Memory Profilers

In this section we'll discuss two commercial memory profilers that specialize in visualizing managed heaps and detecting memory leak sources. Because these tools are fairly complex, we will examine only a small subset of their features, and leave the rest to the reader to explore in the respective user guides.

ANTS Memory Profiler

RedGate's ANTS Memory Profiler specializes in heap snapshot analysis. Below we detail the process for using ANTS Memory Profiler to diagnose a memory leak. If you would like to follow through with these steps as you read this section, download a free 14-day trial of ANTS Memory Profiler from http://www.red-gate.com/products/dotnet-development/ants-memory-profiler/ and use it to profile your own application. In the directions and screenshots below, we used ANTS Memory Profiler 7.3, which was the latest version available at the time of writing.

You can use the FileExplorer.exe application from this chapter's source code folder to follow this demonstration—to make it leak memory, navigate the directory tree on the left to non-empty directories.

1. Run the application from within the profiler. (Similarly to CLR Profiler, ANTS supports attaching to a running process, starting from CLR 4.0.)
2. Use the Take Memory Snapshot button to capture an initial shapshot after the application has finished initializing. This snapshot is the baseline for subsequent performance investigations.
3. As the memory leak accumulates, take additional heap snapshots.
4. After the application terminates, compare snapshots (the baseline snapshot to the last snapshot, or intermediate snapshots among themselves) to understand which types of objects are growing in memory.
5. Focus on specific types using the Instance Categorizer to understand what kinds of references are retaining objects of the suspected types. (At this phase you are inspecting references between *types*—instances of type *A* referencing instances of type *B* will be grouped by type, as if *A* is referencing *B*.)
6. Explore individual instances of the suspected types using the Instance List. Identify several representative instances and use the Instance Retention Graph to determine why they are retained in memory. (At this phase you are inspecting references between individual *objects*, and can see why specific objects are not reclaimed by the GC.)
7. Return to the application's source code and modify it such that the leaking objects are no longer referenced by the problematic chain.

At the end of the analysis process, you should have a good grasp of why the heaviest objects in your application are not being reclaimed by the GC. There are many causes of memory leaks, and the real art is to quickly discern from the haystack that is a million-object heap the interesting representative objects and types that will lead to the major leak sources.

Figure 2-27 *shows the comparison view between two snapshots. The memory leak (in bytes) consists primarily of* `string` *objects. Focusing on the* `string` *type in the Instance Categorizer (in Figure 2-28) leads to the conclusion that there is an event that retains* `FileInformation` *instances in memory, and they in turn hold references to* `byte[]` *objects. Drilling down to inspect specific instances using the Instance Retention Graph (see Figure 2-29) points to the* `FileInformation.FileInformationNeedsRefresh` *static event as the source of the memory leak.*

Figure 2-27. Comparison between two heap snapshots. The total difference between them is +6.23MB, and the largest type currently held in memory is System.String

Figure 2-28. The strings are retained by arrays of strings, which are retained by FileInformation instances, which are in turn retained by an event (through System.EventHandler delegates)

Figure 2-29. The individual string we chose to examine is element 29 in an array of strings, held by the < FirstFewLines > k__BackingField field of a FileInformation object. Following the references points to the FileInformation.FileInformationNeedsRefresh static event

SciTech .NET Memory Profiler

The SciTech .NET Memory Profiler is another commercial tool focused on memory leak diagnostics. Whereas the general analysis flow is quite similar to ANTS Memory Profiler, this profiler can open *dump files*, which means you don't have to run it alongside the application and can use a crash dump generated when the CLR runs out of memory. This can be of paramount importance for diagnosing memory leaks *post mortem*, after the problem has already occurred in the production environment. You can download a 10-day evaluation version from http://memprofiler.com/download.aspx. In the directions and screenshots below, we used .NET Memory Profiler 4.0, which was the latest version available at the time of writing.

Note CLR Profiler can't open dump files directly, but there is an SOS.DLL command called `!TraverseHeap` that can generate a .log file in CLR Profiler's format. We discuss more examples of SOS.DLL commands in Chapters 3 and 4. In the meantime, Sasha Goldshtein's blog post at http://blog.sashag.net/archive/2008/04/08/next-generation-production-debugging-demo-2-and-demo-3.aspx provides an example of how to use SOS.DLL and CLR Profiler together.

To open a memory dump in .NET Memory Profiler, choose the File ➤ Import memory dump menu item and direct the profiler to the dump file. If you have several dump files, you can import them all into the analysis session and compare them as heap snapshots. The import process can be rather lengthy, especially where large heaps are involved; for faster analysis sessions, SciTech offers a separate tool, NmpCore.exe, which can be used to capture a heap session in a production environment instead of relying on a dump file.

Figure 2-30 shows the results of comparing two memory dumps in .NET Memory Profiler. It has immediately discovered suspicious objects held in memory directly by event handlers, and directs the analysis towards the `FileInformation` objects.

Figure 2-30. Initial analysis of two memory snapshots. The first column lists the number of live instances, whereas the third columns lists the number of bytes occupied by them. The main memory hog—string objects—are not visible because of the tooltip

Focusing on the `FileInformation` objects illustrates that there is a single root path from the `FileInformation.FileInformationNeedsRefresh` event handler to the selected `FileInformation` instances (see Figure 2-31) and a visualization of individual instances confirms the same reference chain we have seen previously with ANTS Memory Profiler.

***Figure 2-31**. FileInformation instances. The "Held bytes" column lists the amount of memory retained by each instance (its subtree in the object graph). On the right the shortest root path for the instance is shown*

We will not repeat here the instructions for using the rest of .NET Memory Profiler's functionality—you can find excellent tutorials on SciTech's website, http://memprofiler.com/OnlineDocs/. This tool concludes our survey of memory leak detection tools and techniques, which begun with CLR Profiler's heap dumps.