Username: Pralay Patoria **Book:** Pro .NET Performance. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Interacting with the Garbage Collector

So far, we viewed our application as a passive participant in the GC story. We delved into the implementation of the garbage collector, and reviewed significant optimizations, all performed automatically and requiring little or no involvement from our part. In this section, we will examine the available means of active interaction with the garbage collector in order to tune the performance of our applications and receive diagnostic information that is unavailable otherwise.

The System.GC Class

The System.GC class is the primary point of interaction with the .NET garbage collector from managed code. It contains a set of methods that control the garbage collector's behavior and obtain diagnostic information regarding its work so far.

Diagnostic Methods

The diagnostic methods of the System.GC class provide information on the garbage collector's work. They are meant for use in diagnostic or debugging scenarios; do not rely on them to make decisions at normal run time. The information set returned by some of these diagnostic methods is available as performance counters, under the *.NET CLR Memory* performance category.

- The `GC.CollectionCount` method returns the number of garbage collections of the specified generation since application startup. It can be used to determine whether a collection of the specified generation occurred between two points in time.
- The GC.GetTotalMemory method returns the total memory consumed by the garbage collected heap, in bytes. If its Boolean parameter is true, a full garbage collection is performed before returning the result, to ensure that only memory that could not be reclaimed is accounted for.
- The GC.GetGeneration method returns the generation to which a specific object belongs. Note that under the current CLR implementation, objects are not guaranteed to be promoted between generations when a garbage collection occurs.

Notifications

Introduced in .NET 3.5 SP1, the GC notifications API provides applications an opportunity to learn in advance that a full garbage collection is imminent. The API is available only when using non-concurrent GC, and is aimed at applications that have long GC pause times and want to redistribute work or notify their execution environment when they sense a pause approaching.

First, an application interested in GC notifications calls the `GC.RegisterForFullGCNotification` method, and passes to it two thresholds (numbers between 1 and 99). These thresholds indicate how early the application wants the notification based on thresholds in generation 2 and the large object heap. In a nutshell, larger thresholds make sure that you receive a notification but the actual collection may be delayed for a while, whereas smaller thresholds risk not receiving a notification at all because the collection was too close to the notification trigger.

Next, the application uses the `GC.WaitForFullGCApproach` method to synchronously block until the notification occurs, makes all the preparations for GC, and then calls `GC.WaitForFullGCComplete` to synchronously block until the GC completes. Because these APIs are synchronous, you might want to call them on a background thread and raise events to your main processing code, like so:

```
public class GCWatcher {
  private Thread watcherThread;

  public event EventHandler GCApproaches;
  public event EventHandler GCComplete;

  public void Watch() {
    GC.RegisterForFullGCNotification(50, 50);
    watcherThread = new Thread(() => {
     while (true) {
     GCNotificationStatus status = GC.WaitForFullGCApproach();
     //Omitted error handling code here
     if (GCApproaches != null) {
     GCApproaches(this, EventArgs.Empty);
     }
     status = GC.WaitForFullGCComplete();
     //Omitted error handling code here
     if (GCComplete != null) {
     GCComplete(this, EventArgs.Empty);
     } }
    });
```

```
    watcherThread.IsBackground = true;
    watcherThread.Start();
  }

  public void Cancel() {
    GC.CancelFullGCNotification();
    watcherThread.Join();
  }
}
```

For more information and a complete example using the GC notifications API to redistribute server load, consult the MSDN documentation at http://msdn.microsoft.com/en-us/library/cc713687.aspx.

Control Methods

The `GC.Collect` method instructs the garbage collector to perform a garbage collection of the specified generation (including all younger generations). Starting with .NET 3.5 (and also available in .NET 2.0 SP1 and .NET 3.0 SP1), the `GC.Collect` method is overloaded with a parameter of the enumerated type GCCollectionMode. This enumeration has the following possible values:

- When using `GCCollectionMode.Forced`, the garbage collector performs the collection immediately and synchronously with the current thread. When the method returns, it is guaranteed that the garbage collection completed.
- When using `GCCollectionMode.Optimized`, the garbage collector can decide whether a collection will be productive at this point in time or not. The ultimate decision whether a collection will be performed is delayed to run time, and is not guaranteed by the implementation. This the recommended mode to use if you are trying to assist the garbage collector by providing hints as to when a collection might be beneficial. In diagnostic scenarios, or when you want to force a full garbage collection to reclaim specific objects that you are interested in, you should use `GCCollectionMode.Forced` instead.
- As of CLR 4.5, using `GCCollectionMode.Default` is equivalent to `GCCollectionMode.Forced`.

Forcing the garbage collector to perform a collection is not a common task. The optimizations described in this chapter are largely based on dynamic tuning and heuristics that have been thoroughly tested for various application scenarios. It is not recommended to force a garbage collection or even to recommend a garbage collection (using `GCCollectionMode.Optimized`) in non-exceptional circumstances. With that said, we can outline several scenarios that warrant the careful consideration of forcing a garbage collection:

- When an infrequent repeatable action that required a large amount of long-lived memory completes, this memory becomes eligible for garbage collection. If the application does not cause full garbage collections frequently, this memory might remain in generation 2 (or in the LOH) for a long time before being reclaimed. In this scenario, it makes sense to force a garbage collection when the memory is known to be unreferenced so that it does not occupy unnecessary space in the working set or in the page file.
- When using the low latency GC mode, it is advisable to force garbage collections at safe points when it is known that the time-sensitive work has idled and can afford pausing to perform a collection. Staying in low latency mode without performing a collection for a long period of time might result in out of memory conditions. Generally speaking, if the application is sensitive to garbage collection timing, it's reasonable to force a collection during idle times to influence a biased redistribution of garbage collection overhead into the idle run time regions.
- When non-deterministic finalization is used to reclaim unmanaged resources, it is often desirable to block until all such resources have been reclaimed. This can be accomplished by following a `GC.Collect` call with a `GC.WaitForPendingFinalizers` call. Deterministic finalization is always preferred in these scenarios, but often we do not have control over the internal classes that actually perform the finalization work.

Note As of .NET 4.5, the `GC.Collect` method has an additional overload with a trailing Boolean parameter: `GC.Collect`(int generation, GCCollectionMode mode, **`bool blocking`**). This parameter controls whether a blocking garbage collection is required (the default) or whether the request can be satisfied asynchronously by launching a background garbage collection.

The other control methods are the following:

- The `GC.AddMemoryPressure` and `GC.RemoveMemoryPressure` methods can be used to notify the garbage collector about unmanaged memory allocations taking place in the current process. Adding memory pressure indicates to the garbage collector that the specified amount of unmanaged memory has been allocated. The garbage collector may use this information for tuning the aggressiveness and frequency of garbage collections, or ignore it altogether. When the unmanaged allocation is known to be reclaimed, notify the garbage collector that the memory pressure can be removed.
- The `GC.WaitForPendingFinalizers` method blocks the current thread until all finalizers have finished executing. This method should be used with caution because it might introduce deadlocks; for example, if the main thread blocks inside `GC.WaitForPendingFinalizers` while holding some lock, and one of the active finalizers requires that same lock, a deadlock occurs. Because `GC.WaitForPendingFinalizers` does not accept a timeout parameter, the locking code inside the finalizer must use timeouts for graceful error handling.
- The `GC.SuppressFinalize` and `GC.ReRegisterForFinalize` methods are used in conjunction with finalization and resurrection features. They are discussed in the finalization section of this chapter.
- Starting with .NET 3.5 (and also available in .NET 2.0 SP1 and .NET 3.0 SP1), another interface to the garbage collector is provided

by the GCSettings class that was discussed earlier, for controlling the GC flavor and switching to low latency GC mode.

For other methods and properties of the System.GC class that were not mentioned in this section, refer to the MSDN documentation.

Interacting with the GC using CLR Hosting

In the preceding section, we have examined the diagnostic and control methods available for interacting with the garbage collector from managed code. However, the degree of control offered by these methods leaves much to be desired; additionally, there is no notification mechanism available to let the application know that a garbage collection occurs.

These deficiencies cannot be addressed by managed code alone, and require *CLR hosting* to further control the interaction with the garbage collector. CLR hosting offers multiple mechanisms for controlling .NET memory management:

- The `IHostMemoryManager` interface and its associated IHostMalloc interface provide callbacks which the CLR uses to allocate virtual memory for GC segments, to receive a notification when memory is low, to perform non-GC memory allocations (e.g. for JITted code) and to estimate the available amount of memory. For example, this interface can be used to ensure that all CLR memory allocation requests are satisfied from physical memory that cannot be paged out to disk. This is the essence of the Non-Paged CLR Host open source project (http://nonpagedclrhost.codeplex.com/, 2008).
- The `ICLRGCManager` interface provides methods to control the garbage collector and obtain statistics regarding its operation. It can be used to initiate a garbage collection from the host code, to retrieve statistics (that are also available as performance counters under the *.NET CLR Memory* performance category) and to initialize GC startup limits including the GC segment size and the maximum size of generation 0.
- The `IHostGCManager` interface provides methods to receive a notification when a garbage collection is starting or ending, and when a thread is suspended so that a garbage collection can proceed.

Below is a small code extract from the Non-Paged CLR Host open source project, which shows how the CLR host customizes segment allocations requested by the CLR and makes sure any committed pages are locked into physical memory:

```
HRESULT __stdcall HostControl::VirtualAlloc(
  void* pAddress, SIZE_T dwSize, DWORD flAllocationType,
  DWORD flProtect, EMemoryCriticalLevel dwCriticalLevel, void** ppMem) {

  *ppMem = VirtualAlloc(pAddress, dwSize, flAllocationType, flProtect);
  if (*ppMem == NULL) {
    return E_OUTOFMEMORY;
  }
  if (flAllocationType & MEM_COMMIT) {
    VirtualLock(*ppMem, dwSize);
    return S_OK;
  }
}

HRESULT __stdcall HostControl::VirtualFree(
  LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType) {

  VirtualUnlock(lpAddress, dwSize);
  if (FALSE == VirtualFree(lpAddress, dwSize, dwFreeType)) {
    return E_FAIL;
  }
  return S_OK;
}
```

For information about additional GC-related CLR hosting interfaces, including `IGCHost`, `IGCHostControl` and `IGCThreadControl`, refer to the MSDN documentation.

GC Triggers

We have seen several reasons for the GC to fire, but never listed them all in one place. Below are the triggers used by the CLR to determine whether it's necessary to perform a GC, listed in order of likelihood:

1. Generation 0 fills. This happens all the time as the application allocates new objects in the small object heap.
2. The large object heap reaches a threshold. This happens as the application allocates large objects.
3. The application calls `GC.Collect` explicitly.
4. The operating system reports that it is low on memory. The CLR uses the memory resource notification Win32 APIs to monitor

system memory utilization and be a good citizen in case resources are running low for the entire system.

5. An AppDomain is unloaded.

6. The process (or the CLR) is shutting down. This is a degenerate garbage collection—nothing is considered a root, objects are not promoted, and the heap is not compacted. The primary reason for this collection is to run finalizers.