SNiFF+™

Version 3.2 for Unix and Windows

Java Debugger —sniffjdijdb

TakeFive software
An Integrated Systems Company

TakeFive Software, Inc.
Cupertino, CA
E-mail: info@takefive.com

TakeFive Software GmbH
5020 Salzburg, Austria
E-mail: info@takefive.co.at

Credits

The first version of Sniff was developed at the Informatics Laboratory of the Union Bank of Switzerland. Its development was considerably facilitated by the public domain application framework ET++.

Authors of the first version:

Walter Bischofberger (Sniff)

Erich Gamma (Sniffgdb)

Erich Gamma and André Weinand (ET++)

SNiFF+ Java Debugger — sniffjdijdb

Overview

This paper relates to the new SNiFF+ Java debugger, `sniffjdijdb`. For information on the `sniffjdb` debugger, also supplied in the SNiFF+ package, please refer to the *SNiFF+ Java Tutorial*.

- **System Requirements**

  The new `sniffjdijdb` debugger is supported only on **Windows** and **Solaris**.
  You have to use **Java** versions **1.2.x**, that is, as of version 1.2.1.

- Two debuggers — page 4

  Because of changes in the Sun Java debugger interface, SNiFF+ now provides 2 Java debugger executables. Which executable is used depends on the JDK version you are using and on your platform.

- Breakpoints — page 5

  It is possible to store breakpoints persistently between sessions. Breakpoints are then automatically set at the start of the next debugging session.

- Remote debugging of Java applications — page 5

  This section applies only to JDK versions 1.2 or higher. "Remote" in this context means using two separate Java Virtual Machines (JVM); one running the application being debugged (or target JVM), and one running the debugging front end. Whether or not these two JVMs are running on the same physical machine determines how you can connect to the target JVM. Typical examples for remote debugging are servlets and RMI applications.

- Other enhancements — page 6

  In the Variable Viewer, it is now also possible to edit string objects and array members (if these are simple data types) for on-the-fly tests.

- Command reference — page 7

  All the commands described can at present only be entered at the debugger command line. Only commands that are new, or that differ in functionality from those used in `sniffjdb` are described.

- Troubleshooting — page 8

  These are a few things you might need to be aware if you use `sniffjdijdb`.

Two debuggers

Because of changes in the Sun Java debugger interface, SNiFF+ now provides 2 Java debugger executables. Which executable is used depends on the JDK version you are using and on your platform.

This is automatically detected by SNiFF+, but can also be overridden as described under Overruling automatic debugger detection — page 4.

- `sniffjdb`

  This debugger is used for all JDKs up to and including JDK 1.2.0, as well as for platforms that the new Java Platform Debugger Architecture (JPDA) does not support (see below). Please see the *SNiFF+ Java Tutorial* for more information on `sniffjdb`.

- `sniffjdijdb`

  This debugger is based on the Sun Java Platform Debugger Architecture (JPDA), available as of **JDK 1.2.x** releases, and at present supported **only** for **Windows** and **Sun Solaris**.

- Note that, in the SNiFF+ Preferences and at the debugger command prompt you will see only `sniffjdb`. To see which debugger is currently being used, type

  ```
  about
  ```

  at the debugger command prompt.

Overruling automatic debugger detection

To overrule automatic debugger selection, create an environment variable called

```
SNIFF_USE_JDK12
```

before starting SNiFF+.

- Set the environment variable to `1` (one), to force selection of `sniffjdijdb.`
- Set the environment variable to any other value, to force selection of `sniffjdb.`
- If the environment variable does not exist, SNiFF+ defaults to automatic debugger selection according to JDK version and supported platform.

Breakpoints

It is possible to store breakpoints persistently between sessions. Breakpoints are then automatically set at the start of the next debugging session.

- **Storage/deletion of breakpoints**

  File and line positions of breakpoints are stored in the application class root directory, in a file called `breakpoints.ini`.

  Breakpoint locations are saved to disk (or deleted) when you exit the target application using the **exit** or **detach** commands.

  If you end a debug session using the debugger's **Close Tool** command (which also exits the target application), breakpoints are not written to disk, and previously cleared breakpoints are not deleted.

- **Activated and inactive breakpoints**

  If breakpoints are set in source files, and the target class(es) are not loaded (e.g. JVM is not running the application), breakpoints are generated as "inactive". Once the relevant classes have been loaded, the breakpoints are automatically activated.

- See also Command reference — page 7.

Remote debugging of Java applications

This section applies only to JDK versions **1.2** or higher. "Remote" in this context means using two separate Java Virtual Machines (JVM); one running the application being debugged (or target JVM), and one running the debugging front end. Whether or not these two JVMs are running on the same physical machine determines how you can connect to the target JVM. Typical examples for remote debugging are servlets and RMI applications.

Preparing the environment (Unix)

For remote debugging of a java application residing on a different machine, set the following environment variables on the remote machine:

- `SNIFF_DIR` points ro the remote machine's SNiFF+ installation
- `PATH` points to this `$SNIFF_DIR/bin`
- These environment variables must also be set in the `.cshrc` or equivalent file, because they have to be automatically set at the remote login of the debugger.

Socket connection

You would use this option for a connecting to a target JVM running on a remote machine.

- From the comand line, start the target application (`HelloWorld`) in debug mode as follows (the `start` command applies only to Windows):

  ```
  [start] java -Xdebug -Xnoagent -Xrunjdwp:transport=
  dt_socket,server=y,suspend=n -Djava.compiler=None
  HelloWorld
  ```

  A port number will be printed.
- The SNiFF+ Project with the application’s source code must be open.
- At the debugger command prompt, enter

  ```
  connecthost <hostname> <port>
  ```

  The `<port>` is obtained after starting the application as described above.

Shared memory connection

This option can be used for connection to a target JVM running locally.

- From the comand line, start the target application (`HelloWorld`) in debug mode as follows (the `start` command applies only to Windows):

  ```
  [start] java -Xdebug -Xnoagent -Xrunjdwp:transport=
  dt_shmem,server=y,address=mysharedmemory,
  suspend=n -Djava.compiler=None HelloWorld
  ```

  The parameter `"mysharedmemory"`, above, can be any string, and is used to identify the memory address.
- The SNiFF+ Project for the application’s source code must be open.
- At the debugger command prompt, enter

  ```
  connectsharedmem mysharedmemory
  ```

  The parameter `"mysharedmemory"`, above must be the one you entered as `address` when you started the application as described above.

Other enhancements

In the Variable Viewer, it is now also possible to edit string objects and array members (if these are simple data types) for on-the-fly tests.

Command reference

All the commands described can at present only be entered at the debugger command line. Only commands that are new, or that differ in functionality from those used in `sniffjdb` are described.

Breakpoints

- **breaks**

  Lists all internally registered breakpoints. If a class is not loaded, or the debugged JVM isn't yet started, the breakpoints are printed as “inactive” breakpoints.

- **clearall**

  Clears all breakpoints.

- **break <FileName.java>:<LineNumber>**

  break <ClassName>:<LineNumber> in file <FileName.java>
  Set a breakpoint in `<Filename.java>` at `<LineNumber>`.
  These variants of the can be used where file and class names are not the same, or if there are multiple classes in one file. The following variant can be used if file and class name are the same:
  break <ClassName>:<LineNumber>

- **break in <ClassName>.<MethodName>**

  break in <ClassName>.<MethodName><MethodSignature>
  Sets a breakpoint in the method `<MethodName>` of the class `<ClassName>`. The `<MethodSignature>` can be used to resolve ambigueties in method names. Class and file name have to be the same.

- **Note**: If the target JVM is not running, or the relevant class not loaded, an "inactive" breakpoint is created. The breakpoint is "activated" as soon as the class is loaded.

- See also Troubleshooting — page 8 for caveats.

Remote Debugging

- **connectHost <host> <port>**

  Connect to a remote (in the sense of a different machine) or local JVM by entering host name and port number. See also Socket connection — page 6.

- **connectSharedMem <shared memory name>**

  Connect to a local JVM by entering a shared memory name. See also Shared memory connection — page 6.

- **detach**

  Disconnects the debugger from a JVM running an application that is being debugged. The target JVM itself is not shut down and the application continues normal execution.

- **exit [remoteshutdown]**

  The `remoteshutdown` paramater allows you to detach from a remote JVM and to exit the debugger in one step.

Exceptions

- **catch**

  The `catch` command without arguments breaks at all thrown exceptions. If an exception class name is entered as an argument, all exceptions of the class, and its subclasses, are caught.

- **ignore**

  Ignores all thrown exceptions.

Execution

- **runsuspended**

  Starts up and immediately suspends execution in the JVM.

- **step [in | out | over]**

  Commands in parentheses are equivalents, as used also in `sniffjdb`.
  `step in` (=`step`) — execute next statement, step into methods.
  `step over` (=`next`) — execute the next statement, step over methods.
  `step out` (=`finish`) — execute until a breakpoint in another method is reached.

Troubleshooting

These are a few things you might need to be aware if you use `sniffjdijdb`.

- **breaks**

  The information in Breakpoints tab in the debugger may sometimes be inaccurate. If so, use the `breaks` command to list the existing breakpoints.

- **load <classname> / kill <thread(group)>**

  Make sure that the current thread (shown in the `sniffjdijdb` command prompt) is not running, that is, the thread must be suspended at a breakpoint, step, or exception event.

- **break/clear**

  Take care if you set a breakpoint when the corresponding class is not loaded. The following workarounds are possible to ensure valid breakpoints.

  - If you want to set a breakpoint at the start of a method, use **Break at** at the first line of the method, or type the command directly at the `sniffjdijdb` command line.
  - **Caution**: Constructors are named `<init>`, static initializers are named `<clinit>`. The button **Break in** does not specify the names correctly.

- If the class name differs from file name, enter

  ```
  break <FileName.java>:<LineNumber>
  ```

  instead of

  ```
  break <ClassName>:<LineNumber>
  ```

  This ensures that the breakpoint will be shown correctly in the source editor even if the class is not loaded.