Garbage Collection (GC)

Conditions for a garbage collection

Garbage collection occurs when one of the following conditions is true:

1. The system has low physical memory. This is detected by either the low memory notification from the OS or low memory as indicated by the host.

2. The memory that's used by allocated objects on the managed heap surpasses an acceptable threshold. This threshold is continuously adjusted as the process runs.

3. The `GC.Collect()` method is called. In almost all cases, you don't have to call this method, because the garbage collector runs continuously. This method is primarily used for unique situations and testing.

Generations

The GC algorithm is based on several considerations:

- It's faster to compact the memory for a portion of the managed heap than for the entire managed heap.
- Newer objects have shorter lifetimes and older objects have longer lifetimes.
- Newer objects tend to be related to each other and accessed by the application around the same time.

Garbage collection primarily occurs with the reclamation of short-lived objects. To optimize the performance of the garbage collector, the managed heap is divided into three generations, 0, 1, and 2, so it can handle long-lived and short-lived objects separately. The garbage collector stores new objects in generation 0. Objects created early in the application's lifetime that survive collections are promoted and stored in generations 1 and 2. Because it's faster to compact a portion of the managed heap than the entire heap, this scheme allows the garbage collector to release the memory in a specific generation rather than release the memory for the entire managed heap each time it performs a collection.

- **Generation 0.**

  This is the youngest generation and contains short-lived objects. An example of a short-lived object is a temporary variable. Garbage collection occurs most frequently in this generation.

  Newly allocated objects form a new generation of objects and are implicitly generation 0 collections. However, if they are large objects, they go on the large object heap (LOH), which is sometimes referred to as generation 3. Generation 3 is a physical generation that's logically collected as part of generation 2.

  Most objects are reclaimed for garbage collection in generation 0 and don't survive to the next generation.

If an application attempts to create a new object when generation 0 is full, the garbage collector performs a collection in an attempt to free address space for the object. The garbage collector starts by examining the objects in generation 0 rather than all objects in the managed heap. A collection of generation 0 alone often reclaims enough memory to enable the application to continue creating new objects.

- **Generation 1.**

  This generation contains short-lived objects and serves as a buffer between short-lived objects and long-lived objects.

  After the garbage collector performs a collection of generation 0, it compacts the memory for the reachable objects and promotes them to generation 1. Because objects that survive collections tend to have longer lifetimes, it makes sense to promote them to a higher generation. The garbage collector doesn't have to re-examine the objects in generations 1 and 2 each time it performs a collection of generation 0.

  If a collection of generation 0 does not reclaim enough memory for the application to create a new object, the garbage collector can perform a collection of generation 1, then generation 2. Objects in generation 1 that survive collections are promoted to generation 2.

- **Generation 2.**

  This generation contains long-lived objects. An example of a long-lived object is an object in a server application that contains static data that's live for the duration of the process.

  Objects in generation 2 that survive a collection remain in generation 2 until they are determined to be unreachable in a future collection.

  Objects on the large object heap (which is sometimes referred to as generation 3) are also collected in generation 2.

Garbage collections occur on specific generations as conditions warrant. Collecting a generation means collecting objects in that generation and all its younger generations. A generation 2 garbage collection is also known as a full garbage collection, because it reclaims objects in all generations (that is, all objects in the managed heap).

Survival and promotions

Objects that are not reclaimed in a garbage collection are known as survivors and are promoted to the next generation:

- Objects that survive a generation 0 garbage collection are promoted to generation 1.
- Objects that survive a generation 1 garbage collection are promoted to generation 2.
- Objects that survive a generation 2 garbage collection remain in generation 2.

When the garbage collector detects that the survival rate is high in a generation, it increases the threshold of allocations for that generation. The next collection gets a substantial size of reclaimed memory. The

CLR continually balances two priorities: not letting an application's working set get too large by delaying garbage collection and not letting the garbage collection run too frequently.

GC Class

It is a static class available in System namespace. It controls the system garbage collector, a service that automatically reclaims unused memory.

Example:

```
using System;
namespace GCCollectIntExample
{
    class MyGCCollectClass
    {
        private const long maxGarbage = 1000;
        static void Main()
        {
            MyGCCollectClass myGCCol = new MyGCCollectClass();
            // Determine the maximum number of generations the system
            // garbage collector currently supports.
            Console.WriteLine("The highest generation is {0}", GC.MaxGeneration);
            myGCCol.MakeSomeGarbage();
            // Determine which generation myGCCol object is stored in.
            Console.WriteLine("Generation: {0}", GC.GetGeneration(myGCCol));
            // Determine the best available approximation of the number
            // of bytes currently allocated in managed memory.
            Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(false));
            // Perform a collection of generation 0 only.
            GC.Collect(0);
            // Determine which generation myGCCol object is stored in.
            Console.WriteLine("Generation: {0}", GC.GetGeneration(myGCCol));
            Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(false));
            // Perform a collection of all generations up to and including 2.
            GC.Collect(2);
            // Determine which generation myGCCol object is stored in.
            Console.WriteLine("Generation: {0}", GC.GetGeneration(myGCCol));
            Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(false));
            Console.Read();
        }
        void MakeSomeGarbage()
        {
            Version vt;
            for(int i = 0; i < maxGarbage; i++)
            {
                // Create objects and release them to fill up memory
                // with unused objects.
                vt = new Version();
            }
        }
    }
}
```

The Garbage Collector and Unmanaged Resources

During a collection, the garbage collector will not free an object if it finds one or more references to the object in managed code. However, the garbage collector does not recognize references to an object from unmanaged code, and might free objects that are being used exclusively in unmanaged code unless explicitly prevented from doing so. The `KeepAlive()` method provides a mechanism that prevents the garbage collector from collecting objects that are still in use in unmanaged code.

Aside from managed memory allocations, implementations of the garbage collector do not maintain information about resources held by an object, such as file handles or database connections. When a type uses unmanaged resources that must be released before instances of the type are reclaimed, the type can implement a finalizer.

In most cases, finalizers are implemented by overriding the `Object.Finalize()` method; however, types written in C# or C++ implement destructors, which compilers turn into an override of `Object.Finalize()`. In most cases, if an object has a finalizer, the garbage collector calls it prior to freeing the object. However, the garbage collector is not required to call finalizers in all situations; for example, the `SuppressFinalize()` method explicitly prevents an object's finalizer from being called. Also, the garbage collector is not required to use a specific thread to finalize objects, or guarantee the order in which finalizers are called for objects that reference each other but are otherwise available for garbage collection.

In scenarios where resources must be released at a specific time, classes can implement the `IDisposable` interface, which contains the `IDisposable.Dispose()` method that performs resource management and cleanup tasks. Classes that implement Dispose must specify, as part of their class contract, if and when class consumers call the method to clean up the object. The garbage collector does not, by default, call the Dispose method; however, implementations of the Dispose method can call methods in the GC class to customize the finalization behavior of the garbage collector.

Object aging allows applications to target garbage collection at a specific set of generations rather than requiring the garbage collector to evaluate all generations. Overloads of the `Collect()` method that include a generation parameter allow you to specify the oldest generation to be garbage collected.

Disallowing garbage collection

Starting with the .NET Framework 4.6, the garbage collector supports a no GC region latency mode that can be used during the execution of critical paths in which garbage collection can adversely affect an app's performance. The no GC region latency mode requires that you specify an amount of memory that can be allocated without interference from the garbage collector. If the runtime can allocate that memory, the runtime will not perform a garbage collection while code in the critical path is executing.

You define the beginning of the critical path of the no GC region by calling one of the overloads of the `TryStartNoGCRegion()`. You specify the end of its critical path by calling the `EndNoGCRegion()` method.

You cannot nest calls to the `TryStartNoGCRegion()` method, and you should only call the `EndNoGCRegion()` method if the runtime is currently in no GC region latency mode. In other words, you should not call `TryStartNoGCRegion()` multiple times (after the first method call, subsequent calls will not succeed), and you should not expect calls to `EndNoGCRegion()` to succeed just because the first call to `TryStartNoGCRegion()` succeeded.