Effective Memory Utilization for Reliable, High-Performance Java™ Applications

BIO:

Daniel F. Savarese is a developer of enterprise Java applications. He has worked as a senior scientist at the California Institute of Technology and director of Java product development at ORO, Inc. Daniel is the original author of the Jakarta-ORO text processing packages and the NetComponents network protocol library. He is also a co-author of *How to Build a Beowulf*. Daniel can be reached at **dfs@savarese.org**.

Intro

Java has greatly simplified application development and deployment through its cross-platform virtual machine, object-oriented semantics, language-intrinsic thread model, and garbage-collected memory management. These features have contributed to greater programmer productivity and more reliable software. With distributed computing and Web application support in the form of Remote Method Invocation (RMI), servlets, JavaServer Pages (JSP), and Enterprise JavaBeans (EJB), Java has become a preferred implementation technology for enterprise computing and server-side components of Web applications. As Java becomes adopted increasingly more often for mission-critical tasks and interactive services requiring long up-times and high availability, such as e-commerce applications, it becomes even more important to ensure that your Java applications maintain a high level of performance.

Even though Java accelerates the software development process and increases application stability by relieving programmers from the error-prone task of explicit memory management required by languages such as C and C++, it still requires careful programming to maximize application performance and scalability. Server-side applications tend to be long-lived, allocating and releasing resources many times over their lifetimes. These resources include file descriptors, network and database connections, but primarily memory. Frequent memory allocation and deallocation can adversely affect application performance simply because doing so expends extra CPU cycles, sometimes making a system call to the operating system kernel and incurring a potentially adverse temporal overhead. The temporal costs of memory allocation can be especially detrimental to applications that require fast response times or need to scale to serve large numbers of clients. Java application components, in the form of servlets, JSPs, and EJBs, tend to be long-lived and are especially vulnerable to this problem because Java naturally encourages on-demand object instantiation over more complicated object creation strategies such as object pooling.

Traditional Memory-Related Problems in C/C++

Java memory management is inherently different from that of C/C++ and requires different optimization techniques, even though it is susceptible to some of the same general failings, including memory leaks. C and C++ require a much greater intimacy with memory management on the part of the programmer. Dynamically allocated memory is divided into two regions: the stack and the heap. The stack is a contiguous region of memory that is used to store activation records or function call frames. Every time you call a method or function, a call frame is pushed onto the stack, containing saved registers, function parameters, a return stack pointer, and local variables. This memory allocation is managed for you by the compiler and the operating system. The compiler generates the machine instructions that create the activation records and increment the stack pointer and the operating system handles stack overflows by either dynamically increasing the process stack size or aborting the program with a stack overflow error. Programmers have to make sure they do not create excessively large local objects, else they will consume all of the available stack space.

Large objects are typically allocated on the heap, a global pool of memory available to a program for dynamic object creation. The C `malloc` and `free` functions and the C++ `new` and `delete` operators allow programmers to explicitly allocate and release heap memory, referencing it with pointers. Where stack-allocated memory is automatically released immediately upon the return of a function, heap memory is not released until the programmer explicitly does so. If a programmer is not careful to release memory after it is no longer needed, the heap space available to a program can gradually decrease in size. This is commonly referred to as a memory leak. Eventually, if a program continues to leak memory, system performance will degrade, sometimes to unusable levels. To minimize this risk, C++ object classes can implement destructors, special methods that are invoked when an object is destroyed. Destructors are used to clean up after an object, usually releasing all memory that was dynamically allocated by the object. Stack-allocated local objects have their destructors automatically invoked at the end of their scope, such as at the end of a method call. The destructors of heap-allocated objects are invoked when their memory is explicitly deallocated with the `delete` operator. Destructors have to be implemented by the programmer, an often difficult task given the complicated links objects can form to other objects. For example, a hash table may not only contain a dynamically allocated array, but also many linked lists that contain objects with yet more pointers to other objects. The invocation of a single object's destructor can cause a chain of other destructors to be invoked. This is not only potentially expensive in terms of CPU cycles, but also creates extra opportunities for memory leaks to emerge.

The heap is divided into blocks that are parceled out to a program depending on the size of memory requested. Individual memory allocations must be contiguous. Therefore, when many small memory allocations have been made, it becomes more difficult to allocate a larger contiguous block. The heap is said to become fragmented when this happens. It is not always possible to rearrange the heap in order to coalesce small free blocks to satisfy a large memory request because C++ pointers reference physical memory. If the memory allocation system were to move allocated memory around, it would have to update the values of all the pointers to that memory, which is not possible in a C++ program. It is, however, possible to rearrange unallocated contiguous memory blocks in order to satisfy memory requests. The more fragmented memory becomes, the more difficult and time-consuming it is to do this.

New Memory-Related Problems in Java

Java solves the difficulties of managing memory use present in C++ by taking it largely out of the hands of programmers. Java does not allow programs to contain pointers to physical memory. Instead it only allows references to objects whose physical memory locations cannot be determined from within a program. A reference can be thought of as a handle to an object. Even though you can directly create objects with the Java `new` operator, only an indirect reference to the object is returned rather than a physical memory address. This allows the Java runtime system to rearrange memory at will without disrupting a program's data relationships. It also allows the Java runtime to track all references to objects and detect when an object is no longer reachable by a program accessible reference. If an object becomes unreachable, it becomes eligible to have its memory reclaimed. Unreachable objects are called garbage, which is why automatic memory reclamation is called garbage collection. A resulting difference between Java and C++ memory management is that in C++ both allocations and deallocations are synchronous, whereas in Java only allocations are synchronous because deallocations are handled asynchronously by a low-priority garbage-collection thread.

It might seem, then, that it is not possible to leak memory in Java because the garbage collector always releases memory when it is no longer in use. This is not quite the case because it is possible to retain references to unused memory in a program. It is common for this to happen when caching data. A frequently used idiom in Java programs is to store references to data inside of container objects, often using a Hashtable or HashMap. Even when the keys and values inside a container are no longer being used, so long as a reference to the container remains, those keys and values will never become eligible for garbage collection. Recognizing this problem, the Java 2 Platform introduced the java.lang.ref package, which includes support for weak references (the references discussed up to this point are strong references). In simple terms, an object that is only referenced by weak references may be garbage collected. Programs that leak memory by using a hash table for caching purposes can instead use a WeakHashMap to ensure that keys and their associated values are released when they are no longer needed. It is also possible to remove references to objects as soon as they are no longer needed by assigning a null value to their reference variables. This can make an object eligible for garbage collection earlier than it might have otherwise in the normal course of program execution. It can sometimes be helpful to apply this technique to long-running threads and methods.

Leaking memory is not the only pitfall that can plague Java programs. In Java, each thread is assigned its own private stack, similar in principle to a C runtime stack. However, Java stack memory does not have to be contiguous and can therefore be heap-allocated. Furthermore, all user-defined object class instantiation, as well as array creation, occurs on the heap. It is not possible to declare automatic stack-allocated variables of non-primitive type. Therefore, every time you create an object you affect the structure of the heap. Excessive object creation can lead to memory fragmentation. Unlike C++, the Java garbage collector can coalesce arbitrary free blocks in the heap. But this comes at a price. The garbage collector can stall your entire application if it needs to significantly rearrange the heap. This will happen when a program uses too much memory at one time. In many cases, a program only needs to use a fraction of the memory it has allocated at any given time. The garbage collector must also work harder when objects are constantly being allocated, used for a short time, and then unreferenced. Sometimes it can be more advantageous to pre-allocate a set of objects at application start time and reuse them in a programmer-managed object pool rather than repeatedly reallocating them from the heap. This can reduce the temporal cost of object creation and also reduce the frequency with which the garbage collection thread must interfere with your application.

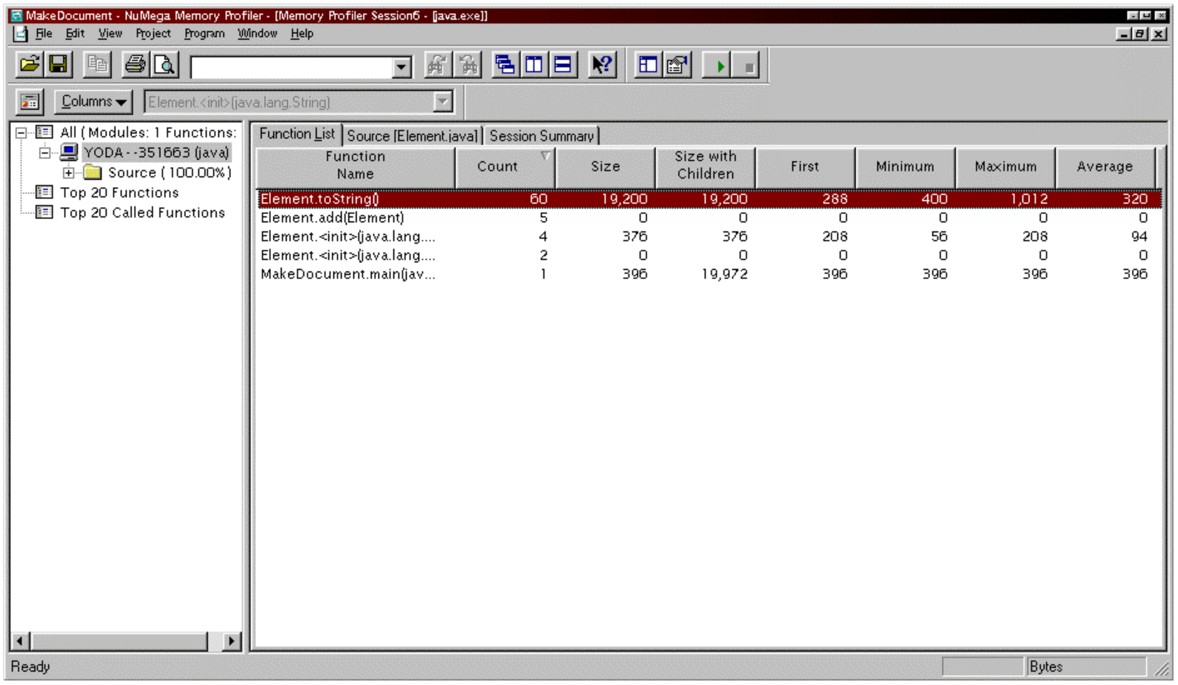
Optimizing Java Memory Utilization

Deciding what memory-use optimization strategies to apply in a program is impossible to do without knowledge of the program's memory-use characteristics. Different applications have different memory-use requirements and cannot necessarily be optimized in the same ways. The DevPartner Memory Profiler, part of the NuMega DevPartner Java Edition application profiling and analysis suite, provides the essential data for understanding how your Java applications use memory and its impact on performance. The Memory Profiler instruments compiled class files and executes them in a virtual machine of your choice. Sun JDKs 1.2 and higher are supported as well as Microsoft VM for Java 4.79.2339 and higher. The instrumentation code allows the Memory Profiler to record the memory use of every object and method in an application. The data for a run, called a session, can be saved for later analysis, comparison with results from running other versions of the code, or sharing with other developers working on the application. Multiple sessions can be grouped into a project so that you can preserve an entire record of the memory-use characteristics of successive versions of an application.

The DevPartner Memory Profiler is capable of profiling Java applications and components such as Servlets, JSPs and EJBs in a server environment, but for this illustration we will use the Memory Profiler’s GUI on a development workstation. Begin by opening the application from the File menu by selecting the class file containing the `main` method from the open file dialog. Finally run the application by selecting Start from the Program menu. Code listing 1 shows a sample program with some memory-use problems. Figures 1 and 2 show the results of a Memory Profiler run for the program.

Code Listing 1

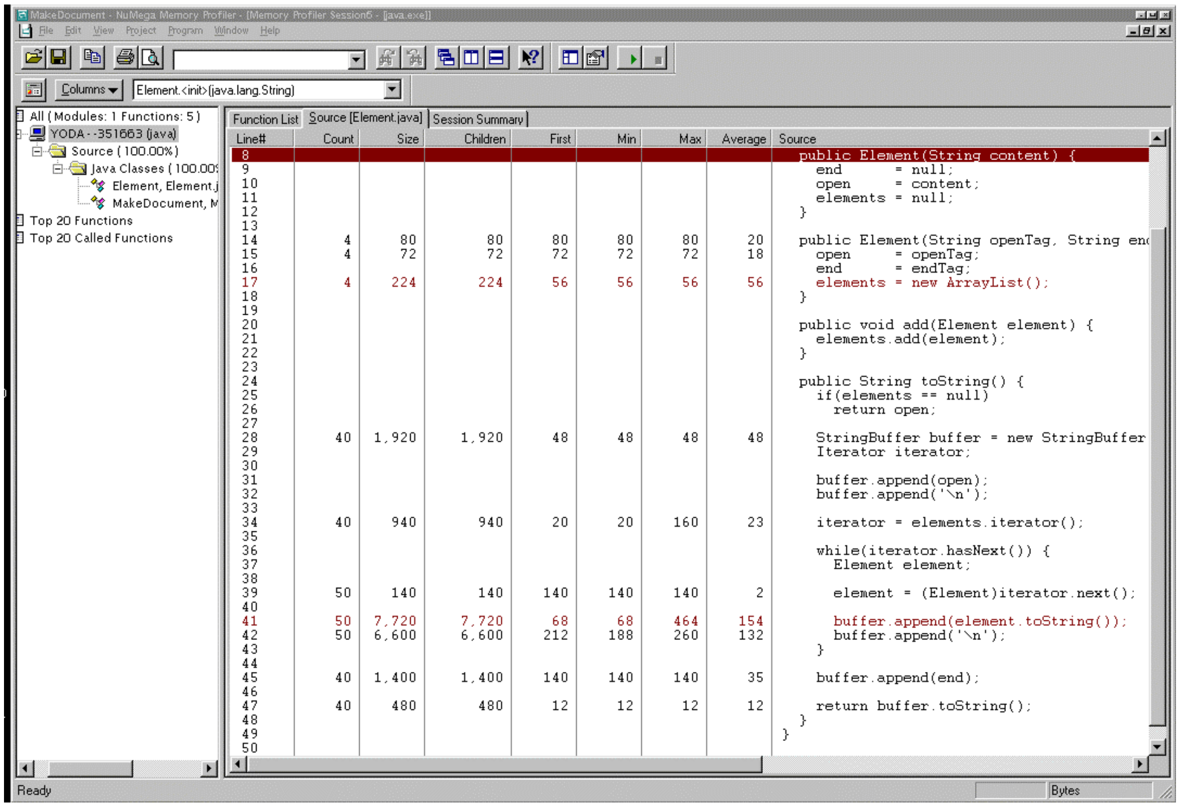
```
import java.util.*;

public class Element {
  private String open, end;
  private ArrayList elements;


  public Element(String content) {
    end      = null;
    open     = content;
    elements = null;
  }

  public Element(String openTag, String endTag) {
    open     = openTag;
    end      = endTag;
    elements = new ArrayList();
  }
  public void add(Element element) {
    elements.add(element);
  }

  public String toString() {
    if(elements == null)
      return open;

    StringBuffer buffer = new StringBuffer();
    Iterator iterator;

    buffer.append(open);
    buffer.append('\n');

    iterator = elements.iterator();

    while(iterator.hasNext()) {
      Element element;

      element = (Element)iterator.next();

      buffer.append(element.toString());
      buffer.append('\n');
    }

    buffer.append(end);

    return buffer.toString();
  }
}

public class MakeDocument {
  public static final void main(String[] args) {
    Element html  = new Element("<html>", "</html>");
    Element title = new Element("<title>", "</title>");
    Element body  = new Element("<body>", "</body>");
    Element paragraph = new Element("<p>", "</p>");

    title.add(new Element("The Lord of the Rings"));
    paragraph.add(new Element(
                      "When Mr. Bilbo Baggins of Bag End announced ..."));
    body.add(paragraph);
    html.add(title);
    html.add(body);

    /* Simulate repeated output generation. */
    for(int i=0; i < 10; ++i)
      System.out.println(html);
  }
}
```


MakeDocument - NuMega Memory Profiler - [Memory Profiler Session6 - [java.exe]]
File Edit View Project Program Window Help
Columns
Element.<init>(java.lang.String)
All (Modules: 1 Functions:
YODA - -351663 (java)
Source (100.00%)
Top 20 Functions
Top 20 Called Functions
Function List
Source [Element.java]
Session Summary
Ready
Bytes

Function Name	Count	Size	Size with Children	First	Minimum	Maximum	Average
Element.toString()	60	19,200	19,200	288	400	1,012	320
Element.add(Element)	5	0	0	0	0	0	0
Element.<init>(java.lang....	4	376	376	208	56	208	94
Element.<init>(java.lang....	2	0	0	0	0	0	0
MakeDocument.main(jav...	1	396	19,972	396	396	396	396

Figure 1

The example program in listing 1, although simplified, is representative of the types of operations frequently performed by servlets and JavaServer Pages. They often create HTML or XML output from document tree hierarchies. The example program uses a generic Element class to represent the elements of an HTML document, but a more complex HTML generation library would be used in reality. The fundamental approach to generating output is the same, though, using a depth -first tree traversal. Looking at figure 1, you will notice that the Memory Profiler has generated a listing of all the method calls made by the program, the number of times each was called, and the amount of memory used by each. When looking for memory problems, you will usually want to zero in on the methods that are called the most times or use the most memory. Frequently called methods deserve attention because they may unnecessarily allocate objects on each call, even if they do not use inordinate amounts of memory. Methods that use a lot of memory should be investigated because they may not need to use as much memory or they may be a source of memory leaks.



Line#	Count	Size	Children	First	Min	Max	Average	Source
8								public Element(String content) {
9								end = null;
10								open = content;
11								elements = null;
12								}
13								
14	4	80	80	80	80	80	20	public Element(String openTag, String en
15	4	72	72	72	72	72	18	open = openTag;
16								end = endTag;
17	4	224	224	56	56	56	56	elements = new ArrayList();
18								}
19								
20								public void add(Element element) {
21								elements.add(element);
22								}
23								
24								public String toString() {
25								if(elements == null)
26								return open;
27								
28	40	1,920	1,920	48	48	48	48	StringBuffer buffer = new StringBuffer
29								Iterator iterator;
30								
31								buffer.append(open);
32								buffer.append('\n');
33								
34	40	940	940	20	20	160	23	iterator = elements.iterator();
35								
36								while(iterator.hasNext()) {
37								Element element;
38								
39	50	140	140	140	140	140	2	element = (Element)iterator.next();
40								
41	50	7,720	7,720	68	68	464	154	buffer.append(element.toString());
42	50	6,600	6,600	212	188	260	132	buffer.append('\n');
43								}
44								
45	40	1,400	1,400	140	140	140	35	buffer.append(end);
46								
47	40	480	480	12	12	12	12	return buffer.toString();
48								}
49								}
50								

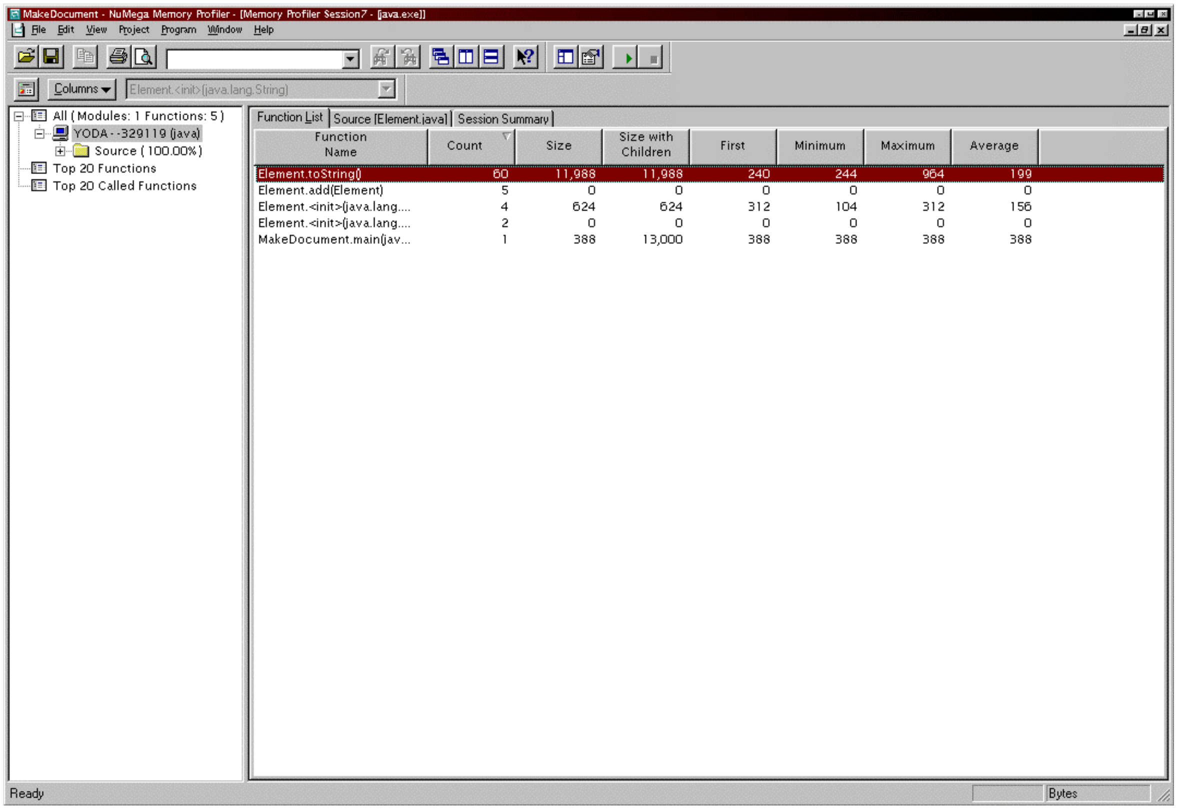
Figure 2

Figures 1 and 2 show us that Element.toString() is called the most times and uses the most memory, so it deserves close inspection. Figure 1 shows the Function List view, which summarizes method call memory use on a global basis for the entire life of the application. Figure 2 shows the Source Code view, which provides memory use statistics on a line by line basis for the application code. It is clear from looking at Element.toString() that it allocates a new StringBuffer on every call. Even though our example program only simulates the reuse of Element objects by repeatedly printing the same document, a real servlet would reuse its Element objects to dynamically create different documents. The frequent toString() calls would quickly create a lot of garbage for the garbage collector to clean up, interfering with program execution. It would be more appropriate to use a single StringBuffer member instance such as is done in a refined implementation of the Element class shown in listing 2.

Code Listing 2

```
import java.util.*;

public class Element {
  private String open, end;
  private ArrayList elements;
  private StringBuffer buffer;

  public Element(String content) {
    end      = null;
    open     = content;
    elements = null;
    buffer   = null;
  }

  public Element(String openTag, String endTag) {
    open     = openTag;
    end      = endTag;
    elements = new ArrayList();
    buffer   = new StringBuffer();
  }

  public void add(Element element) {
    elements.add(element);
  }

  public String toString() {
    if(elements == null)
      return open;

    Iterator iterator;

    buffer.setLength(0);
    buffer.append(open);
    buffer.append('\n');

    iterator = elements.iterator();

    while(iterator.hasNext()) {
      Element element;

      element = (Element)iterator.next();

      buffer.append(element.toString());
      buffer.append('\n');
    }

    buffer.append(end);

    return buffer.toString();
  }
}
```

In the new version of the toString() method, the StringBuffer is only allocated once, when the object is created, and reused every time the method is called. Running this version of the code through the Memory Profiler shows a 35% improvement in memory utilization. Figure 3 shows the results. Where the first version of the code used a total of 19,972 bytes, the second version uses 13,000 bytes. However, we find that the toString() method still dominates overall memory use. This is because we still associate a StringBuffer with every Element and convert each StringBuffer to a String before appending it to a parent Element's StringBuffer. This is clearly wasteful, but the observation allows us to refine our program even further until we reach the version in listing 3.


MakeDocument - NuMega Memory Profiler - [Memory Profiler Session7 - [java.exe]]
File Edit View Project Program Window Help
Columns
Element.<init>(java.lang.String)
All (Modules: 1 Functions: 5)
YODA - -329119 (java)
Source (100.00%)
Top 20 Functions
Top 20 Called Functions
Function List
Source [Element.java]
Session Summary
Ready
Bytes

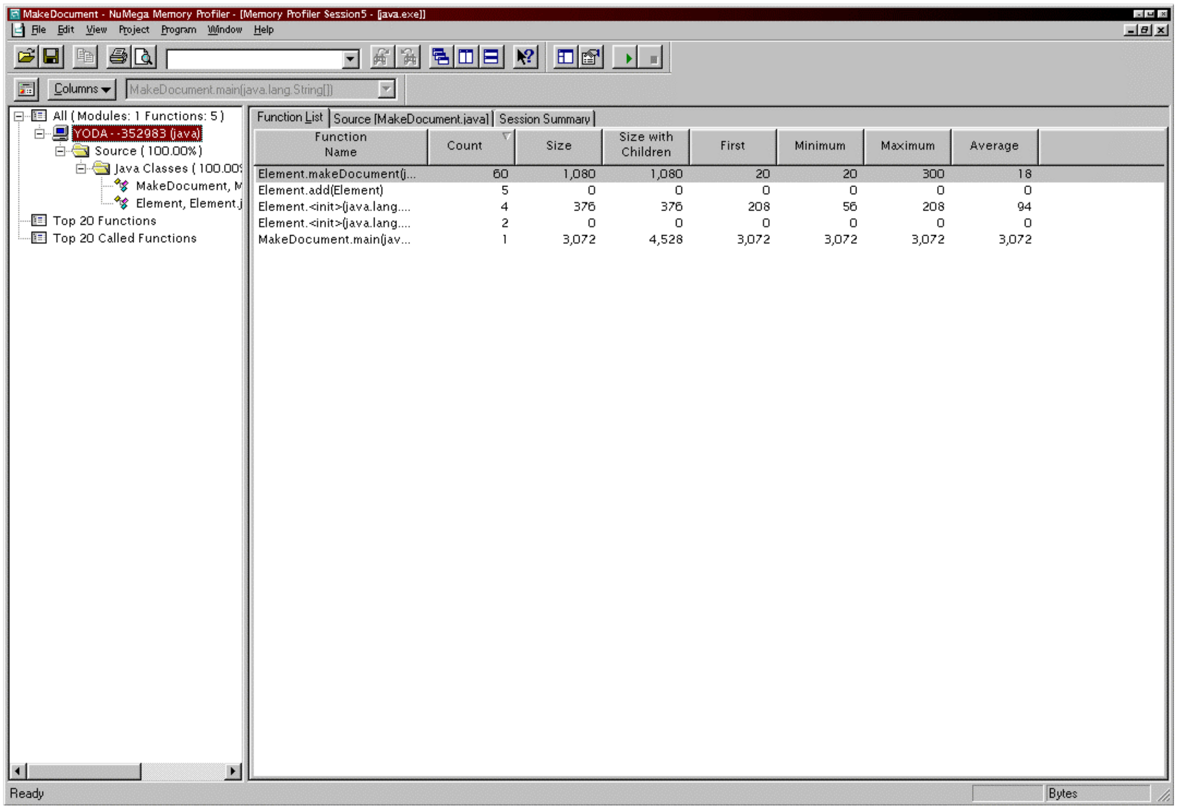
Function Name	Count	Size	Size with Children	First	Minimum	Maximum	Average
Element.toString()	60	11,988	11,988	240	244	964	199
Element.add(Element)	5	0	0	0	0	0	0
Element.<init>(java.lang....	4	624	624	312	104	312	156
Element.<init>(java.lang....	2	0	0	0	0	0	0
MakeDocument.main(jav...	1	388	13,000	388	388	388	388

Figure 3

Code Listing 3

```
import java.util.*;

public class Element {
  private String open, end;
  private ArrayList elements;

  public Element(String content) {
    end      = null;
    open     = content;
    elements = null;
  }

  public Element(String openTag, String endTag) {
    open     = openTag;
    end      = endTag;
    elements = new ArrayList();
  }

  public void add(Element element) {
    elements.add(element);
  }

  public void makeDocument(StringBuffer buffer) {

    if(elements == null) {
      buffer.append(open);
      return;
    }

    Iterator iterator;

    buffer.append(open);
    buffer.append('\n');

    iterator = elements.iterator();

    while(iterator.hasNext()) {
      Element element;

      element = (Element)iterator.next();

      element.makeDocument(buffer);
      buffer.append('\n');
    }

    buffer.append(end);
  }
}

public class MakeDocument {
  public static final void main(String[] args) {
    Element html  = new Element("<html>", "</html>");
    Element title = new Element("<title>", "</title>");
    Element body  = new Element("<body>", "</body>");
    Element paragraph = new Element("<p>", "</p>");
    StringBuffer buffer = new StringBuffer(128);

    title.add(new Element("The Lord of the Rings"));
    paragraph.add(new Element(
                  "When Mr. Bilbo Baggins of Bag End announced ..."));
    body.add(paragraph);
    html.add(title);
    html.add(body);

    /* Simulate repeated output generation. */
    for(int i=0; i < 10; ++i) {
      buffer.setLength(0);
      html.makeDocument(buffer);
      System.out.println(buffer);
    }
  }
}
```

In this last version of our program, we no longer use the toString() method and have optimized the code to use a single StringBuffer. The Memory Profiler revealed that by using multiple StringBuffer objects that are converted to Strings before appending them to yet another StringBuffer consumed extra memory. This time, the caller of the document-generating method makeDocument() must provide a StringBuffer that is used by all the other makeDocument() calls during the depth-first document traversal. Running this last version of the program through the Memory Profiler yields the results in figure 4, which indicate that memory use is now almost a factor of 3 smaller than in the second version of the code and almost a factor of 4.5 smaller than the original version of the code. Even though this example is artificial in certain respects, it demonstrates the dramatic improvements you can make to the memory-use of applications with just a few small changes. The key to knowing which changes to make is to use the Memory Profiler to understand how your programs use memory.


MakeDocument - NuMega Memory Profiler - [Memory Profiler Session5 - [java.exe]]
File Edit View Project Program Window Help
Columns
MakeDocument.main(java.lang.String[])
All (Modules: 1 Functions: 5)
YODA - -352983 (java)
Source (100.00%)
Java Classes (100.00%
MakeDocument, M
Element, Element.j
Top 20 Functions
Top 20 Called Functions
Function List
Source [MakeDocument.java]
Session Summary
Ready
Bytes

Function Name	Count	Size	Size with Children	First	Minimum	Maximum	Average
Element.makeDocument(j...	60	1,080	1,080	20	20	300	18
Element.add(Element)	5	0	0	0	0	0	0
Element.<init>(java.lang....	4	376	376	208	56	208	94
Element.<init>(java.lang....	2	0	0	0	0	0	0
MakeDocument.main(jav...	1	3,072	4,528	3,072	3,072	3,072	3,072

Figure 4

In some ways, efficient memory management techniques appear to go against the grain of object-oriented programming. The first version of our example program is probably the most natural to write since it restricts member variables to those that are absolutely necessary and cleanly encapsulates document generation in the toString() method. The last version appears more fragile. It mandates a contract that makeDocument() will only append to its StringBuffer parameter and not destroy any data that it already contains. The caller of the method must trust that the method will behave itself. In the original toString() method, there is no risk of overwriting data because it maintains complete control over result generation, returning an immutable String. In truth, both approaches are object-oriented and simply demonstrate an ease of use tradeoff driven by memory optimization requirements.

Java encourages the creation of object on demand because many of the core APIs do not have setter methods, only allowing their state to be set at the time of creation in a constructor. The tendency, then, is to create objects exactly when you need them rather than create a single object or pool of objects that you reuse. There are ways of avoiding this tendency and improving memory use as a result. In the case of immutable objects like the String class, try to use mutable objects like StringBuffers or a char array instead until you absolutely need to convert a result into a String. When you define your own classes, don't restrict object state initialization to the arguments passed to a constructor. Also provide a zero-argument constructor that creates reasonable default values and include setter methods or an init method to allow objects of that class to be reused. Take java.awt.Color for example. It only contains argument-based constructors and no setter methods. This means the only way you can use the Color class to any effect is to repeatedly instantiate it whenever you need a new color rather than simply alter the values of a pre-existing Color object with a setRGB() method. If you have to wrap primitive types, such as an int, in a class for the purpose of creating a reference to it, such as is necessary to store a value in a container class, define your own wrapper class instead of using java.lang.Integer. Only use the primitive wrapper classes in the java.lang package when you really need a read-only value or need to serialize an object. The wrapper classes do not define setter methods and can therefore not be reused. In addition, if you need to create many instances of a wrapper class like Integer, consider writing your algorithm to accept primitive types. Otherwise you run the risk of incurring unacceptably large memory costs. Let's say you want to store thousands of integers in a priority queue using an inverted heap structure. You could implement the data structure to store references to Integer types or primitive int types. Using primitive ints will save you from allocating an Integer every time a value is added to or removed from the priority queue, assuming the underlying representation of the queue is an array. This can save up to hours of CPU time when amortized over the lifetime of a long-running application.

You can often hide the details of memory optimization by using careful object-oriented design. Rather than allocating objects explicitly with the `new` operator, you can let a factory class create them for you. The factory class might implement an object pool, hiding the details of memory optimization from the core program logic. This technique is similar to both thread pooling and database connection pooling, which are applied to avoid the overhead of creating new threads and re-establishing database connections. Object pools can be implemented in different ways depending on the aspect of memory management that requires optimization in an application. When strictly seeking to avoid the latency of object creation in time-critical portions of code, an object pool may allocate many objects at application start-up to be able to satisfy object creation requests quickly. When the emphasis is more on keeping garbage collection to a minimum and avoiding the global cost of memory allocation across the entire application lifetime, rather than just optimizing localized allocations, a lazy allocation object pool can be implemented. A lazy allocation pool starts off with a few pre-allocated objects or none at all and gradually increases the number of objects in the pool as they are requested. This approach tries to keep memory use below an upper bound equal to the maximum number of objects concurrently in use by an application during its lifetime. Object pools themselves can be sources of memory leaks and are candidates for implementation with weak references. They also trade away ease of use by requiring programmers to indicate when they have finished using an object. In database connection pools this is easy to detect because close() will be called when a database connection is no longer needed. Generic object-pools will require that poolable objects implement an interface containing a release() method that is called when the object can be returned to the pool.

Summary

Maximizing the performance of your applications can be a daunting task. Complex server applications will contain many classes that can interact in unpredictable ways. Simply looking at source code and guessing at a classes' runtime behavior when looking for ways to improve performance can be a time-consuming task that leads down many blind alleys. Useful performance optimization requires empirical data that describes the runtime behavior of an entire application. Without performance data, you may inadvertently concentrate effort at optimizing object methods that are rarely used. Performance data keeps you from making guesses and guides you to improve the most important sections of your code. The DevPartner Memory Profiler provides the necessary data to learn how your applications use memory and where to concentrate your efforts in improving program performance. However, it does not do the entire job for you. It is up to you to decide what memory-use optimization techniques are most appropriate for your application. The Memory Profiler will point out object methods and lines of code that allocate a lot of memory or are invoked frequently over the life of a program. This can help you track down memory leaks and allocation hotspots that may be fragmenting the heap. Some algorithms are always going to be memory intensive and cannot be optimized beyond a certain limit. Others my not require a lot of memory, but can be made to run faster by using more memory. Some sorting algorithms fall into this category. Ultimately, the responsibility of analyzing your programs rests on your shoulders. Equipped with the right tools, you will be better able to reach the right conclusions and write more efficient programs.

Code Online:

All the code in this article is available for free at http://www.java-pro.com/. Use the Locator+ codes shown below to download the code. Free online registration is required. See the Online Resources box in “Letters to the Editor” for more information.

Article Title

Registered Locator+: **JPmmyy**
Premier Locator+: **JPmmyyxx**