BEA WebLogic JRockit™ SDK

Using Code Coverage with WebLogic JRockit

Version 8.1 Service Pack 2 (Internal)
December 2003

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

Using Code Coverage with WebLogic JRockit

Using Code Coverage with WebLogic JRockit

This document describes how to use code coverage on applications you plan to run on WebLogic JRockit. It includes information on the following subjects:

What is Code Coverage?

Code coverage ensures that the tests you run are actually testing your code. When you run a tests, the possibility exists that the parameters of that test might not expose all of the code in your applicaiton; thus it will not be tested. Code coverage anticipates this situation and will tell you how much of your code was actually tested. Consider that even though your tests might all pass successfully, if you've only exercised half of your code, those tests could be woefully inaccurate.

How Code Coverage Works

Code coverage is an iterative tool; that is, it is most useful when you can compare the results of one application coverage with an earlier version of coverage for the same application. Code coverage is a simple process that requires you to define the coverage parameters at the command line and then let the application run. During the run, the coverage tool determines what code is actually being tested an which code is not. As it makes this determination, it writes to a filter file

all code not covered. You can then compare the information in the latest filter file to the same information in an earlier iteration to determine if the test suite you are using is covering more or less code.

When to Use Code Coverage

Code coverage requires knowledge of and access to the code itself rather than simply using the interface provided and is therefore considered a "white box" text. You will find code coverage most effective during the module testing and to some extent during integration testing. Depending upon what you are testing, you might find other occasions where code coverage will be helpful. Regression tests are usually black box tests and as such may be unsuitable for use with code coverage.

Enabling Code Coverage

Code coverage is run from the command line upon program startup. Simply inlcude the option `-XXcodecoverage` along with the rest of your command string; for example:

```
java -XXcodecoverage -Djrockit.codecoverage.filter=java.util.Hashtable;com.bea.*;
   -com.bea.blabla.* -Xgc:parallel -Xms:64 -Xmx:64 myClass
```

(where `myClass` is the name of the class that contains the `main()` method.)

Command Line Options for Code Coverage

Along with `-XXcodecoverage`, WebLogic JRockit provides a number of additional options that you can use to define the classes to cover, specify an alternate name for the filter file, identify the initial test, and so on. These options are also specified at startup; for example:

```
java -XXcodecoverage -Djrockit.codecoverage.filter=java.util.Hashtable;
   com.bea.*;-com.bea.blabla.* -Xgc:parallel -Xms:64 -Xmx:64 myClass
```

This section describes the how to use these command line options to enable these functions:

- Enabling Code Coverage
- Identifying the Classes to be Covered
- Naming the Filter File
- Identifying the File Location
- Identifying the Initial Test

- Displaying Code Coverage in the Verbose Mode
- Appending an Output File

Enabling Code Coverage

`-XXcodecoverage`

As mentioned in Enabling Code Coverage, `-XXcodecoverage` enables code coverage. This option cannot be used together with `-Xdebug`.

Identifying the Classes to be Covered

`-Djrockit.codecoverage.filter=<filterspec>`

The `filter=` option identifies which classes should be covered. Filterstrings starting with "-" will be considered as classes that should not be covered. Separate filters with ";" on windows and ":" on linux

Example:

-Djrockit.codecoverage.filter=java.util.Hashtable;com.bea.*;-com.bea.blabla.*

Naming the Filter File

`-Djrockit.codecoverage.filterfile=<filename>`

The `filterfile=` option sets the name of a file that will include the filter (specified or default). The file format is one filterstring per line. If no filter or filterfile is specified JRockit will default to filter.txt in the current directory.

Identifying the File Location

`-Djrockit.codecoverage.outputfile=<filename>`

The `outputfile=` option to set the file where output is written. If the output file cannot be opened for writing it will sequence through `<filename>_0`, `<filename>_1` until a usable name can be found. This can be useful if sevaral JVMs share a common commandline. If no outputfile is specified JRockit will default to coverage.txt in the current directory.

Identifying the Initial Test

`-Djrockit.codecoverage.testid=<id-string>`

The `testid` option sets the initial test identifier. If no test identifier is specified, WebLogic JRockit will default to an empty string.

Modifying the Testid During Runtime

The application you are covering can modify the testid during runtime by calling one of the following methods shown in Listing 1.

Listing 1 Modifying the Test ID During Runtime

```
package COM.jrockit.internal;
public final class CodeCoverage
{
   public static native void setTestID(String str);
   public static native void setTestIDAndReset(String str);
}
```

`setTestID` changes the test id so that all code that we has not covered been before will be reported with the new id.

`setTestIDAndReset` reports all code covered by the new test.

You will need to invoke these methods through reflection, for example:

```
Class.forName("COM.jrockit.internal.CodeCoverage").getMethod("setTestIDAnd
   Reset", new Class[]{String.class}).invoke(null, new Object[] {
   myNewTestIdString })
```

Displaying Code Coverage in the Verbose Mode

```
-Djrockit.codecoverage.verbose
```

The verbose option causes code coverage information to display on the screen. This option is useful when you want to see textual differences between coverage files.All information appears in plain text.

Appending an Output File

```
-Djrockit.codecoverage.appendoutput
```

The `appendoutput` option allows you to append code coverage results to the output file, rather than overwrite it.

Code Coverage Output

Code coverage results are output to a filter file. These files will appear as plain text unless you specify `-Djrockit.codecoverage.verbose` at startup, in which case, the format will be verbose.

Events Described

The file shows one event per line for six different event types, as identified by the following prefixes:

- `f`: information about the filter used
- `c`: class load
- `m`: method load (happens together at class load time)
- `g`: method code generated (happens at first excecution of the method)
- `l`: code generated for a line of source code (only reported for source lines actually containing code)
- `x`: line excecuted

Filter File Rules

The following rules apply to the filter file:

- Events are written to the file in exactly the order they occur in the JVM, with no sorting or buffering.
- The file is flushed and closed upon termination of the JVM.
- Abnormal exit may result in incomplete data at the end of the file.
- Every line begins with `<event>:<testid>`.

File Format

Table 1 describes the filter file format in both the plain text and verbose modes.

Table 1 Filter File Format by Event Type

Event Type	File Format
Filter	**Plain text:** `f:<testid>:<filterspec>:<timestamp>` **Verbose:** Same as plain text
Class load	**Plain text:** `c:<testid>:<classname>:<source filename>:<type>` Type can be one of the following • c - regular class • a - abstract class • i - interface **Verbose:** Same

Method load	**Plain text:**
	`m:<testid>:<classname>.<methodname><descriptor>:<methodid>`
	The method id is an unique identifier for this method valid for this run only.
	Verbose:
	`m:<testid>:<classname>.<methodname><descriptor>`

Method generated (excecuted) event	**Plain text:**
	`g:<testid>:<methodid>`
	Verbose:
	`g:<testid>:<classname>.<methodname><descriptor>`

Table 1 Filter File Format by Event Type

Line generated event	**Plain text:**
	`l:<testid>:<methodid>:<lineNo>,<lineNo>,...`
	Line numbers are in no specific order, so duplicates might exist.
	Verbose:
	`l:<testid>:<classname>.<methodname><descriptor>:<lineNo>`
	Only one line reported per event.

Line excecuted (covered) event	**Plain text:**
	`x:<testid>:<methodid>:<lineNo>,<lineNo>,...`
	Verbose:
	`x:<testid>:<classname>.<methodname><descriptor>:<lineNo>`
	• Only one line reported per event.
	• Line numbers are excactly the order they are excecuted, so duplicates might exist.