the framework to save or not save the intermediate output files of tasks that fail. It is set to `true` to when the task output is needed to debug a failing job.

public boolean getKeepFailedTaskFiles()

Returns the value stored in the configuration under `keep keep.failed.task.files` converted to a `boolean`. If no value is found, or the value is not exactly `true` or `false`, the value `false` is returned.

public void setKeepTaskFilesPattern(String pattern)

Stores a Java regular expression `String pattern` into the configuration under the key `keep.task.files.pattern`. If the task id of a task matches this regular expression, its temporary files will not be removed The file names are written as `*_[mr]_[jobid]_[tasknumber]`. The job id is 0-padded on the left. The pattern `*_m_000027_5` would match the fifth map task of job `00027`. The pattern `*_r_000027_5` would match the fifth reduce task of job `00027`.

This is used to aid in debugging the framework.

public String getKeepTaskFilesPattern()

Returns the value stored in the configuration under the key `keep.task.files.pattern`. The framework calls this in the TaskTracker before cleaning up temporary files after a task completes. If the task id matches the pattern, the temporary files are not removed. This is a framework debugging aid.

public void setWorkingDirectory(Path dir)

This method builds a path by concatenating `dir` with the value of `getWorkingDirectory()` and stores that value in the configuration under the key `mapred.working.dir`. The user calls it and if it has not been called by job submission time, the `JobClient` object will initialize it to the current working directory of the process submitting the job.

> **Note** If the working directory has not been initialized by the time this method is called, the default working directory for the default file system will be used. For HDFS, it is generally `/user/USERNAME`. The default file system is the file system defined by the configuration key `fs.default.name`.

public Path getWorkingDirectory()

Returns the value stored in the configuration under `mapred.working.dir`. If this value is unset, this method first sets the value for the key to the default working directory for the default file system.

The default file system is defined by the configuration parameter `fs.default.name`; for HDFS, the default working directory is `/user/USERNAME`.

public void setNumTasksToExecutePerJvm(int numTasks)

Prior to Hadoop 0.19.0, a new JVM was created for each task run by the TaskTracker. As of Hadoop 0.19.0, the TaskTracker has the capability to reuse the task JVM for additional tasks. The configuration key `mapred.job.reuse.jvm.num.tasks`'s value is the number of times that a JVM may be reused. This method stores `numTasks` in the configuration under the key `mapred.job.reuse.jvm.num.tasks`.

Note Calling `setNumTasksToExecutePerJvm` with a value that is `<= 0` will result in erroneous behavior.

public int getNumTasksToExecutePerJvm()

Looks up the value of `mapred.job.reuse.jvm.num.tasks` in the configuration and converts the value to an integer. If the value does not exist or if the value cannot be converted to an integer, it returns `1`.

Prior to Hadoop 0.19.0, a new JVM was created for each task run by the TaskTracker. As of Hadoop 0.19.0, the TaskTracker has the capability to reuse the task JVM for additional tasks. The value stored in the configuration under `mapred.job.reuse.jvm.num.tasks` is the number of times to use a JVM for a task.

Methods for Controlling the Input and Output of the Job

The methods described in this section are used to configure how the jobs' input and output will be handled. This includes how the input is parsed and presented to the framework, the compression of intermediate and final output, and how the output is written.

public InputFormat getInputFormat()

This method looks up the value of the key `mapred.input.format.class` in the configuration and instantiates a class of that name. If the value is missing, a `TextInputFormat.class` will be returned. If the class name cannot be instantiated, or if the instantiated class is not an instance of `InputFormat`, a `RuntimeException` will be thrown.

The returned class will be used by the framework to read the input data set for the job. The key/value pairs that the class extracts from the input will be passed to the map method of the mapper class in the map tasks. There will be one instance created per map task, and that instance will receive the input split for that map task as input.

public void setInputFormat(Class<? extends InputFormat> theClass)

This method stores the class name of `theClass` in the configuration under the key `mapred.input.format.class`. An instance of this class will be instantiated in each map task to convert the input split data into a set of key/value pairs for the map method of the mapper class. If `theClass` does not implement the interface `InputFormat`, a `RuntimeException` will be thrown.

public OutputFormat getOutputFormat()

This method looks up the value of the key `mapred.output.format.class` in the configuration and instantiates a class of that name. If the value is missing, a `TextOutputFormat.class` will be returned. If the class name cannot be instantiated or if the instantiated class is not an instance of `OutputFormat`, a `RuntimeException` will be thrown.

The returned class will be used by the framework to write each of the key/value pairs output by the `reduce()` method of the reducer class, and if not explicitly configured, each of the key/value pairs output by the map method of the mapper class.

There will be one instance of this class created for each reduce task. By default, one instance of this class is created for each map task.

public void setOutputFormat(Class<? extends OutputFormat> theClass)

This method stores the class name of `theClass` in the configuration under the key `mapred.output.format.class`. This class transforms the key/value pairs passed to the output in the `reduce()` method of the reducer into the output format. The default value is `TextOutputFormat`. If `theClass` does not implement the `OutputFormat` interface, a `RuntimeException` will be thrown.

public OutputCommitter getOutputCommitter()

The framework provides a unique output directory for each task and stores this directory in the per-task configuration under the key `mapred.output.dir`. As of Hadoop 0.18, this key is set via the `FileOutputFormat.setOutputPath` static method.

As of 0.19.0, the `OutputCommitter` object is used to process the files in the per-task temporary area on successful task completion, and is responsible for deciding which output files are moved to the actual output area. Prior to this, any files present were moved to the userspecified output path.

This method retrieves the value stored in the configuration under the key `mapred.output.committer.class`. If the retrieved value is `null`, the `FileOutputCommitter` class will be returned. If the retrieved value is not `null`, the method will attempt to instantiate a class using the value as the class name. If the class name cannot be instantiated or if the instantiated class is not derived from the class `OutputCommitter`, a `RuntimeException` will be thrown.

public void setOutputCommitter(Class <? extends OutputCommitter> theClass)

This method will store the class name of `theClass` in the configuration under the key `mapred.output.committer.class`. If `theClass` does not implement the `OutputCommitter` interface, a `RuntimeException` will be thrown. See `getOutputCommitter` for a description of what the `OutputCommitter` is used for.

public void setCompressMapOutput(boolean compress)

This method stores the `String` equivalent of the value of `compress` in the configuration under the key `mapred.compress.map.output`. If the stored value is `true`, the map output data that will be consumed by the reduce phase will be compressed, using either the default compression codec or the codec specified by the method `setMapOutputCompressorClass`.

Map Task Output Compression

During a Hadoop job that has a reduce phase, the map phase produces intermediate output that will be further processed by the framework. This output will eventually become the input to the reduce phase. This output may be compressed to reduce transitory disk space requirements and network transfer requirements. The call `setCompressMapOutput(true)` will enable this compression. To enable map output compression when the job will not have a reduce phase, the call `FileOutputFormat.setCompressOutput(conf, true)` must be made.

Having the map output compressed can save substantial time because the amount of data that must traverse the network between the map and the reduce phase may be substantially reduced.

Having the job output compressed may also save substantial time because the amount of data to be stored in HDFS may be substantially reduced, greatly reducing the amount of network traffic for the replicas.

---

public boolean getCompressMapOutput()

This method returns the value stored in the configuration under the key `mapred.compress.map.output`. If the value is unset or is not one of `true` or `false`, the value `false` will be returned. If this value is `true`, map task output that will be reduced will be compressed using the compression defined for `SequenceFiles`.

public void setMapOutputCompressorClass(Class <? extends CompressionCodec> codecClass)

This method stores the class name of `codecClass` in the configuration under the key `mapred.map.output.compression.codec`. An instance of this class will be used to compress the map task output that is to be passed to the reduce tasks if the configuration key `mapred.compress.map.output` has the value of `true`. This key may be set by the `JobConf` method `setCompressMapOutput(boolean)`. If `codecClass` does not implement the `CompressionCodec` interface, a `RuntimeException` will be thrown.

public Class<? extends CompressionCodec> getMapOutp utCompressorClass(Class<? extends CompressionCodec> defaultValue)

This method looks at the value of the key `mapred.output.compression.codec` in the configuration. If the value is not found, `defaultValue` is returned. If the value cannot be instantiated as a class that is derived from `CompressionCodec`, a `RuntimeException` will be thrown.

Output Key and Value Classes

The Hadoop framework is responsible for loading the job input and converting that input into key/value pairs that are passed to the map method of the mapper, passing the key/value pairs output by the map method of the mapper to the `reduce()` method of the reducer, and taking the key/value pairs output by the `reduce()` method of the reducer and writing them to the job output.

The class that loads and transforms the input into key/value pairs is derived from `InputFormat` and requires that the type of the key and the type of the value be specified. The class that handles loading the input is responsible for producing keys and values of the correct type. A commonly used class is the `KeyValueTextInput` class, which parses the input as text files, with each record on a single line and the key and value separated by the first tab character. The key type is `org.apache.hadoop.io.Text`, and the value type is `org.apache.hadoop.io.Text`. If the job does not explicitly configure the map output class as `org.apache.hadoop.io.Text` or the job output class as `org.apache.hadoop.io.Text`, the reduce will fail with a key type mismatch error.

An `InputFormat` object has an associated `RecordReader` object. The `RecordReader` must provide `createKey` and `createValue` objects. The types of these objects will be used to define the mapper class input key and value types.

The class to receive and transform the output key/value pairs is derived from `OutputFormat`. The default value for the key class is `LongWritable`, and the default value for the value class is `org.apache.hadoop.io.Text`. The job may specify different classes via the `setOutputKeyClass` and `setOutputValueClass` methods, respectively. By default, the expected map output types are the same as the expected reduce input and output types.

The job may specify that the map output key type and or the map output value type is different from the job output key and value type. The `setMapOutputKeyClass` method allows the job to specify the map output key class and the reduce input key class as being different from the job output key class. The `setMapOutputValueClass` method allows the job to specify the map output value class and reduce input value class as being different that the job output key class.

The class specified under the key `map.sort.class` in the configuration will be used to sort the key objects if a reduce has been requested by the job. The default value for this key is `org.apache.hadoop.util.QuickSort`, an implementation of `org.apache.hadoop.util.IndexedSorter`.

The key and value classes can be any type as long as the framework is provided with serializer classes and deserializer classes that implement `org.apache.hadoop.io.serializer.Serializer` and `org.apache.hadoop.io.serializer.Deserializer`, and the class names are added to the list stored in the configuration under the key `io.serialization`.

---

public void setMapOutputKeyClass(Class<?> theClass)

This method stores the class name of `theClass` in the configuration under the key `mapred.mapoutput.key.class`. The type of this class will be used as the type of the map method output keys and the `Reducer.reduce()` method input keys. This class must be serializable and deserializable by a class defined in the list of serializers specified in the value of the configuration key `io.serializations`. `theClass` must be sortable by the class returned by `getOutputKeyComparator()`.

public Class<?> getMapOutputKeyClass()

This method looks up the value of the key `mapred.mapoutput.key.class` in the configuration. If the value is unset, `null` is returned. If the value cannot be instantiated as a class, a `RuntimeException` is thrown. This class will also be the `Reducer.reduce()` method input key class. The default class for this is the job output key class, `getOutputKeyClass(),` and the default for it is `LongWritable`.

public Class<?> getMapOutputValueClass()

This method looks up the value of the key `mapred.mapoutput.value.class` in the configuration. If the value is unset, the value of `getOutputValueClass()` is returned. If the value cannot be instantiated as a class, a `RuntimeException` is returned. This class will also be the `reduce()` method input value class. The default value for the output value class is `org.apache.hadoop.io.Text`.

public void setMapOutputValueClass(Class<?> theClass)

This method stores the class name of `theClass` in the configuration under the key `mapred.mapoutput.value.class`, which will be the class for the `Mapper.map()` value output and the `Reducer.reduce ()`values. The default is the type used for the reduce output value, `org.apache.hadoop.io.Text`. This class must be serializable by a class defined in the list of serializers specified in the value of the configuration key `io.serializations`.

public Class<?> getOutputKeyClass()

This method looks up the key `mapred.output.key.class` in the configuration. If the value is unset, the class object `org.apache.hadoop.io.LongWritable` will be returned. If the value is set and a class of that name cannot be instantiated, a `RuntimeException` will be thrown. This class is the key class that the `Reducer.reduce()` method will output.

public void setOutputKeyClass(Class<?> theClass)

This method stores the name of `theClass` in the configuration under the key `mapred.output.key`. This will be the type of key output by the `Reducer.reduce()` method. Unless overridden by `setMapOutputKeyClass`, `theClass` will also be the `Mapper.map()` output key. `theClass` must be sortable by the class returned by `getOutputKeyComparator()` if it will also be used as the `Mapper.map()` output key class. `theClass` class must be serializable by a class defined in the list of serializers specified in the value of the configuration key `io.serializations`. The default value is `org.apache.hadoop.io.LongWritable`.

public Class<?> getOutputValueClass()

This method looks up the value of the key `mapred.output.value.class` in the configuration. If the value is unset, the class `org.apache.hadoop.io.Text` is returned. The value is instantiated as a class, and the class is returned. If the value cannot be instantiated as a class, a `RuntimeException` will be thrown.

public void setOutputValueClass(Class<?> theClass)

This method stores the name of `theClass` in the configuration under the key `mapred.output.value.class`. This value will be used as the type of the `Reducer.reduce()` output value; if not overridden by `setMapOutputValueClass()`, it will be the type of the `Mapper.map()` output value. If this class is used as a map output value, it must be serializable by a class defined in the list of serializers specified in the value of the configuration key `io.serializations`.

Methods for Controlling Output Partitioning and Sorting for the Reduce

The partitioner determines which key/value pair is sent to which reduce task. The comparator, the class returned by `getOutputKeyComparator()`, determines the ordering of the key/ value pairs, and the class returned by `getOutputValueGroupingComparator()` determines which adjacently sorted keys are considered equal for producing a value group to pass to the `Reducer.reduce()` method. Classes used as comparators must implement the `RawComparator` interface.

Defining Optimized Comparators

A class used as a key object in Hadoop may define an optimized comparator class. The comparator has to implement the `org.apache.hadoop.io.WritableComparable` interface. The comparator must be registered with the framework by calling `org.apache.hadoop.io.WritableComparator.define(Key.class, ComparatorInstance)`. The common key class `org.apache.hadoop.io.Text` defines a custom comparator that does a byte-wise comparison of the actual serialized text. This avoids having to deserialize the `Text` object and then run `String` comparisons on the data in the reconstituted objects.

---

public RawComparator getOutputKeyComparator()

This method looks up the value of the key `mapred.output.key.comparator.class` in the configuration. If the value is unset, the class `org.apache.hadoop.io.WritableComparable` `WritableComparator` will be returned. If the value cannot be instantiated as a class that is an instance of `org.apache.hadoop.io.RawComparator`, a `RuntimeException` will be thrown.

public void setOutputKeyComparatorClass(Class <? extends RawComparator> theClass)

This method stores the class name of `theClass` in the configuration under the key `mapred.output.key.comparator.class`. `theClass` will be used to order the keys being presented to the `Reducer.reduce()` method. The default class is the comparator for the `Mapper.map()` key output class. If *`theClass`* does not implement the `RawComparator` interface, a `RuntimeException` will be thrown.

public void setKeyFieldComparatorOptions(String keySpec)

This method stores the `String` *`keySpec`* in the configuration under the key `mapred.text.key.comparator.options`. This method also changes the `OutputKeyComparatorClass` (key `mapred.output.key.comparator.class`) to the class `org.apache.hadoop.mapred.lib.KeyFieldBasedComparator`.

The key fields are separated by the character that is the value of the configuration key `map.output.key.field.separator`. If there is no value set for the key `map.output.key.field.separator`, the separator character will be the ASCII tab character.

The key will be split on `map.output.key.field.separarator` characters into pieces. These pieces are numbered from 1.

The *`keySpec`* `String` is composed of one or more space-separated groups. Each group defines the following items:

- The piece number to start the comparison region.
- The character number in the piece to start the comparison. The first character is `1`; the last character is `0`. This is optional and defaults to position `1`.
- How to sort, either numerically via the `n` option or in reverse order via the `r` option.This is optional and defaults to the standard `String` comparison ordering.
- The piece number to end the comparison region. This is optional and defaults to the starting piece number.
- The character number in the piece to end the comparison. This is optional and defaults to the last character in the `String`: `0`.

The specification `-k1.5nr` specifies numeric reverse-order sorting using the characters from position 5 through the end of the first piece of the key.

The specification `-k2.2,3.4r` specifies reverse `String` comparison using the characters from character 2 in key 2 through to character 4 in piece 3.

Given the line `01234 6789`, key piece `1` would be `01234`, and key piece 2 would be `6789`. The key spec `-k1.2,2.3` would provide a comparison segment of `1234 678`. There is a test class for these key fields in the examples that can be run by giving it three arguments: the key, the key spec for the combiner, and the key spec for the partitioner. The field separator is hard coded as a space: `bin/hadoop jar '\Documents and Settings\Jason\My Documents\HadoopSource\hadoop-0.19.0\hadoopprobook.jar' com.apress.hadoopbook.examples.jobconf.KeyFieldDemonstrator "01234 6789 abcd" "-K1,2" "-k3,3"`. The summarized output is `Partitioner[key: (abc)] Comparator[key: (01234 6789 abcd), key: (01234 6789 abcd)]`.

Note Changing the output key comparator class via `setOutputKeyComparatorClass` disables field-based key comparisons. The output key comparator class must be `org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner` or a functional equivalent.

public String getKeyFieldComparatorOption()

This method looks up the value of the key `mapred.text.key.comparator.options` and returns the value. Please see `setKeyFieldComparatorOption` for a discussion of the appropriate values.

Partitioning

When a job is configured to have a reduce phase, the output will be split into partitions (one partition per reduce task). The framework has a default partitioning strategy of using the hash code of the key, modulus the number of partitions, `key.hashCode() % conf. getNumReduceTasks()`. If your job has three reduces specified, the default partition for a key will be `key.hashCode() % 3`. The user is free to specify a custom partitioning class. The framework provides three partitioning classes:

- `org.apache.hadoop.mapred.lib.HashPartitioner`: default partition based on the key's hash code
- `org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner`: partition based on a segment of the key
- `org.apache.hadoop.mapred.lib.TotalOrderPartitioner`: partition by absolute range of the keys

A custom partitioning class must implement the interface `org.apache.hadoop.mapred.Partitioner`.

---

public Class<? extends Partitioner> getPartitionerClass()

This method looks up the value of the key `mapred.partitioner.class` in the configuration. If the value is unset, the class `org.apache.hadoop.mapred.lib.HashPartitioner` is returned. If the value is set, it is instantiated as a class that must be an instance of `org.apache.hadoop.mapred.Partitioner.class`. If the value cannot be instantiated or is not an instance of the `Paritioner` class, a `RuntimeException` will be thrown. `HashPartitioner` simply uses the hash value of the key, modulus the number of reduce tasks, to determine which reduce will receive any given key/value pair.

public void setPartitionerClass(Class<? extends Partitioner> theClass)

This method stores the class name of `theClass` in the configuration under the key `mapred.partitioner.class`. An instance of this class will be created for each map task and used to determine which reduce will receive which key/value pair that the `Mapper.map()` method outputs. If `theClass` does not implement the `org.apache.hadoop.mapred.Partitioner` interface, `RuntimeException` will be thrown.

public void setKeyFieldPartitionerOptions(String keySpec)

This method stores the `String keySpec` in the configuration under the key `mapred.text.key.partitioner.options`. The output partitioning class will also be set to `org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner` via a call to the `setPartitionerClass()` method.

The portion of the key selected will be hashed, and that hash modulus the number of reduces will be the partition number.

The `keySpec String` is composed of one or more space-separated groups. Each group defines the following items:

- The piece number to start the comparison region.
- The character number in the piece to start the comparison: `1` is the first character; `0` is the last character. This is optional and defaults to position `1`.
- How to sort, either numerically via the n option and or in reverse order via the `r` option. This is optional and defaults to the standard `String` comparison ordering.

- The piece number to end the comparison region. This is optional and defaults to the starting piece number.
- The character number in the piece to end the comparison. This is optional and defaults to the last character in the `String`: `0`.

Note The key parser (at least through Hadoop 0.19.0) has an issue: it doesn't understand that the last piece of the key might not have a separator character after it. If your job generates `ArrayIndexOutOfBounds` exceptions, explicitly end the key piece selection for the second key piece: `-k2. 2` explicitly ends the piece at the last character; `-k2` includes the second key piece and the separator after piece 2.

The specification `-k1.5nr` specifies numeric reverse order sorting using the characters from position 5 through the end of the first piece of the key.

The specification `-k2.2,3.4r` specifies reverse `String` comparison using the characters from character 2 in key 2 to character 4 in piece 3.

Given the line `01234 6789`, key piece `1` would be `01234`, and key piece `2` would be `6789`. The key spec `-k1.2,2,3` would provide a comparison segment of `234 678`.

public String getKeyFieldPartitionerOption()

This looks up the key `mapred.text.key.partitioner.options` in the configuration and returns the value. For this value to have an effect, the output partitioner class must be `org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner`. See `setKeyFieldPartionerOptions` for a description of the returned value.

Output Value Grouping

It is often the case that there is a requirement for grouping output data. Hadoop Core provides a way to group output values that acts very much like a secondary sort on the key data. For this to work in the manner that the user expects, the output partitioner, the output comparator, and the output grouping comparator have to cooperate.

The `outputKeyComparator` must order the keys using the primary and secondary sort. Because keys that must group together may not be equal in this method, the `outputPartitioner` has to be able to place keys that must group together into the same partition. The `outputValueGroupingComparator` must return equality only for those keys that are equal in the primary sort. This will result in a call to the `Reduce.reducer` method for each group of keys.

---

public RawComparator getOutputValueGroupingComparator()

This method looks up the value of the key `mapred.output.value.groupfn.class` in the configuration and attempts to instantiate a class that is an instance of `org.apache.hadoop. io.RawComparator`. If the value is unset, the comparator class for the `Map` key class is returned. If the value cannot be instantiated or the resulting class does not implement `org.apache.hadoop.io.RawComparator`, a `RuntimeException` is thrown.

public void setOutputValueGroupingComparator(Class <? extends RawComparator> theClass)

This method stores the class name of `theClass` in the configuration under the key `mapred.output.value.groupfn.class`. If `theClass` does not implement the `org.apache.hadoop.io.RawComparator` interface, a `RuntimeException` will be thrown.

The use of this method enables a grouping operator on keys and a secondary sort. The user must set both a partitioner and a comparator that cooperate for this to be used. It is common for the default output comparator to be used to force complete sorting of the keys output by the `Mapper.map()` method. The output comparator must compare keys so all keys that are to be grouped together are adjacent in the sort. The partitioner must ensure that all keys that are to be grouped together are sent to the same partition.

The `Reducer.reduce()` method will receive the first key in the group, and the values will be the values from all adjacent keys that the output value grouping comparator considers equal. If keys are of the form `item rank` and the values are of the form `data`, the partitioner must use only `item` to partition. The standard output comparator will sort lexically on `item` `rank`. The output value grouping operator will use only `item` for comparing keys. The `Reducer.reduce()` method will receive all keys that share `item`, and the values will be lexically sorted by `rank`.

The keys are composed of `item rank`, where the `item` is one of Key1 or Key2, and the `rank` is one of 00, 01, 02. The partitioner would use the `item` for partitioning. The output comparator would fully sort the keys by `item rank`. The output value grouping comparator would use only `item` for comparing keys. (See Table A-1 and Table A-2.)

Table A-1: Sample Input

Key	Value	Partitioner Value	Output Comparator Sort
Key1 00	00	Key1	Key1 00
Key1 01	01	Key1	Key1 01
Key1 02	02	Key1	Key1 02
Key2 00	00	Key2	Key2 00
Key2 01	01	Key2	Key2 01
Key2 02	02	Key2	Key2 02

Table A-2: Reducer.reduce Calls

Key	Values		
Key1 00	00	01	02
Key2 00	00	01	02

Methods that Control Map and Reduce Tasks

These methods actually specify the class that provides the `Mapper.map()` and `Reducer.reduce()` methods. They specify if the map methods may be run from multiple threads or in a single thread.

They specify if the framework will attempt to run multiple instances of a task to see if one will run faster, and when to consider a task completely failed and a job completely failed.

Single Threaded or Multi-Threaded Mappers

The framework creates an instance of the mapper class in each map task. By default, a single-threaded map runner is used, and the key/value pairs are passed to the `Mapper.map()` method serially. The user may inform the framework that multiple threads are to run the `Mapper.map()` method. There will be multiple simultaneous calls to the `map()` method of the single instance of the Mapper class, running in the JVM that hosts the map task. The input of key/value pairs are treated as a queue, being serviced by a thread pool, which invokes the `Mapper.map()` method on each pair pulled from the queue.

The user specifies this behavior by setting the map runner class to `org.apache.hadoop.mapred.lib.MultithreadedMapRunner` and by storing the number of threads to run in the configuration under the key `mapred.map.multithreadedrunner.threads`.

---

public Class<? extends Mapper> getMapperClass()

This method looks up the value of the key `mapred.mapper.class` in the configuration and attempts to instantiate the value as a class of type `org.apache.hadoop.mapred.Mapper`. If the value is unset, the class `org.apache.hadoop.mapred.lib.IdentityMapper` is returned. If the value cannot be instantiated as a class of the correct type, a `RuntimeException` is thrown.

The returned class will provide the map method that all the input data will be passed through.

public void setMapperClass(Class<? extends Mapper> theClass)

This method stores the name of `theClass` class in the configuration under the key `mapred.mapper.class`. An instance of this class will be created in each map task, and each input key/value pair will be passed to `theClass` map method. If `theClass` does not implement the `org.apache.hadoop.mapred Mapper` interface, a `RuntimeException` will be thrown.

public Class<? extends MapRunnable> getMapRunnerClass()

This method looks up the key `mapred.map.runner.class` in the configuration and instantiates the value as a class of type `org.apache.hadoop.mapred.MapRunnable`. If the value is unset, the class `org.apache.hadoop.mapred.lib.MapRunnable` is returned.

public void setMapRunnerClass(Class<? extends MapRunnable> theClass)

This method stores the name of `theClass` in the configuration under the key `mapred.map.runner.class`. This is commonly used when the `Mapper.map()` method is to be threaded, and `theClass` in this case is `org.apache.hadoop.mapred.lib.MultithreadedMapRunner.class`. When this is done, there is usually a `setInt("mapred.map.multithreadedrunner.threads",threadCount)` call.

The multithreaded map runner is very handy when the map method is not blocked waiting on local CPU or IO, such as when the map method is used to fetch URLs.

public Class<? extends Reducer> getReducerClass()

This method looks up the key `mapred.reducer.class` in the configuration and instantiates the value as a class of type `org.apache.hadoop.mapred.Reducer`. If the value is unset, the class `org.apache.hadoop.mapred.lib.IdentityReducer` is returned. If the value cannot be instantiated as a class of the correct type, a `RuntimeException` will be thrown.

public void setReducerClass(Class<? extends Reducer> theClass)

This method stores the name of `theClass` in the configuration under the key `mapred.reducer.class`. If `theClass` does not implement the `Reducer` interface, a `RuntimeException` will be thrown.

One instance of this class will be created in each reduce task. Each unique key will be passed to one instance of the `Reducer.reduce()` method of `theClass`, with all the values that share that key.

Combiners: A Way to Reduce Intermediate Data

A combiner class is a minireducer that is run in the context of the map task to pregroup key/value pairs that share a key.

Combiners can greatly minimize the amount of output that has to pass between the map and reduce tasks and speed up the job.

The class used for combining must implement the `Reducer` interface, and the class's `reduce()` method will be called to combine map output values that share a key.

If the job's reducer class is being used as a combiner, `reduce()` must not have side effects because there is no constraint on the number of times the `reduce()` method will be called in as a map output combiner. In particular, if the same class is used for combing and reducing, unless care is taken to change the counter names, the counts displayed at job end will be the sum of the combiner and reducer counts. Please see `com.apress.hadoopbook.examples.ch5.CounterExamplesWithCombiner`, and look at the `NaiveReducer` counter values and compare them against the `reducer` and `combiner` counter values.

---

public Class<? extends Reducer> getCombinerClass()

This method looks up the key `mapred.combiner.class` in the configuration and instantiates the value as a class implementing the `Reducer` interface. If the value is unset, `null` is returned. If the value cannot be instantiated as a class implementing the `Reducer` interface, a `RuntimeException` is thrown.

public void setCombinerClass(Class<? extends Reducer> theClass)

This method stores the name of `theClass` in the configuration under the key `mapred.combiner.class`. If `theClass` does not implement the `Reducer` interface, a `RuntimeException` is thrown.

Speculative Execution: Friend and Foe

Hadoop has its roots in clusters of heterogeneous machines. In this environment, the amount of wall clock time for any given machine to execute a map or reduce task could vary widely because of differing machine capabilities. In addition, there is no guarantee that any given `InputSplit` will take the same amount of wall clock time to execute.

Speculative execution informs the cluster that any unused task slots may be used to run duplicate instances of an already running task. The first of these duplicates to complete has its results used, and the other task has its output discarded.

If your tasks do not have side effects that Hadoop cannot undo, do not consume resources with some real costs or load your machines so that other tasks run slower. Speculative execution is your friend.

Note Hadoop only knows how to discard task output that is in the form of job counters or output that is placed in the per-task output directory. Ensure that speculative execution is disabled if your tasks have output that Hadoop cannot discard or side effects that Hadoop cannot undo.

public boolean getSpeculativeExecution()

This method returns `true` if either `getMapSpeculativeExecution()` or `getReduceSpeculativeExecution()` is `true`. The default Hadoop configuration has speculative execution enabled for map tasks and for reduce tasks.

public void setSpeculativeExecution (boolean speculativeExecution)

This method calls `setMapSpeculativeExecution(speculativeExecution)` and `setReduceSpeculativeExecution(speculativeExecution)`. If `speculativeExecution` is `true`, speculative execution will be enabled for both map and reduce tasks. If `speculativeExecution` is `false`, speculative execution will be disabled for both map and reduce tasks.

public boolean getMapSpeculativeExecution()

This method looks up the value of the key `mapred.map.tasks.speculative.execution` in the configuration and converts that value to a `boolean` value, which is then returned. If the value is unset, `true` is returned. If the value is not the `String true`, `false` is returned.

public void setMapSpeculativeExecution (boolean speculativeExecution)

This method stores the `String` value of the `boolean speculativeExecution` in the configuration under the key `map.tasks.speculative.execution`.

public boolean getReduceSpeculativeExecution()

This method looks up the value of the key `mapred.reduce.tasks.speculative.execution` in the configuration and converts that value to a `boolean` value, which is then returned. If the value is unset, `true` is returned. If the value is not the `String true`, `false` is returned.

public void setReduceSpeculativeExecution (boolean speculativeExecution)

This method stores the `String` value of the `boolean speculativeExecution` in the configuration under the key `mapred.reduce.tasks.speculative.execution`.

public int getNumMapTasks()

This method looks up the value of the key `mapred.map.tasks` in the configuration and returns the value converted to an `int`. If the value is unset, `1` is returned. If the value cannot be converted to an `int`, a `NumberFormatException` is thrown. This value is the suggested number of map tasks to run. The actual number of map tasks will be determined by the number of `InputSplits` that the framework constructs from the input data. In general, there is at least one `InputSplit` for each input file. The input format might be able to make multiple `InputSplits` from a single file. The `FileInputFormat` set of input formats will split uncompressed files on HDFS block boundaries, which by default are 64MB. Many installations increase this size to 128MB or higher.

public void setNumMapTasks(int n)

This stores the `String` representation of `n` in the configuration under the key `mapred.map.tasks`. The input format will attempt to ensure that this is the maximum number of map tasks, but may not be able to do so if there are more individual files that this in the input directory. In general, tuning this and the split size `setInt("mapred.min.split.size ", NUMBER)`, so map tasks take more than a minute to run is considered optimal.

public int getNumReduceTasks()

This method looks up the value of the key `mapred.reduce.tasks` in the configuration and returns the value converted to an `int`. If the value is unset, `1` is returned. If the value cannot be converted to an `int`, a `NumberFormatException` is thrown.

Unlike the number of map tasks, this is exactly the number of reduce tasks that will be run.

public void setNumReduceTasks(int n)

This method stores the `String` representation of `n` in the configuration under the key `mapred.reduce.tasks`. Exactly this number of reduce tasks will be run by the framework. If this number is `0`, no reduce tasks will be run, and no output partitioning or sorting will be done. There will be one output file per map task, written to the output directory configured for the job.

public int getMaxMapAttempts()

This method looks up the value of the key `mapred.map.max.attempts` in the configuration and returns the value converted to an `int`. If the value is unset, the value `4` is returned. If the value cannot be converted to an `int`, a `NumberFormatException` is thrown.

The framework will attempt to reschedule map tasks that fail up to `getMaxMapAttempts()` times before the job is considered failed.

public void setMaxMapAttempts(int n)

This method stores the `String` representation of `n` in the configuration under the key `mapred.map.max.attempts`. This is rarely changed by the user other than to set it to `0` to disable the retrying of failed jobs.

The framework will attempt to reschedule map tasks that fail up to `getMaxMapAttempts()` times before the job is considered failed.

public int getMaxReduceAttempts()

This method looks up the value of key `mapred.reduce.max.attempts` in the configuration and returns the value converted to an `int`. If the value is unset, the value `4` is returned. If the value cannot be converted to an `int`, a `NumberFormatException` is thrown.

The framework will attempt to reschedule reduce tasks that fail up to this value times before the job is considered failed.

public void setMaxReduceAttempts(int n)

This method stores the `String` representation of `n` in the configuration under the key `mapred.reduce.max.attempts`. This is rarely changed by the user other than to set it to `0` to disable the retrying of failed jobs.

The framework will attempt to reschedule reduce tasks that fail up to this value times before the job is considered failed.

public void setMaxTaskFailuresPerTracker(int noFailures)

This method stores the `String` representation of `noFailures` in the configuration under the key `mapred.max.tracker.failures`. This value is the number of tasks for this job that may fail on a specific TaskTracker before that TaskTracker is considered failed for this job.

public int getMaxTaskFailuresPerTracker()

This method looks up the value of the key `mapred.max.tracker.failures` in the configuration and returns the value converted to an `int`. If the value is unset, the value 4 is returned. If the value cannot be converted to an `int`, a `NumberFormatException` will be thrown. This value is the number of tasks for this job that may fail on a specific TaskTracker before that TaskTracker is considered, failed, for this job.

public int getMaxMapTaskFailuresPercent()

This method looks up the value of the key `mapred.max.map.failures.percent` in the configuration. If the value is unset, `0` is returned. If the value cannot be converted to an `int`, a `NumberFormatException` is thrown.

If this value is not zero, a job may succeed if less than this value as a percentage of the map tasks cannot be successfully completed. So if the job has 100 map tasks, and this returns 1, only 99 of the map tasks have to complete successfully for the job to be considered a success.

Map tasks that do not succeed are retried up to `getMaxMapAttempts()` times before being considered failed.

public void setMaxMapTaskFailuresPercent(int percent)

This method stores the `String` representation of `percent` in the configuration under the key `mapred.max.map.failures.percent`. This is the percentage of map tasks that can fail without the job being marked as a failure. The default value for this parameter is `0`.

A map task that does not succeed is retried `getMaxMapAttempts()` times, which defaults to `4`, before being that task is considered failed.

public int getMaxReduceTaskFailuresPercent()

This method looks up the value of the key `mapred.max.reduce.failures.percent` in the configuration. If the value is unset, `0` is returned. If the value cannot be converted to an `int`, a `NumberFormatException` is thrown.

If this value is not zero, a job may succeed if less than this value as a percentage of the reduce tasks cannot be completed successfully. So if the job has 10 reduce tasks, and this returns `10`, only 9 of the reduce tasks have to complete successfully for the job to be considered a success.

Map tasks that do not succeed are retried up to `getMaxReduceAttempts()` times before being considered failed.

public void setMaxReduceTaskFailuresPercent(int percent)

This method stores the `String` representation of `percentage` in the configuration under the key `mapred.max.reduce.failures.percent`. This is the percentage of reduce tasks that can fail without the job being marked as a failure. The default value for this parameter is `0`.

A reduce task that does not succeed is retried `getMaxReduceAttempts()` times, which defaults to 4 before being that task is considered failed.

Methods Providing Control Over Job Execution and Naming

These methods provide a way to specify a job name and a session identifier as well as to specify a priority for a job. The naming is also helpful for distinguishing jobs in the reporting frameworks.

They also provide a way to enable profiling of specific tasks and of running a debugging script on failed tasks.

public String getJobName()

This method looks up the value of the key `mapred.job.name` in the configuration and returns the result. If the value is unset, an empty `String` is returned.

This is the name that the job will be identified by to the user.

public void setJobName(String name)

This method stores `name` in the configuration under the key `mapred.job.name`. `name` will be used to identify the job in user-reporting mechanisms.

Hadoop on Demand

Hadoop On Demand (HOD) is a package that provides virtual map/red clusters on top of a larger HDFS installation. It is used extensively inside of Yahoo. The use of HOD requires an understanding of torque: `http://www.clusterresources.com/pages/products/torque- resource-manager.php`. The author and the team the author was working with found it too complex for the benefits provided and discontinued using it.

HOD is described on the Hadoop site: `http://hadoop.apache.org/core/docs/r0.19.1/hod_user_guide.html`. HOD has probably improved significantly because the author used it last with Hadoop 0.16.1. The author recommends avoiding HOD unless there is a local torque expert to handle the torque installation and day-to-day operation.

---

public String getSessionId()

This method looks up the value of the key `session.id` in the configuration and returns it. If the value is unset, an empty `String` is returned. This is primarily used by HOD to distinguish different virtual clusters. The session name may also help distinguish this job in the metrics reporting framework.

public void setSessionId(String sessionId)

This method stores `sessionId` in the configuration under the key `session.id`. This value will be used as a token in the name used to identify any metrics that are reported by this job. This method is primarily intended for use by HOD.

public JobPriority getJobPriority()

This method looks up the value of the key `mapred.job.priority` in the configuration. If the value is unset, `JobPriority.NORMAL` is returned. If the value cannot be parsed as a `JobPriority`, an `IllegalArgumentException` is thrown. (Hadoop versions prior to 0.19 had only this simple mechanism for handling multiple running jobs on a cluster.)

A job with a higher priority has first right of refusal for any map or reduce task slot available on the cluster. If jobs have equal priority, the first requester gets the open task slots. There is no preemption of executing tasks.

Caution Queuing multiple jobs into a cluster with this mechanism can result in a cluster deadlock in which no job can complete.

Hadoop 0.19 also provides a queuing mechanism that provides rich control over how task slots are allocated between multiple competing jobs. (Refer to Chapter 8.)

public void setJobPriority(JobPriority prio)

Store the `String` representation of `prio` in the configuration under the key `mapred.job.priority`.

Jobs with a higher priority have first choice of available task slots when executing in an environment in which multiple jobs are queued into a cluster.

public boolean getProfileEnabled()

This method looks up the value of the key `mapred.task.profile` in the configuration. If the value is unset or not the

String, true, false is returned; otherwise, true is returned.

If this is true, the framework may profile specific tasks by using the results of getProfileTaskRange() to select individual tasks to profile. Profiling is performed on both map tasks and reduce tasks if enabled. If only profiling on maps is required, the user must specify a range of reduce values that is not available to the setProfileTaskRange() method, with false as the first argument. If the number of reduces is 10, the reduces will be 0 through 9, and calling setProfileTaskRange(false, "10") would effectively disable profiling for reduces. It is harder to absolutely know the number of map tasks, but the same technique applies.

public void setProfileEnabled(boolean newValue)

This method stores the String value of newValue in the configuration under the key mapred.task.profile. If newValue is true, profiling information will be collected for tasks that match the getProfileTaskRange() method.

public String getProfileParams()

This method looks up the value of the key mapred.task.profile.params in the configuration, returning that value. If the value is unset, the following String is returned:

```
-agentlib:hprof=cpu=samples,heap=sites,force=n,thread=y,verbose=n,file=%s
```

This String is passed to the JVM to control how the profiling is performed for the task to be profiled. At runtime, for a profiled task a single %s will be substituted in the value with the name of the task-specific profile.out file.

public void setProfileParams(String value)

This method stores value in the configuration under the key mapred.task.profile.params. This value, with a single %s substituted with the name of the task-specific profile output file, is passed to the JVM of a task to be profiled.

public Configuration.IntegerRanges getProfileTaskRange (boolean isMap)

This method looks up the value of the key mapred.task.profile.maps if isMap is true, or the value of the key mapred.task.profile.reduces if isMap is false. If the value is unset, the range 0-2 is constructed. If the value cannot be parsed as a set of ranges, an IllegalArgumentException is thrown. Ranges are specified as a set of comma-separated values, in which each value is a single positive integer or two positive integers separated by a dash. Some valid ranges include the following:

- 0-2: tasks 0, 1, and 2
- 2: task 2 only
- 0-2,5–7: tasks 0, 1, 2, 5, 6, and 7
- -7,0,6–11: tasks 0, 5, 6, 7, 8, 9, 10, and 11 (ordering is not needed, and overlap is allowed)
- 0-3, 9–11,13: tasks 0, 1, 2, 3, 9, 10, 11, and 13

No checking is performed to ensure that the individual ranges in a comma-separated set do not overlap and ordering is not required. A linear search through the list in the order supplied is performed for each task when profiling is enabled.

public void setProfileTaskRange(boolean isMap, String newValue)

This method stores newValue under the key mapred.task.profile.maps if isMap is true, or the key mapred.task.profile.reduces if isMap is false. The value must be a comma-separated list of ranges composed of positive integers. During task setup, the TaskRunner will get this value via getProfileTaskRange(), if the value stored in the key is not a valid range, an exception will be thrown and the task will be aborted. (See Configuration.IntegerRanges getRange(), earlier in this chapter, for a discussion of range formats.)

Some valid ranges include the following:

- 0–2: tasks 0, 1, and 2

- 2: task 2 only
- `0–2,5–7`: tasks 0, 1, 2, 5, 6, and 7
- `5-7,0,6–11`: tasks 0, 5, 6, 7, 8, 9, 10, and 11 (ordering is not needed, and overlap is allowed)
- `0–3, 9–11,13`: tasks 0, 1, 2, 3, 9, 10, 11, and 13

No checking is performed to ensure that the individual ranges in a comma-separated set do not overlap and ordering is not required. A linear search through the list in the order supplied is performed for each task, when profiling is enabled.

public String getMapDebugScript()

This method returns the value of the key `mapred.map.task.debug.script` from the configuration. If the value is unset, `null` is returned. This script will be run for a map task that the framework is going to mark as failed or about to kill.

The value is the script and script arguments to be used to debug failed tasks. The value will be split into tokens using the space character as a separator. Five additional arguments are added:

- The path to the task standard output file
- The path to the task standard error file
- The path to the task syslog output file
- The path to the file containing the XML representation of the `JobConf` object for the task
- The program name if this is a pipes job or empty `String`

All the tokens are passed to the shell to be executed as a command. The input of the command will be connected to `/dev/null`, and the standard and error output collected in a single stream.

The script is run with the current working directory as the task local directory. If the script is not resident on all the TaskTracker nodes and normally executable, it must be distributed via the `DistributedCache` and symlinked.

The following code fragment arranges for the executable program that is on the local file system at `LocalFileSystemPathToDebugScript` to be distributed to all tasks and made available for execution as `./MyDebugScript`. In Listing A-11, the URI fragment `#MyDebugScript` informs the framework to create a symbolic link named `MyDebugScript` between the task local copy of `LocalFileSystemPathToDebugScript` and the current working directory of the task.

Listing A-11: Adding a Debug Script to the DistributedCache

```
Job.setMapDebugScript("./MyDebugScript map argument2 argument3" );
DistributedCache.createSymlink(job);
DistributedCache.addFile("HDFSFileSystemPathToDebugScript#MyDebugScript");
```

The script will be invoked in the task local directory via the following shell command:

```
./MyDebugScript map argument1 argument2 argument3 taskStdoutFile taskStderrFile➥
taskSyslogFile taskJobConfXmlFile pipesProgramName➥
< /dev/null 2>&1 > ./debugout
```

The user can specify how many lines to keep from the output by setting an `int` value on the key `mapred.debug.out.lines`. The default value `-1` keeps all the output lines. The value specified is the number of lines from the tail of the file to keep. If the value is `10`, the last 10 lines of the output file are saved.

This information is made available via the JobTracker web interface in the task detail output.

> **Caution** Having shell metacharacters in the value of `mapred.map.task.debug.script` may lead to unpredictable results.

public void setMapDebugScript(String mDbgScript)

This method stores `mDbgScript` in the configuration under the key `mapred.map.task.debug.script`. (See `getMapDebugScript()` for details on the format and use of `mDbgScript`.)

public String getReduceDebugScript()

This method return the value stored under the key `mapred.reduce.task.debug.script`. If the value is unset, `null` is returned. The usage is the same as the usage of `getMapDebugScript()`, except it reduces tasks.

public void setReduceDebugScript(String rDbgScript)

This method stores `rDbgScript` in the configuration under the key `mapred.reduce.task.debug.script`. (See `getMapDebugScript()` for details on the format and use of `rDbgScript`.) This script will be used for failed or about to be killed reduce tasks.

Job End Notification

If a URL is stored in the configuration under the key `job.end.notification.url` or via `setJobEndNotification()`, an HTTP GET will be made on this URL when the job finishes.

The text `$jobId` and `$jobStatus`, if present in the URL, is replaced with the job id and the job status, respectively. The job status will be either `SUCCEEDED` or `FAILED`.

The parameter `job.end.retry.attempts` controls the number of retry attempts that will be made if the HTTP GET does not return the numeric status code of `200`. The default is `0` retries.

The parameter `job.end.retry.interval` controls the delay between retry attempts, with a default value of 30,000 msec.

If either parameter is set and the value cannot be converted to an `int`, a `NumberFormatException` will be thrown in the context of the JobTracker, which may cause the JobTracker to abort or otherwise behave unpredictably.

---

public String getJobEndNotificationURI()

This method looks up the value of the key `job.end.notification.url` in the configuration and returns that value. If the value is unset, `null` is returned. The value will be used as a URL in an HTTP GET.

public void setJobEndNotificationURI(String uri)

This method stores `uri` in the configuration under the key `job.end.notification.url`.

public String getQueueName()

This method looks up the key `mapred.job.queue.name` in the configuration and returns the value. If the value is unset, `default` is returned.

Queues, which are new to Hadoop 0.19.0, provide a mechanism to allow multiple jobs to share cluster resources in a specified manner (refer to Chapter 8).

public void setQueueName(String queueName)

This method stores `queueName` in the configuration under the key `mapred.job.queue.name`. If `queueName` is not a valid queue name, the JobTracker behavior is unpredictable.

Memory Limits for Tasks and Their Children

Hadoop provides a mechanism to control the limit of virtual memory that an individual task and the task's children use.

The user can specify the maximum amount of memory, in kilobytes, in the configuration under the key `mapred.task.maxmemory`; the method `setMaxVirtualMemoryForTask(vmem)` can also be used. The overall default can be specified by storing the value in kilobytes under the key `mapred.task.default.maxmemory`.

When the virtual memory consumption of a task and its children exceed this value, the task is killed by the framework, and marked as failed. This is predicated on the system reporting virtual memory usage for processes in kilobytes.

The default value for `mapred.task.maxmemory` is `-1`. The value of `-1` tells the framework to use the framework limit, which is stored under the key `mapred.task.default.maxmemory`. The default value for this key is `536,870,912` kilobytes (roughly one-half terabyte).

---

long getMaxVirtualMemoryForTask() {

This method looks up the value of the key `mapred.task.maxmemory` and returns it as a `long`. If the value is unset, `-1` is returned. If the value cannot be converted to a long, a `NumberFormatException` will be thrown.

void setMaxVirtualMemoryForTask(long vmem) {

This method stores the `String` version of `vmem` in the configuration under the key `mapred.task.maxmemory`. If a task and its children's virtual memory usage exceed this value, the task will be killed by the framework.

Convenience Methods

These methods provide convenience functions for accessing the configuration data.

public int size()

This method returns the number of keys in the configuration.

public void clear()

This method completely clears all keys and values from the configuration.

public Iterator<Map.Entry<String,String>> iterator()

This method returns an integrator for to the key/value pairs stored in the configuration.

public void writeXml(OutputStream out) throws IOException

This method serializes the key/value pairs in the configuration to XML in the standard configuration file format and writes the data to `out`.

The destination for this output data can be used as input to the `addResource()` method. This method is used by the framework to serialize the job configuration and store it in HDFS so that the individual tasks load the job configuration at task start.

public ClassLoader getClassLoader()

This method returns the class loader that is used to search for resources that are added via the `addResource()` methods and to instantiate classes when a class is being returned.

public void setClassLoader(ClassLoader classLoader)

This method sets the class loader to be used for locating resources and instantiating classes to `classLoader`.

This is primarily used by the framework when preparing map and reduce tasks to include the `DistributedCache` classpath items in the classpath.

public String toString()

This returns a `String` composed of the names of all the resources that were loaded into this configuration. This method does not return the key/value pairs that are stored in the configuration.

Methods Used to Pass Configurations Through SequenceFiles

The configuration class implements the `Writable` interface, which allows the framework to serialize and deserialize the configuration. These two methods are required for the `Writable` interface. It is not clear that these methods are used by the framework at the current time.

public void readFields(DataInput in) throws IOException

This method will deserialize the configuration key/value pairs. The key value pairs will be read from the `DataInput` stream `in`.

public void write(DataOutput out) throws IOException

This method serializes the configuration into a form that is suitable for use in `SequenceFiles`. The serialized data is written to the `DataOutput` stream `out`.