An Oracle White Paper
September 2009

Oracle JDBC Logging using java.util.logging

Introduction

The Oracle JDBC drivers use two different mechanisms to generate log output. Versions of the JDBC drivers for older versions of Java, 1.2 and 1.3, use an proprietary mechanism. Versions of the JDBC drivers for newer versions of Java, 1.4 and later, use the Java standard logging mechanism, java.util. logging. This note describes how to use java.util.logging with the Oracle JDBC drivers.

Basic Configuration

In order to generate any logging output from the JDBC drivers you have to use a logging enabled Oracle JDBC jar file, enable JDBC logging output, and configure java.util.logging. A simple configuration file for java.util.logging is sufficient in simple cases.

Configure the classpath

Oracle ships several jar files for each version of the drivers. The optimized jar files to do not contain any logging code. There will be no Oracle JDBC log output when using the optimized jar files. The jar files that do contain logging code are

- ojdbc5_g.jar
- ojdbc6_g.jar
- ojdbc5dms.jar minimal logging
- ojdbc6dms.jar minimal logging
- ojdbc5dms_g.jar
- ojdbc6dms_g.jar

Step 1: Make sure that a logging enabled jar file is the only Oracle JDBC jar file in your classpath.

Enable Logging

In order to get any log output from the Oracle JDBC drivers you must enable logging. There is a global switch that turns logging on and off. When it is off, the drivers will not produce any log output. When it is on, what logging is produced is controlled by the configuration of java.util.logging. There are two ways to enable the global logging switch, programmatically or setting a Java system property. You can use the programmatic way to control what parts of your program generate log output. If you cannot or do not want to change the source, you can set the Java system property to enable logging for the entire program execution.

Step 2a: globally enable logging by setting the oracle.jdbc.Trace system property

```
java -Doracle.jdbc.Trace=true ...
```

OR

Step 2b: In 11.1 you programmatically enable/disable logging with the following:

```
import java.lang.management.ManagementFactory;
import javax.management.Attribute;
import javax.management.MBeanServer;
import javax.management.ObjectName;
```

```
// get the MBean server
MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

// create an ObjectName pattern
ObjectName pattern  =
  new ObjectName("com.oracle.jdbc:type=diagnosability,*");

// get the JDBC diagnosability ObjectName
ObjectName diag = ((ObjectName[])
  (mbs.queryNames(pattern, null).toArray(new ObjectName[0])))[0];

// find out if logging is enabled or not
System.out.println("LoggingEnabled = "
  + mbs.getAttribute(diag, "LoggingEnabled"));

// enable logging
mbs.setAttribute(diag, new Attribute("LoggingEnabled", true));

// disable logging
mbs.setAttribute(diag, new Attribute("LoggingEnabled", false));
```

In 10g you programmatically enable/disable logging with the following:

```
oracle.jdbc.driver.OracleLog.setTrace(true);  // enable logging
...
oracle.jdbc.driver.OracleLog.setTrace(false); // disable logging
```

If this is all you do you will get minimal logging of serious errors written to the console. Usually this is not useful. In order to generate more and probably more useful output, you must configure java.util.logging.

Configue java.util.logging

java.util.logging is a very rich and powerful tool. Describing all the things you can do with it is beyond the scope of this note. This note provides a basic set of tools that will let you generate useful log output. For more complex configurations look at the JavaDoc for java.util.logging.

You can configure java.util.logging either programatically or via a configuration file. For the most part there is little need to configure it programatically. You can turn Oracle JDBC logging on and off programmatically using OracleLog.setTrace. In most cases there is no need to change the configuration during the course of execution. This note will only cover configuration files. If you must use programmatic configuration, information in the rest of this note should help you figure out what values to use when configuring java.util.logging programmatically.

One place to look for configuration information is the OracleLog.properties file in the demo directory of your JDBC installation. This file contains basic information on how to configure java.util.logging and provides some initial settings that you can start with. In order to use a config file you must identify that file to the Java runtime. You tell Java about your file by setting a system property. You can use both java.util.logging.config.file and oracle.jdbc.Trace at the same time.

Step 3:

```
java -Djava.util.logging.config.file=/jdbc/demo/OracleLog.properties \
-Doracle.jdbc.Trace=true …
```

This will use the default OracleLog.properties file (assuming it is reachable from /jdbc/demo). That may get you the output you want, or it may not. The rest of this note will show you how to create your own config file and in the process help you understand how to modify the sample OracleLog.properties file.

Step 4: create a file, for example myConfig.properties, and insert the following.

```
level=SEVERE
oracle.jdbc.level=FINE
oracle.jdbc.handlers=java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level=FINE
java.util.logging.ConsoleHandler.formatter = \
java.util.logging.SimpleFormatter
```

Save this file and execute the Java command above replacing OracleLog.properties with myConfig.properties. This can produce a large amount of output, so make sure your program execution is short.

Advanced Configuration

The basic configuration provides a working setup that traces everything to the console. Let's modify the config file to dump everything to a file instead. Instead of using the ConsoleHandle, use the FileHandler.

```
.level=SEVERE
oracle.jdbc.level=FINE
oracle.jdbc.handlers=java.util.logging.FileHandler
java.util.logging.FileHandler.level=FINE
java.util.logging.FileHandler.pattern = jdbc.log
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
```

This will generate exactly the same log output, but instead send it to a file named jdbc.log in the current directory.

Step 5: You will want to reduce the amount of detail. You control the level of detail by changing the level settings. The defined levels from least detail to most are

OFF	
SEVERE	SQLExceptions and internal errors
WARNING	SQLWarnings and bad but not fatal internal conditions
INFO	Infrequent events
CONFIG	SQL strings
FINE	User code calls to the public API
FINER	Calls to internal methods and internal calls to the public API
FINEST	Calls to high volume internal methods and internal debug messages
ALL	High volume internal debug messages not in FINEST

In order to reduce the amount of detail, change java.util.logging.FileHandler.level from FINE to CONFIG

```
java.util.logging.FileHandler.level=CONFIG
```

Setting the FileHandler level will control what log messages end up in the log file. This will log the SQL that is executed and any errors or warnings.

Using Loggers

Setting the level as above reduces all the logging output from JDBC. Sometimes we want to see a lot of output from one part of the code and very little from other parts. To do that you must understand more about loggers.

Loggers exist in a tree structure defined by their names. The root logger is named "", the empty String. If you look at the first line of the config file you see '.level=SEVERE'. This is setting the level of the root logger. The next line is 'oracle.jdbc.level=FINE'. This sets the level of the logger named 'oracle.jdbc'. The oracle.jdbc logger is a member of the logger tree. Its parent is named 'oracle'. The parent of the oracle logger is the root logger. Logging messages are sent to a particular logger, for example oracle.jdbc. If the message passes the level check at that level the message is passed to the handler at that level, if any, and to the parent logger. So a log message sent to oracle.log is compared against that logger's level, CONFIG if you are following along. If the level is the same or less (less detailed) then it is sent to the FileHandler and to the parent logger, 'oracle'. Again it is checked against the level. If as in this case, the level is not set then it uses the parent level, SEVERE. If the message level is the same or less it is passed to the handler, which there isn't one, and sent to the parent. In this case the parent is the root logger.

All this tree stuff didn't help you reduce the amount of output. What will help is that the JDBC drivers use several subloggers. If you restrict the log messages to one of the subloggers you will get substantially less output. The loggers used by the Oracle JDBC drivers include

• oracle.jdbc	almost all Oracle JDBC messages
• oracle.jdbc.aq	Advanced Queuing
• oracle.jdbc.driver	the core driver code
• oracle.jdbc.pool	DataSources and Connection pooling
• oracle.jdbc.rowset	RowSets
• oracle.jdbc.xa	distributed transactions
• oracle.sql	complex SQL data types

The drivers may use other loggers as well. That will vary from release to release.

A Detailed Example

Suppose you want to trace what is happening in the oracle.sql component, but you also want to capture some basic information about the rest of the driver. This is a more complex use of logging. Here is the config file.

```
        #
        # set levels
        #
1       level=SEVERE
2       oracle.jdbc.level=ALL
3       oracle.jdbc.driver.level=CONFIG
4       oracle.jdbc.pool.level=SEVERE
5       oracle.jdbc.util.level=SEVERE
6       oracle.sql.level=FINE
7       oracle.net.level=SEVERE
        #
        # Config handlers
        #
8       oracle.handlers=java.util.logging.ConsoleHandler
9       java.util.logging.ConsoleHandler.level=ALL
10      java.util.logging.ConsoleHandler.formatter = \
        java.util.logging.SimpleFormatter
```

Let's consider what each line is doing.

1 Set the root logger to SEVERE. We don't want to see any logging from other, non-Oracle components unless something fails badly, so we set the default level for all loggers to SEVERE. Each logger inherits its level from its parent unless set explicitly.

By setting the root logger to SEVERE we insure that all other loggers inherit that level except for the ones we set otherwise.

2 We want output from both the oracle.sql and oracle.jdbc.driver loggers. Their common ancestor is oracle, so we set the level there to ALL. We will control the detail more explicitly at lower levels.

3 We only want to see the SQL execution from oracle.jdbc.driver so we set that to CONFIG. This is a fairly low volume level but will allow us to keep track of what our test is doing.

4 We are using a DataSource in our test and don't want to see all of that logging so we turn it SEVERE.

5 Similarly we don't want to see the logging from the oracle.jdbc.util package. If we were using XA or rowsets we would turn them off as well.

6 We want to see how our app is using oracle.sql so we set oracle.sql.level to FINE. This provides a lot of information about the public method calls without overwhelming detail.

7 We don't want to see the SQL*Net trace so we set oracle.net.level to SEVERE. This will show only fatal errors.

8 We are going to dump everything to stderr. When we run the test we will redirect stderr to a file.

9 We want to send everything to the console. We are doing the filtering with the loggers rather than the Handler this time.

10 We will use a simple, more or less human readable format.

When you run your test with this config file you will get a human readable document that contains moderately detailed information from the oracle.sql package, a little bit of information from the core driver code and nothing from any other code.

XMLFormatter

For large or complex log files, the Oracle JDBC development team prefers to use the XMLFormatter. The XMLFormatter includes all of the information in each log record while the SimpleFormatter omits some information. XML format log files are more suitable for automatic processing. Reading the raw XML format logs, however, is somewhat painful. With a little work you can view an XML format log file in whatever layout is most suitable for the problem you are trying to solve.

To generate an XML format log file replace the SimpleFormatter with XMLFormatter. It is also more convenient to send XML output to a file.

```
oracle.handlers = java.util.logging.FileHandler
java.util.logging.FileHandler.level=ALL
java.util.logging.FileHandler.pattern = jdbc.log
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter
```

In order to work with an XML format output file do the following:

1) rename the log output file to <foo>.html.

2) edit the file

a) remove everything outside the <log> ... </log> element.

b) before the <log> tag put

```
<html>
<head>
<link rel="stylesheet" type="text/css" href="java.util.logging.style.css" />
</head>
<body>
```

c) after the </log> tag put

```
</body>
</html>
```

Create a java.util.logging.style.css file is in the same directory as <foo>.html with the following content:

```
log {
         display: table;
}
record {
         display: table-row;
}
date {
         display: none;
}
millis {
         display: none;
}
sequence {
         display: none;
}
logger {
        display: none;
}
level {
        display: none;
}
class {
        display: table-cell;
        padding-right: 2px;
}
method {
        display: table-cell;
        padding-right: 2px;
}
thread {
        display: table-cell;
        padding-right: 2px;
}
message {
        display: table-cell;
        width: 400px;
}
```

You can then open <foo>.html with your favorite browser.

By editing the java.util.logging.style.css file you can control which information is displayed and highlight particularly important columns. Details of how to use CSS are beyond the scope of this white paper, but the above example should get you started.

Of course you can name the file anything you want so long as your browser recognizes the file content as html. You can rename java.util.logging.style.css so long as you change the <link> tag to match.

JavaNet Logging

Not all issues can be resolved by looking at the JDBC logging messages. For some it is helpful to look at the network packets the driver exchanges with the server. The OCI C library has a network trace mode that provides this information. This is well documented and can be found in the OCI documentation. The Thin driver does not use the OCI C library so that does not help when you are using the Thin driver. Beginning with Oracle Database Release 11.2, the Oracle JDBC Thin driver include a network trace capability. Like all other JDBC logging, it uses java.util.logging.

In order to generate network level trace information add the following line to your config file:

```
oracle.net.ns.level = FINEST
```

This will log the byte content of every packet that crosses the network. Obviously the volume will be high.

If your application has a single connection, it is easy enough to sift through the network log output. If your application has many connections, it becomes exceedingly difficult to separate the network packets for each connection. To aid in this situation, the Oracle JDBC jar files include oracle.jdbc.diagnostics.DemultiplexingLogHandler. This subclass of java.util.logging.FileHandler routes different log messages to different files. When used with oracle.net.ns logging, the packets for each connection will be written to a different file.

To use the DemultiplexingLogHandler add the following to your config file:

```
oracle.jdbc.diagnostics.DemultiplexingLogHandler.pattern = sqlnet_%s.log
oracle.jdbc.diagnostics.DemultiplexingLogHandler.limit = 50000000
oracle.jdbc.diagnostics.DemultiplexingLogHandler.count = 1
oracle.jdbc.diagnostics.DemultiplexingLogHandler.formatter = \
java.util.logging.XMLFormatter
oracle.net.ns.handlers = oracle.jdbc.diagnostics.DemultiplexingLogHandler
```

DemultiplexingLogHandler is a subclass of FileHandler. It supports the same pattern variables as FileHandler plus one additional pattern variable, %s. This is the connection or stream identifier. While you don't have to include it in your pattern, it will help you identify which log file goes with which connection.

Note that only the network packet log messages can be demultiplexed. The other JDBC logging messages do not have the information required to demultiplex them so there is no benefit of passing them through the demultiplexer.

With the above configuration, the packet log messages will be written to the DemultipleixengLogHandler and to whatever other handlers are attached to parent loggers. If you add the following line to your config file you will prevent the packet log messages from cluttering up your other log(s) while still sending the SQL*Net log to the DemultiplexingLogHandler..

```
oracle.net.level = SEVERE
```

The DemultiplexingLogHandler is attached to the logger named "oracle.net.ns" and the level for that logger is set to FINEST. That enables log messages to be written to the DemultiplexingLogHandler and to be passed to the parent logger. Setting the level of the parent logger, "oracle.ns" to SEVERE stops the propagation of the log messages and prevents them from appearing in other logs.

A rather obvious extension would be to enable all Oracle JDBC log messages to be demultiplexed by connection. We are working on it.

Conclusion

This paper shows you all the tools you will need for most purposes. java.util.logging is a powerful tool with lots of switches and knobs. You can send different parts of the log stream to different places and write custom filters that pick out exactly the log messages you want to see. These advanced uses are beyond the scope of this note. The basic tools described above should cover most of your needs and will give you a head start in learning the more advanced techniques. The best source for more information about java.util.logging is the JavaDoc.

One final note. The Oracle JDBC logging code varies dramatically from release to release. We are constantly striving to make it more useful and more maintainable. Older releases don't always do exactly what we or you would like them to do. This note more describes a philosophy rather than exactly what will happen in any given release. The techniques described here should get you something close to what you want, even if not exactly. Experiment. This note describes how we want logging to work and each subsequent release should be closer to this ideal. I hope this helps.

Oracle JDBC Logging using java.util.logging
September 2009
Author: Douglas Surber

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Oracle is committed to developing practices and products that help protect the environment