Chapter 23

Memory Usage

The garbage collector is one of the most sophisticated pieces of the Microsoft .NET Framework architecture. Each time an application runs short of memory, the .NET runtime fires a garbage collection (GC) and examines all the objects that are in use, either because the application has a reference to them (these are known as *roots*) or because the object is referenced by another object in use. All the objects in the managed heap that aren't reachable are candidates for the GC. If the object has no finalizer, the object is immediately reclaimed and the heap is compacted; if the object has a Finalize method, the .NET runtime runs the method but doesn't reclaim the object until the next GC. (Actually, it might take more than one GC to collect an object with a finalizer because the object has been promoted to Generation 1 or Generation 2 and won't be reclaimed by a Generation-0 GC.) When all unreachable objects have been reclaimed, the managed heap can be compacted and the garbage collector can update all object references to point to the new position of the object in the heap.

This very concise description of what the garbage collector does explains why a GC is a relatively time-consuming operation. All .NET developers worth their salt should feel the responsibility of reducing the number of GCs that occur during the application's lifetime. We have illustrated several techniques that guarantee optimal usage of memory elsewhere in the book; in this chapter we focus on how you can help the garbage collector perform its job as efficiently as possible.

The simplest way to check whether your application uses memory efficiently is to have a look at a few counters of the .NET CLR Memory performance object:

- # Bytes in all Heaps (the sum of the four heaps used by the application, that is, Generation-0, Generation-1, Generation-2, and Large Object heaps)
- Gen 0 heap size, Gen 1 heap size, Gen 2 heap size, Large Object Heap size (the size of the four heaps used by the garbage collector)
- Gen 0 Collections, Gen 1 Collections, Gen 2 Collections (the number of GCs fired by the runtime)
- % Time in GC (the percentage of CPU time spent performing GCs)

Note One or more code examples in this chapter assume that the following namespaces have been imported by means of *Imports* (Visual Basic) or *using* (C#) statements:

```
System.IO
System.Runtime.InteropServices
```

23.1 Early creation of long-lasting objects

As a rule of thumb, objects that are meant to live for the entire duration of the application lifetime should be created earlier, if possible immediately after starting the application, and shouldn't be reallocated during the application lifetime.

Why: This simple technique has a twofold benefit. First, after a couple of garbage collections these objects will be promoted to Generation 2 and the garbage collector will ignore them during all Generation-0 and Generation-1 collections. (Having fewer objects to scan speeds up the garbage collection step.) Second, these objects will be allocated near the beginning of the managed heap and the .NET runtime won't need to move them in memory when the heap is compacted.

More details: For the aforementioned reason, fields in long-lasting objects shouldn't point to short-term objects; otherwise, these short-term objects will quickly be promoted to Generation 2 and, even if your code later sets them to *Nothing* (Visual Basic) or *null* (C#), they will take memory until the .NET runtime fires a Generation-2 collection.

23.2 Boxing

Ensure that you don't box value types without a good reason. You have a boxing operation when you assign a value type (for example, a number, a DataTime value, a structure) to an Object variable, pass it as an argument to a method that takes Object parameters, or assign it to an interface variable.

Why: Boxing consumes memory in the managed heap and indirectly causes the garbage collection process to run more frequently, thus slowing down the application.

See also: Read 8.8, 9.5, 9.6, and 14.19 for techniques that allow you to reduce the number of boxing operations in your applications.

23.3 Clearing object references

Don't explicitly set an object reference to null when you don't need it any longer inside a method.

Why: When optimizations are enabled, both Visual Basic and C# compilers can detect when an object variable isn't used later in the method and the garbage collector can reclaim the memory without the developer's assistance.

```
' [Visual Basic]
Sub PerformTask()
   Dim p As New Person
   ' Use the object.
   ...
   ' No need to explicitly set the variable to Nothing
End Sub
```

```
// [C#]
void PerformTask()
{
   Person p = new Person();
   // Use the object.
   ...
   // No need to explicitly set the variable to null
}
```

Exception: See rule 23.4 for an exception to this rule.

23.4 Clear object references explicitly in loops

Explicitly set an object reference to null if you are inside a loop and want the .NET runtime to collect the object before the loop ends.

Why: When compiling a loop, the Visual Basic and C# compilers can't automatically detect whether the variable is going to be used during subsequent iterations of the loop, and therefore the garbage collector can't automatically reclaim the memory used by the object. By clearing the object variable explicitly, you can help the garbage collector understand that the object can be reclaimed.

```
' [Visual Basic]
Sub PerformTask()
   Dim p As New Person
   ' Use the object inside the loop.
   For i As Integer = 1 To 100
      If i <= 50 Then
         ' Use the object only in the first 50 iterations.
         Console.WriteLine(p.CompleteName)
         ' Explicitly set the variable to Nothing after its last use.
         If i = 50 Then p = Nothing
      Else
         ' Do something else here, but don't use the p variable.
         ...
      End If
   Next
End Sub

// [C#]
void PerformTask()
{
   Person p = new Person();
   // Use the object inside the loop.
   for ( int i=1; i <= 100; i++ )
   {
      if ( i <= 50 )
      {
         // Use the object only in the first 50 iterations.
         Console.WriteLine(p.CompleteName);
         // Explicitly set the variable to null after its last use.
         if ( i == 50 )
            p = null;
      }
      else
      {
         // Do something else here, but don't use the p variable.
         ...
      }
   }
}
```

23.5 Keeping objects alive

Because the .NET runtime can collect objects even if they haven't been set explicitly to null (see rule 23.3), you must use the GC.KeepAlive method to keep an object alive if the object is supposed to continue to run until the current method completes.

More details: You might need to adopt this technique if the object uses a timer to perform background operations or if the object wraps an unmanaged resource that shouldn't be released until the method completes. You should notice, however, that in both these cases the object should implement the IDisposable interface and it should be used inside a *using* block (see rule 16.4) or a *Try...Finally* block (see rule 16.5). For this reason, in practice you rarely need to keep an object alive by means of the GC.KeepAlive method.

```
' [Visual Basic]
Sub PerformTask()
   Dim p As New Person
   ' Use the object.
   ...
   ' Keep the object alive until the end of the method.
   GC.KeepAlive(p)
End Sub

// [C#]
void PerformTask()
{
   Person p = new Person();
   // Use the object.
   ...
   // Keep the object alive until the end of the method.
   GC.KeepAlive(p);
}
```

23.6 Explicit garbage collections in server applications

Never explicitly invoke the GC.Collect method in server-side applications except for testing and debugging purposes. (Server-side applications include Web Forms, Web Services, Serviced Components, and Windows Services projects.)

C23621721.fm Page 284 Wednesday, January 12, 2005 10:52 PM

23.7 Explicit garbage collections in client applications

Use the GC.Collect method judiciously in client-side applications and only at the conclusion of a time-consuming operation such as file load or save. (Client-side applications include Console and Windows Forms projects.)

Never use the GC.Collect method in a class library or a control library unless you clearly document under which conditions a garbage collection is fired. In other words, induced garbage collections should always be under the control of the main application.

23.8 Waiting for finalizers after explicit garbage collections

Always use the GC.WaitForPendingFinalizers method after invoking the GC.Collect method.

Why: Finalizer methods run in a separate thread; by calling the GC.WaitForPendingFinalizers method, you ensure that all finalizers have completed and your objects are destroyed before continuing.

```
' [Visual Basic]
' The correct way to force a garbage collection
GC.Collect()
GC.WaitForPendingFinalizers()

// [C#]
// The correct way to force a garbage collection
GC.Collect();
GC.WaitForPendingFinalizers();
```

23.9 Large object heap fragmentation

Don't frequently create and destroy *large objects*, that is, objects that occupy more than 85,000 bytes. Consider splitting large objects into two or more smaller (regular) objects.

Why: Large objects are stored in a separate managed heap that is never compacted (because moving such large memory blocks would be too time-consuming). If you frequently allocate and free large objects, this separate heap can become fragmented, so even though you have enough free memory in the heap overall, you might run out of memory if there isn't a block large enough for the object you're trying to create.

More details: The majority of large objects are arrays. In some cases you can avoid the creation of a large object by using alternate but equivalent objects, such as an ArrayList or a BitArray. Likewise, you can often replace a large two-dimensional array with a jagged array whose constituent subarrays are smaller than 85,000 bytes.

You can control how the .NET runtime uses the large object heap by monitoring the Large Object Heap size counter of the .NET CLR Memory performance object.

C23621721.fm Page 285 Wednesday, January 12, 2005 10:52 PM

23.10 Compound finalizable objects

If you have a finalizable object that occupies a lot of memory, consider splitting the definition of the object into two separate types: one type that contains the unmanaged resource and the Finalize method, and another type that contains other variables.

Why: The garbage collector requires at least two collections to reclaim entirely the memory of an object that overrides the Finalize method. (Often, more collections are required because after the first collection the object is promoted to Generation 1.) By splitting the type into two classes, you enable the garbage collector to reclaim the memory of the nonfinalizable portion of the type during the first collection.

Example: The following code shows a naive implementation of a class that opens the system Clipboard and implements the Dispose-Finalize pattern to ensure that the Clipboard is closed when the object is disposed or when the garbage collector finalizes the object.

```
' [Visual Basic]
Public Class SimpleObject
   Implements IDisposable

   ' Windows API declarations
   Private Declare Function OpenClipboard Lib "User32" (ByVal hWnd As Integer) As Integer
   Private Declare Function CloseClipboard Lib "User32" () As Integer

   ' This array takes a lot of memory.
   Dim arr() As Integer

   Sub New(ByVal hWnd As Integer, ByVal elements As Integer)
      ReDim arr(elements - 1)
      OpenClipboard(hWnd)
   End Sub

   ' Close the Clipboard when the object is disposed.
   Public Sub Dispose() Implements System.IDisposable.Dispose
      CloseClipboard()
      GC.SuppressFinalize(Me)
   End Sub

   ' Close the Clipboard when the object is finalized.
   Protected Overrides Sub Finalize()
      Dispose()
   End Sub
End Class

// [C#]
public class SimpleObject : IDisposable
{
   // Windows API declarations
   [DllImport("User32")]
   private static extern int OpenClipboard (int hWnd);
   [DllImport("User32")]
   private static extern int CloseClipboard ();

   // This array takes a lot of memory.
   int[] arr;

   public SimpleObject(int hWnd, int elements)
   {
      arr = new int[elements];
      OpenClipboard(hWnd);
   }

   // Close the Clipboard when the object is disposed.
   public void Dispose()
   {
      CloseClipboard();
      GC.SuppressFinalize(this);
   }

   // Close the Clipboard when the object is finalized.
   ~SimpleObject()
   {
      Dispose();
   }
}
```

In the following improved version, we have split the previous type into two disposable classes, with only one of them containing the Finalize method. This new version can run remarkably faster than the previous one if many objects are created and destroyed frequently.

```
' [Visual Basic]
Public Class CompoundObject
   Implements IDisposable

   ' This array takes a lot of memory.
   Dim arr() As Integer
   ' An instance of the inner (finalizable) object
   Dim clipWrapper As ClipboardWrapper

   Sub New(ByVal hWnd As Integer, ByVal elements As Integer)
      ReDim arr(elements - 1)
      clipWrapper = New ClipboardWrapper(hWnd)
   End Sub

   ' Dispose of the inner object.
   Public Sub Dispose() Implements System.IDisposable.Dispose
      clipWrapper.Dispose()
   End Sub

   ' The inner type that wraps the Finalize method
   Private Class ClipboardWrapper
      Implements IDisposable

      ' Windows API declarations
      Private Declare Function OpenClipboard Lib "User32" (ByVal hWnd As Integer) _
         As Integer
      Private Declare Function CloseClipboard Lib "User32" () As Integer

      Sub New(ByVal hWnd As Integer)
         OpenClipboard(hWnd)
      End Sub
      Public Sub Dispose() Implements System.IDisposable.Dispose
         CloseClipboard()
         GC.SuppressFinalize(Me)
      End Sub

      Protected Overrides Sub Finalize()
         Dispose()
      End Sub
   End Class
End Class

// [C#]
public class CompoundObject : IDisposable
{
   // This array takes a lot of memory.
   int[] arr = null;
   // An instance of the inner (finalizable) object
   ClipboardWrapper clipWrapper = null;

   public CompoundObject(int hWnd, int elements)
   {
      arr = new int[elements];
      clipWrapper = new ClipboardWrapper(hWnd);
   }

   // Close the Clipboard when the object is disposed.
   public void Dispose()
   {
      clipWrapper.Dispose();
   }

   // The inner type that wraps the Finalize method.
   private class ClipboardWrapper : IDisposable
   {
      // Windows API declarations
      [DllImport("User32")]
      private static extern int OpenClipboard (int hWnd);
      [DllImport("User32")]
      private static extern int CloseClipboard ();

      public ClipboardWrapper(int hWnd)
      {
         OpenClipboard(hWnd);
      }

      public void Dispose()
      {
         CloseClipboard();
         GC.SuppressFinalize(this);
      }

      ~ClipboardWrapper()
      {
         Dispose();
      }
   }
}
```

23.11 WeakReference objects

Always wrap WeakReference objects in a custom type so other developers can use them safely and in a robust way.

More details: Weak references enable you to mantain a reference to another object (typically, an object that consumes a lot of memory) but don't prevent that object from being reclaimed when a garbage collection occurs. Weak references are therefore very useful for creating a cache of large objects: the object stays in memory only until there is a need to reclaim them. Wrapping these weak references in a custom type enables clients to use them in a more robust way.

Example: The following CachedTextFile class wraps a WeakReference object and enables clients to read the contents of a text file (which is supposed to be immutable during the application's lifetime) without having to reload it from the file each time the application uses it.

```
' [Visual Basic]
Public Class CachedTextFile
   ' The name of the file cached
   Public ReadOnly Filename As String
   ' A weak reference to the string that contains the text
   Dim wrText As WeakReference

   ' The constructor takes the name of the file to read.
   Sub New(ByVal fileName As String)
      Me.Filename = fileName
   End Sub

   ' Read the contents of the file.
   Private Function ReadFile() As String
      Dim sr As New StreamReader(Me.Filename)
      Dim text As String = sr.ReadToEnd()
      sr.Close()
      ' Create a weak reference to the return value.
      wrText = New WeakReference(text)
      Return text
   End Function

   ' Return the textual content of the file.
   Public Function GetText() As String
      Dim text As Object
      ' Retrieve the target of the weak reference.
      If Not (wrText Is Nothing) Then text = wrText.Target
      If Not (text Is Nothing) Then
         ' If non-null, the data is still in the cache.
         Return text.ToString()
      Else
         ' Otherwise, read it and put it in the cache again.
         Return ReadFile()
      End If
   End Function
End Class
```

```
// [C#]
public class CachedTextFile
{
   // The name of the file cached
   public readonly string Filename;
   // A weak reference to the string that contains the text
   WeakReference wrText;

   // The constructor takes the name of the file to read.
   public CachedTextFile (string fileName)
   {
      this.Filename = fileName;
   }

   // Read the contents of the file.
   private string ReadFile()
   {
      StreamReader sr = new StreamReader(Me.Filename);
      string text = sr.ReadToEnd();
      sr.Close();
      // Create a weak reference to the return value.
      WeakReference wrText = new WeakReference(text);
      return text;
   }

   // Return the textual content of the file.
   public string GetText()
   {
      object text = null;
      // Retrieve the target of the weak reference.
      if ( wrText != null )
         text = wrText.Target;
      if ( text != null )
      {
         // If non-null, the data is still in the cache.
         return text.ToString();
      }
      else
      {
         // Otherwise, read it and put it in the cache again.
         return ReadFile();
      }
   }
}
```

Here's how you can use this class.

```
' [Visual Basic]
' Read and cache the contents of the "c:\alargefile.txt" file.
Dim cf As New CachedTextFile("c:\alargefile.txt")
Console.WriteLine(cf.GetText())
...
' Uncomment next line to force a garbage collection.
' GC.Collect(): GC.WaitForPendingFinalizers()
...
```

C23621721.fm Page 290 Wednesday, January 12, 2005 10:52 PM

```
' Read the contents again some time later. (No disk access
' is performed, unless a GC has occurred in the meantime.)
Console.WriteLine(cf.GetText())

// [C#]
// Read and cache the contents of the "c:\alargefile.txt" file.
CachedTextFile cf = new CachedTextFile(@"c:\alargefile.txt");
Console.WriteLine(cf.GetText());
...
// Uncomment next line to force a garbage collection.
// GC.Collect(); GC.WaitForPendingFinalizers();
...
// Read the contents again some time later. (No disk access
// is performed, unless a GC has occurred in the meantime.)
Console.WriteLine(cf.GetText());
```

By examining the CachedTextFile class, you can easily prove that in most cases the file contents can be retrieved through the weak reference and that the disk isn't accessed again. By uncommenting the statement in the middle of the previous code snippet, you force a garbage collection, in which case the internal WeakReference object won't keep the String object alive and the code in the class will read the file again. The key point in this technique is that the client code doesn't know whether the cache object is used or not: the client just uses the CachedTextFile object as a "black box" that deals with large text files in an optimized way.

Remember that you create a weak reference by passing your object (a string, in this example) to the constructor of a System.WeakReference object. However, if the object is also referenced by a regular, nonweak reference, it survives any intervening garbage collection. In our example, the code using the CachedTextFile class should not store the return value of the GetText method in a string variable because that would prevent the string from being garbage collected. The following code snippet illustrates the incorrect way to use the CachedTextFile class:

```
' [Visual Basic]
Dim cf As New CachedTextFile("c:\alargefile.txt")
Dim text As String = cf.GetText()

// [C#]
CachedTextFile cf = new CachedTextFile(@"c:\alargefile.txt");
string text = cf.GetText();
```

23.12 Garbage collector settings

Carefully consider using nondefault settings for the garbage collector.

More details: The .NET Framework comes with two different garbage collectors: the workstation garbage collector and the server garbage collector. These garbage collectors are implemented in two different DLLs: the workstation garbage collector (in mscorwks.dll) is used on single-processor systems, whereas the server garbage collector (in mscorsvr.dll) is used on multiprocessor systems.

The server garbage collector pauses the application during GCs and uses one thread and one managed heap for each CPU; these settings guarantee the best overall throughput, even if an individual client might be slowed down if a GC occurs during a method call. The workstation garbage collector minimizes pauses by running the garbage collector concurrently with the application's worker threads.

On a single-CPU computer, both the server and the workstation garbage collectors behave in the same way and perform collections concurrently. For particular applications, however, you might slightly improve the overall performance by suppressing concurrency at the expense of longer delays in user interface operations. You can do this by using the <gcConcurrent> element in machine.config or the application's config file.

```
<configuration>
   <runtime>
      <gcConcurrent enabled="false" />
   </runtime>
</configuration>
```

You can also change the garbage collector behavior at the machine level by means of the Microsoft .NET Framework version 1.1 Configuration utility, which you can run from the Windows Administrative Tools menu.

By default, noninteractive .NET applications (such as a Windows service or an application that uses remoting) running on a multiprocessor system use the workstation GC and might perform poorly. You can force such applications to use the server GC by adding a <gcServer> element in the configuration file:

```
<configuration>
   <runtime>
      <gcServer enabled="true" />
   </runtime>
</configuration>
```

See also: For more information about this issue, read the MSDN Knowledge Base article at *http://support.microsoft.com/default.aspx?scid=kb;en-us;840523*. To configure an ASP.NET application to use the workstation GC, read *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt06.asp*.