CELEBRATING
10 YEARS OF JAVA & .NET

Creating .NET-based Mappers and Reducers for Hadoop with JNBridgePro

Apache Hadoop | .NET-based MapReducers

Creating .NET-based Mappers and Reducers for Hadoop with JNBridgePro

Summary

The Apache Hadoop framework enables distributed processing of very large data sets. Hadoop is written in Java, and has limited methods to integrate with "mapreducers" written in other languages. This lab demonstrates how you can use JNBridgePro to program directly against the Java-based Hadoop API to create .NET-based mapreducers.

Hadoop mapreducers, Java, and .NET

Apache Hadoop (or Hadoop, for short) is an increasingly popular Java-based tool used to perform massively parallel processing and analysis of large data sets. Such large data sets requiring special processing techniques are often called "Big Data." The analysis of very large log files is a typical example of a task suitable for Hadoop. When processed using Hadoop, the log files are broken into many chunks, then farmed out to a large set of processes called "mappers," that perform identical operations on each chunk. The results of the mappers are then sent to another set of processes called "reducers," which combine the mapper output into a unified result. Hadoop is well-suited to running on large clusters of machines, particularly in the cloud. Both Amazon EC2 and Microsoft Windows Azure, among other cloud offerings, either provide or are developing targeted support for Hadoop.

In order to implement the functionality of a Hadoop application, the developer must write the mappers and reducers (sometimes collectively called "mapreducers"), then plug them into the Hadoop framework through a well-defined API. Because the Hadoop framework is written in Java, most mapreducer development is also done in Java. While it's possible to write the mapreducers in languages other than Java, through a mechanism known as Hadoop Streaming, this isn't an ideal solution, as the data sent to the mapreducers over standard input needs to be parsed and then converted from text to whatever native form is being processed. Handling the data being passed through standard input and output incurs overhead, as well as additional coding effort.

The alternative that we present in this lab is a way to create .NET-based mapreducers by programming against the Hadoop API using JNBridgePro. In this lab, the .NET-based mapreducers run in the actual Hadoop Java processes (which is possible if the Hadoop cluster is running on Windows machines), but we will also discuss ways to run the .NET sides outside the Java processes. In this example, we show how to host the maximal amount of mapreducer functionality in .NET, although you could use the same approach to host as much or as little of the functionality in .NET as you like, and host the rest in Java. You will come away with an understanding of how to create .NET-based Hadoop mapreducers and deploy them as part of a Hadoop application. The code we provide can be used as a pattern upon which you can create your own .NET-based mapreducers.

You might want or need to write mapreducers in .NET for a number of reasons. As examples, you might have an investment in .NET-based libraries with specialized functionality that needs to be used in the Hadoop application. Your organization may have more developers with .NET skills than with Java skills. You may be planning to run your Hadoop application on Windows Azure, where, even though the Hadoop implementation is still in Java and there is support for Java, the majority of the tooling is far more friendly to .NET development.

This lab is not a tutorial in Hadoop programming, or in deploying and running Hadoop applications. For the latter, there is a good tutorial here (http://v-lad.org/Tutorials/Hadoop/00 - Intro.html). The tutorial refers to some older versions of Eclipse and Hadoop, but will work with more recent versions. Even if you're familiar with Hadoop, the example builds on some of the setup in the tutorial, so it might be worth working through the tutorial beforehand. We will point out these dependencies when we discuss how the Hadoop job should be set up, so you can decide whether to build on top of the tutorial or make changes to the code in the lab.

Example

The lab is based on the standard "word count" example that comes with the Hadoop distribution, in which the occurrences of all words in a set of documents are counted. We've chosen this example because it's often used in introductory Hadoop tutorials, and is usually well understood by Hadoop programmers. Consequently, we won't spend much time talking about the actual functionality of the example: that is, how the word counting actually works.

What this example does is move all the Java-based functionality of the "word count" mapreducers into C#. As you will see, we need to leave a small amount of the mapreducer in Java as a thin layer. Understanding the necessity of this thin layer, and how it works, provides a design pattern that can be used in the creation of other .NET-based mapreducers.

Interoperability strategy

At first glance, the apparent approach to interoperability would be to use JNBridgePro to proxy the Java-based Mapper and Reducer interfaces and the MapReduceBase abstract class into .NET, then program in C# against these proxies. Then, still using JNBridgePro, proxy the .NET-based mapper and reducer classes, and register those proxies with the Hadoop framework. The resulting project would use bidirectional interoperability and would be quite straightforward. Unfortunately, this approach leads to circularities and name clashes: the proxied mapper and reducer will contain proxied parameters with the same name as the actual Java-based Hadoop classes. In other words, there will be proxies of proxies, and the result will not work. While it is possible to edit the jar files and perform some other unusual actions, the result would be confusing and would not work in all cases. So we need to take a different approach.

Instead, we will create thin Java-based wrapper classes implementing the Mapper and Reducer interfaces, which will interact with the hosting Hadoop framework, and which will also call the .NET-based functionality through proxies, making this a Java-to-.NET project. In the cases where the .NET functionality needs to access Java-based Hadoop objects, particularly OutputCollectors and Iterators, it will be done indirectly, through callbacks. The resulting code is much simpler and more elegant.

The original WordCount example

Let's start with the original Java-based "word count" mapper and reducer, from the example that comes with Hadoop. We will not be using this code in our example, and we will not be discussing how it works (it should be fairly straightforward if you're familiar with Hadoop), but it will be useful as a reference when we move to the .NET-based version.

Here is the mapper:

```
/**
 * WordCount mapper class from the Apache Hadoop examples.
 * Counts the words in each line.
 * For each line of input, break the line into words and emit them as
 * (word , 1).
 */
public class WordCountMapper extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value,
                        OutputCollector<Text, IntWritable> output,
                        Reporter reporter) throws IOException {
                String line = value.toString();
                StringTokenizer itr = new StringTokenizer(line);
                while (itr.hasMoreTokens()) {
                        word.set(itr.nextToken());
                        output.collect(word, one);
                }
        }
}
```

And here is the reducer:

```
/**
 * From the Apache Hadoop examples.
 * A WordCount reducer class that just emits the sum of the input values.
 */
public class WordCountReducer extends MapReduceBase
        implements Reducer<Text, IntWritable, Text, IntWritable> {
            public void reduce(Text key, Iterator<IntWritable> values,
                        OutputCollector<Text, IntWritable> output,
                        Reporter reporter) throws IOException {
                  int sum = 0;
                  while (values.hasNext()) {
                        sum += values.next().get();
                  }
                  output.collect(key, new IntWritable(sum));
            }
      }
```

Migrating the functionality to .NET

What we want to do next is migrate as much of the mapper and reducer functionality as possible to .NET (in this case, C#) code. Note that we can't migrate all of it verbatim; the Java code references Hadoop-specific classes like Text, IntWritable, LongWritable, OutputCollector, and Reporter, as well as other crucial Java classes such as Iterator. Text, IntWritable, and LongWritable can be easily converted to string, int, and long, which are automatically converted by JNBridgePro. However, while it is possible to convert classes like OutputCollector and Iterator to and from native .NET classes like ArrayList, such conversions are highly inefficient, since they involve copying every element in the collection, perhaps multiple times. Instead, we will continue to use the original OutputCollector and Iterator classes on the Java side, and the .NET code will only use them indirectly, knowing nothing about the actual classes. Callbacks provide a mechanism for doing this.

Here is the C# code implementing the mapper and reducer functionality:

```
namespace DotNetMapReducer
{
      // used for callbacks for the OutputCollector
      public delegate void collectResult(string theKey, int theValue);

      // used for callbacks to the Iterator
      public delegate object getNextValue();
      // returns null if no more values, returns boxed integer otherwise

      public class DotNetMapReducer
      {
            public void map(string line, collectResult resultCollector)
            {
                  StringTokenizer st = new StringTokenizer(line);
                  while (st.hasMoreTokens())
                  {
                        string nextToken = st.nextToken();
                        resultCollector(nextToken, 1);
```

```
                }
            }
            public void reduce(string key, getNextValue next, collectResult resultCollector)
            {
                int sum = 0;
                object nextValue = next(); // get the next one, if there
                while (nextValue != null)
                {
                    sum += (int)nextValue;
                    nextValue = next();
                }
                resultCollector(key, sum);
            }
        }
        public class StringTokenizer
        {
            private static char[] defaultDelimiters = { ' ', '\t', '\n', '\r', '\f' };

            private string[] tokens;
            private int numTokens;
            private int curToken;

            public StringTokenizer(string line, char[] delimiters)
            {
                tokens = line.Split(delimiters);
                numTokens = tokens.Length;
                curToken = 0;
            }

            public StringTokenizer(string line)
                : this(line, defaultDelimiters)
            {
            }

            public bool hasMoreTokens()
            {
                if (curToken < numTokens) return true;
                else return false;
            }

            public string nextToken()
            {
                if (hasMoreTokens()) return tokens[curToken++];
                else throw new IndexOutOfRangeException();
            }
        }
```

```
    }
```

StringTokenizer is just a .NET-based reimplementation of the standard Java StringTokenizer class, and we won't be discussing it further.

Note the two delegates collectResult and getNextValue that are used by the mapreducer. These are ways to call back into the Java code for additional functionality, possibly using classes like OutputCollector and Iterator that the .NET code knows nothing about. Also note that the .NET code uses string and int where the Java code had Text and IntWritable (and LongWritable); the wrapper code will handle the conversions.

Once we have the .NET functionality built and tested, we need to proxy the mapreducer class and supporting classes. We then incorporate the proxy jar file, jnbcore.jar, and bcel-5.1-jnbridge.jar into our Java Hadoop project and can start writing the Java-based mapper and reducer wrappers. Here they are:

```
public class MapperWrapper extends MapReduceBase
implements Mapper<LongWritable, Text, Text, IntWritable> {

        private static MapReducerHelper mrh = new MapReducerHelper();

        private DotNetMapReducer dnmr = new DotNetMapReducer();

        public void map(LongWritable key, Text value,
                OutputCollector<Text, IntWritable> output,
                Reporter reporter) throws IOException {

                OutputCollectorHandler och = new OutputCollectorHandler(output);

                dnmr.map(value.toString(), och);

                Callbacks.releaseCallback(och);
        }
}

public class ReducerWrapper extends MapReduceBase
implements Reducer<Text, IntWritable, Text, IntWritable> {

        private static MapReducerHelper mrh = new MapReducerHelper();

        private DotNetMapReducer dnmr = new DotNetMapReducer();

        public void reduce(Text key, Iterator<IntWritable> values,
                OutputCollector<Text, IntWritable> output,
                Reporter reporter) throws IOException {

                IteratorHandler ih = new IteratorHandler(values);
                OutputCollectorHandler och = new OutputCollectorHandler(output);
```

```
                    dnmr.reduce(key.toString(), ih, och);
                    Callbacks.releaseCallback(ih);
                    Callbacks.releaseCallback(och);
          }
    }
```

Note that the only purpose of these thin wrappers is to interface with the Hadoop framework, host the .NET-based functionality, and handle passing of values to and from the .NET components (along with necessary conversions).

There are two callback objects, IteratorHandler and OutputCollectorHandler, which encapsulate the Iterator and OutputCollector objects. These are passed where the .NET map() and reduce() methods expect delegate parameters, and are used to hide the actual Hadoop Java types from the .NET code. The mapreducer will simply call the resultCollector() or getNextValue() delegate, and the action will be performed, or value returned, without the .NET side knowing anything about the mechanism used by the action.

Since callbacks consume resources (particularly, a dedicated thread for each callback object), and there can be many invocations of map() and reduce(), it is important to release the callback objects (using the Callbacks.releaseCallback() API) to release those threads when they are no longer needed. If you do not make those calls, performance will degrade substantially.

Here is the Java code for the two callback classes:

```
    public class OutputCollectorHandler implements collectResult
    {
          private OutputCollector<Text, IntWritable> outputCollector = null;

          public OutputCollectorHandler(OutputCollector<Text, IntWritable>
                    theCollector)
          {
                    outputCollector = theCollector;
          }

          public void Invoke(String theKey, int theValue)
          {
                    try
                    {
                              outputCollector.collect(new Text(theKey),
                                        new IntWritable(theValue));
                    }
                    catch(IOException e)
                    {
                              // not sure why it would throw IOException anyway
                    }
          }
}
import System.BoxedInt;
import System.Object;

public class IteratorHandler implements getNextValue
{

          private Iterator<IntWritable> theIterator = null;

          public IteratorHandler(Iterator<IntWritable> iterator)
          {
                    theIterator = iterator;
          }

          // returns null if no more values, otherwise returns a boxed integer
          public Object Invoke()
          {
                    if (!theIterator.hasNext()) return null;
                    else
                    {
                              IntWritable iw = theIterator.next();
                              int i = iw.get();

                              return new BoxedInt(i);
                    }
          }
}
```

The two callback objects encapsulate the respective Java collections and perform the appropriate conversions when their Invoke() methods are called. The IteratorHandler, rather than providing the typical hasNext()/getNext() interface, has a single Invoke() method (this is how callbacks work in Java-to-.NET projects), so we've written Invoke() to return null if there are no more objects, and to return the integer (boxed, so that it can be passed in place of a System.Object), when there is a new value. There are other ways you can choose to do this, but this method will work for iterators that return primitive objects.

Finally, we need to configure JNBridgePro. For maximum flexibility, we've chosen to configure it programmatically, through the MapReducerHelper class. Since configuration can only happen once in each process, and must happen before any proxy call, we've created MapReducerHelper to perform the configuration operation inside its static initializer, which is executed when the class is loaded. This will happen only once per process and is guaranteed to be done before any proxies are called. Here is Java-based MapReducerHelper:

```
public class MapReducerHelper
{
        static
        {
                Properties p = new Properties();
                p.put("dotNetSide.serverType", "sharedmem");
                p.put("dotNetSide.assemblyList.1",
                        "C:/DotNetAssemblies/DotNetMapReducer.dll");
                p.put("dotNetSide.javaEntry",
                        "C:/Program Files/JNBridge/JNBridgePro v6.0/4.0-targeted/JNBJavaEntry.dll");
                p.put("dotNetSide.appBase",
                        "C:/Program Files/JNBridge/JNBridgePro v6.0/4.0-targeted");
                DotNetSide.init(p);
        }
}
```

The paths in the configuration will likely be different in your deployment, so you will need to adjust them accordingly.

Finally, we create the Java-based Hadoop driver in the usual way, specifying the new wrappers as the mapper and reducer classes:

```
public class WordCountDotNetDriver
{
        public static void main(String[] args)
        {
                JobClient client = new JobClient();
                JobConf conf = new JobConf(WordCountDotNetDriver.class);

                // specify output types
                conf.setOutputKeyClass(Text.class);
                conf.setOutputValueClass(IntWritable.class);

                conf.setInputFormat(TextInputFormat.class);
                conf.setOutputFormat(TextOutputFormat.class);

                // specify input and output DIRECTORIES (not files)
                FileInputFormat.setInputPaths(conf, new Path("In"));
                FileOutputFormat.setOutputPath(conf, new Path("Out"));

                // specify a mapper
                conf.setMapperClass(MapperWrapper.class);
                conf.setCombinerClass(ReducerWrapper.class);
                // specify a reducer
                conf.setReducerClass(ReducerWrapper.class);

                client.setConf(conf);
                try {
                        JobClient.runJob(conf);
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Deploying and running the Hadoop application

At this point we can deploy and run the application. Start by deploying the application to your Hadoop cluster. (It needs to be a cluster of Windows machines, since we're using shared memory. We'll talk about Linux clusters later.) If you have a setup like the one described in the aforementioned tutorial, just copy the source files over and build.

Make sure that the appropriate version (x86 or x64, depending on whether you're running your Hadoop on 32-bit or 64-bit Java) of JNBridgePro is installed on all the machines in the cluster and that an appropriate license (which may be an evaluation license) is installed, and make sure that the dotNetSide.javaEntry and dotNetSide.appBase properties in MapReducerHelper agree with the installed location of JNBridgePro. If not, either install JNBridgePro in the correct place, or edit MapReducerHelper and rebuild.

You will need to put the proxy jar file as well as jnbcore.jar and bcel-5.1-jnbridge.jar in the Hadoop classpath. There are a couple of ways to do this. You can add the paths to these files to the HADOOP_CLASSPATH environment variable. Alternatively, you can copy these jar files to the Hadoop lib folder.

Finally, copy to each machine in the Hadoop cluster the .NET DLL file containing the .NET classes, and put it in the location specified in the dotNetSide.assemblyList.1 property in MapReducerHelper.

Once this is all done, start up all your Hadoop nodes. Make sure that in your HDFS service you've created a directory "In", and that you've uploaded all the .txt files in the root Hadoop folder: mostly licensing information, release notes, and readme documents. Feel free to load additional documents. If there is an "Out" directory, delete it along with its contents. (If the "Out" directory exists when the program is run, an exception will be thrown.)

Now, run the Hadoop application. It will run to completion, and you will find an HDFS folder named "Out" containing a document with the result.

The Hadoop job that you just ran worked in exactly the same way as any ordinary all-Java Hadoop job, but the mapreducer functionality was written in .NET and was running in the same processes as the rest of the Hadoop framework. No streaming was required to do this, and we were able to program against native Hadoop APIs.

Running the Hadoop job on Linux machines

As we've chosen to run the Hadoop application using JNBridgePro shared memory communications, we need to run our Hadoop cluster on Windows machines. This is because the .NET Framework needs to be installed on the machines on which the Hadoop processes are running.

It is possible to run the application on a cluster of Linux machines, but you will need to change the configuration to use tcp/binary communications, and then run .NET-side processes on one or more Windows machines. The simplest way to run a Java side is to configure and use the JNBDotNetSide.exe utility that comes with the JNBridgePro installation. Configure each Java side to point to one of the .NET-side machines. You can share a .NET side among multiple Java sides without any problem, although the fewer Java sides talking to each .NET side, the better performance you will get.

Note that changing from shared memory to tcp/binary does not require any changes to your .NET or Java code. You can use the same binaries as before; you only need to change the configuration.

Conclusion

This lab has shown how you can write .NET-based Hadoop mapreducers without having to use Hadoop streaming or implement parsing of the stream. The .NET code can include as much or as little of the mapreducer functionality as you desire; the rest can be placed in Java-based wrappers. In the example we've worked through, the .NET code contains all of the mapreducer functionality except for the minimal functionality required for connectivity with the Hadoop framework itself. The .NET code can run in the same processes as the rest of the Hadoop application (in the case that Hadoop is running on a Windows cluster), or on different machines if Hadoop is running on a Linux cluster.

You can use the example code as a generalized pattern for creating the wrappers that connect the Hadoop framework to the .NET-based mapreducers. The code is simple and straightforward, and variants will work in most mapreduce scenarios.

You can enhance the provided code for additional scenarios. For example, if you want to use tcp/binary, you can modify the Java-side configuration (in class MapReducerHelper) so that any specific instance of the Java side can choose to connect to one of a set of .NET sides running on a cluster; the assignments do not have to be made by hand. You can also use the provided code to support tight integration of .NET mapreducers in a Hadoop application running on Windows Azure. This approach provides more flexibility and improved performance over the Hadoop streaming used by the Azure implementation.

We expect that the provided code and design patterns will be useful in many scenarios we haven't even thought of. We'd love to hear your comments, suggestions, and feedback – you can contact us at labs@jnbridge.com.

You can download the source code for the lab at www.jnbridge.com/labs.