What's so wrong about using GC.Collect()?

Asked 16 years, 3 months ago Modified 4 years, 2 months ago

Viewed 75k times

114

Although I do understand the serious implications of playing with this function (or at least that's what I think), I fail to see why it's becoming one of these things that respectable programmers wouldn't ever use, even those who don't even know what it is for.

Let's say I'm developing an application where memory usage varies extremely depending on what the user is doing. The application life cycle can be divided into two main stages: editing and real-time processing. During the editing stage, suppose that billions or even trillions of objects are created; some of them small and some of them not, some may have finalizers and some may not, and suppose their lifetimes vary from a very few milliseconds to long hours. Next, the user decides to switch to the real-time stage. At this point, suppose that performance plays a fundamental role and the slightest alteration in the program's flow could bring catastrophic consequences. Object creation is then reduced to the minimum possible by using object pools and the such but then, the GC chimes in unexpectedly and throws it all away, and someone dies.

The question: In this case, wouldn't it be wise to call GC.Collect() before entering the second stage?

After all, these two stages never overlap in time with each other and all the optimization and statistics the GC could have gathered would be of little use here...

Note: As some of you have pointed out, .NET might not be the best platform for an application like this, but that's beyond the scope of this question. The intent is to clarify whether a GC.Collect() call can improve an application's overall behaviour/performance or not. We all agree that the circumstances under which you would do such a thing are extremely rare but then again, the GC tries to guess and does it perfectly well most of the time, but it's still about guessing.

Thanks.

.net performance memory-management garbage-collection

Share

Improve this question

Follow

edited Oct 20, 2008 at 0:36

community wiki
9 revs
Trap

26 "the slightest alteration in the program's flow could bring catastrophic consequences ... someone could die" - are you sure C# .NET is sufficiently deterministic for your purposes? – Steve Jessop Sep 23, 2008 at 1:35

4 Not Windows nor .NET are realtime platforms and therefore you cannot guarantee performance metrics, at least not enough to the point of risking human lives. I agree with onebyone that either you are exaggerating or careless. – Sergio Acosta Sep 23, 2008 at 1:45

3 LOL at "one of these things that respectable programmers wouldn't ever use, even those who don't even know what it is for"! Programmers who use stuff not knowing why are hardly the most respectable in my book. :) – The Dag Jan 23, 2013 at 16:32

20 Answers

Sorted by: Highest score (default)

92

From Rico's Blog...

> **Rule #1**
>
> Don't.
>
> This is really the most important rule. It's fair to say that most usages of GC.Collect() are a bad idea and I went into that in some detail in the orginal posting so I won't repeat all that here. So let's move on to...
>
> **Rule #2**

> Consider calling GC.Collect() if some non-recurring event has just happened and this event is highly likely to have caused a lot of old objects to die.
>
> A classic example of this is if you're writing a client application and you display a very large and complicated form that has a lot of data associated with it. Your user has just interacted with this form potentially creating some large objects... things like XML documents, or a large DataSet or two. When the form closes these objects are dead and so GC.Collect() will reclaim the memory associated with them...

So it sounds like this situation may fall under Rule #2, you know that there's a moment in time where a lot of old objects have died, and it's non-recurring. However, don't forget Rico's parting words.

> Rule #1 should trump Rule #2 without strong evidence.

Measure, measure, measure.

Share Improve this answer Follow

edited Sep 25, 2020 at 11:15

mio
101 ● 4 ● 9

answered Sep 23, 2008 at 2:15

Jon Norton
3,009 ● 22 ● 20

12 I'd say this is just the old thing. Nothing is really bad or dangerous if you know what you're doing and therefore know when and how to do it, as well as it side effects. Things like don't never, ever use xxxx are put there to protect the world from lousy programmers :D – Jorge Córdoba Jun 2, 2009 at 21:39

see also stackoverflow.com/questions/233596/…
– Ian Ringrose Sep 24, 2009 at 15:45

1 I'm not saying using GC.Collect is good practice. But sometimes it is a quick way to resolve issues without knowing its real cause. It's ugly, I know, but it does work, and it seems to me not a bad approach, especially when there's not much time to figure out the root cause of the issue and your boss is standing behind you... you know.
– Silent Sojourner Jan 3, 2018 at 20:22

61

If you call GC.Collect() in production code you are essentially declaring that you know more then the authors of the GC. That may be the case. However it's usually not, and therefore strongly discouraged.

Share Improve this answer Follow

edited May 11, 2012 at 19:34
GregC
7,957 ● 2 ● 59 ● 70

answered Sep 23, 2008 at 1:49
Aaron Fischer
21.2k ● 18 ● 78 ● 117

5 That's very true, but I don't know if they could make assumptions that apply for all developments. – MasterMastic Sep 9, 2012 at 5:18

2 @Ken No, they can't. But are you in a better position to do so? Or are you going to write code assuming specific hardware, a specific OS version and so on? The pain/gain ratio is too high on this one. – The Dag Jan 23, 2013 at 16:43

2 @TheDag IMO of course I am. When I'm releasing memory and whatnot I don't really care about the hardware because that's the OS job to deal with that. I also don't care about the OS because I have an interface common to all of the ones I'm programming for. (e.g. I don't care if it's Windows, Mac, or Linux: when I'm allocating/freeing memory in C/C++ it's new/delete malloc/dealloc). I could always be wrong so feel free to correct me. – MasterMastic Jan 24, 2013 at 1:28

@MasterMastic `malloc` only has a very simple interface, and it's implementations can vary enough to matter. It all depends on what kind of a problem you're trying to solve. If `malloc` was "good enough", you wouldn't need buffer pooling, would you? C/C++ development is *full* of examples where you try to second-guess the OS/runtime/libraries because you know better (and sometimes, you really do). Many performance-critical applications avoid using system/runtime allocators entirely. Games used to pre-allocate all memory at startup (constant-sized arrays etc.). – Luaan Feb 13, 2017 at 9:41

27

So how about when you are using COM objects like MS Word or MS Excel from .NET? Without calling `GC.Collect` after releasing the COM objects we have found that the Word or Excel application instances still exist.

In fact the code we use is:

```
Utils.ReleaseCOMObject(objExcel)

' Call the Garbage Collector twice. The GC needs to be
get the
' Finalizers called - the first time in, it simply mak
be finalized,
' the second time in, it actually does the finalizing.
object do its
' automatic ReleaseComObject. Note: Calling the GC is
' but one that may be necessary when automating Excel
way to
' release all the Excel COM objects referenced indirec
' Ref: http://www.informit.com/articles/article.aspx?p
' Ref: http://support.microsoft.com/default.aspx?scid=
GC.Collect()
GC.WaitForPendingFinalizers()
GC.Collect()
GC.WaitForPendingFinalizers()
```

So would that be an incorrect use of the garbage collector? If so how do we get the Interop objects to die? Also if it isn't meant to be used like this, why is the `GC`'s `Collect` method even `Public`?

Share Improve this answer Follow

edited May 20, 2019 at 8:09

community wiki
4 revs, 4 users 86%
Dib

4 This would make a great new StackOverflow question, i.e: How to eradicate COM instances without calling the GC. With particular regard to unmanaged circular references. It's one

of the challenges that made me wary of upgrading my VB6 Outlook add-in to C#. (We did a lot of work to develop coding patterns and test cases on the VB side that guaranteed COM references were killed in a deterministic fashion when no longer needed). – rkagerer Nov 7, 2012 at 23:25

2 If this applies to COM objects in general, perhaps this is a valid scenario. But off the bat I'd say the problem is likely that you are using a client application designed for an interactive desktop as a COM server. From MSDN knowledge base: "Microsoft does not currently recommend, and does not support, Automation of Microsoft Office applications from any unattended, non-interactive client application or component (including ASP, ASP.NET, DCOM, and NT Services), because Office may exhibit unstable behavior and/or deadlock when Office is run in this environment." – The Dag Jan 23, 2013 at 17:08

3 @TheDag - Microsoft may not recommend, but many of us have had to port old VB6 code with with office interop to .Net windows apps. I spent months of work until I finally got rid rid of all invisible hanging references for a big VB6 to .Net conversion project. Learning to release in reverse assignment order and holding local refs to EVERY single com object including the collections helped though. – Dib Oct 5, 2014 at 22:17

17

Well, the GC is one of those things I have a love / hate relationship with. We have [broken it in the past](#) through VistaDB and blogged about it. They have fixed it, but it takes a LONG time to get fixes from them on things like this.

The GC is complex, and a one size fits all approach is very, very hard to pull off on something this large. MS has done a fairly good job of it, but it is possible to fool the GC at times.

In general you should not add a `Collect` unless you know for a fact you just dumped a ton of memory and it will go to a mid life crisis if the GC doesn't get it cleaned up now.

You can screw up the entire machine with a series of bad `GC.Collect` statements. The need for a collect statement almost always points to a larger underlying error. The memory leak usually has to do with references and a lack of understanding to how they work. Or using of the `IDisposable` on objects that don't need it and putting a much higher load on the GC.

Watch closely the % of time spent in GC through the system performance counters. If you see your app using 20% or more of its time in the GC you have serious object management issues (or an abnormal usage pattern). You want to always minimize the time the GC spends because it will speed up your entire app.

It is also important to note that the GC is different on servers than workstations. I have seen a number of small difficult to track down problems with people not testing both of them (or not even aware that their are two of them).

And just to be as full in my answer as possible you should also test under Mono if you are targeting that platform as well. Since it is a totally different implementation it may experience totally different problems that the MS implementation.

Share Improve this answer Follow

edited Jan 18, 2017 at 22:06
jrh
408 ● 2 ● 10 ● 30

answered Sep 23, 2008 at 3:09
Jason Short
5,314 ● 1 ● 31 ● 44

The culprit is often events. Whenever an instance method is used as an event handler, the publisher of the event has a reference to the subscriber via the event delegate. The only "easy" way to avoid problems from this is to only use publishers that are at most as long-lived as the subscribers (e.g. a TextBox publishing an event handled by the containing form is no problem, as the textbox isn't supposed to live outside of the form). Example problem scenario: Singleton model, temporary views handling model events. – The Dag Jan 23, 2013 at 16:53

8 How can one screw up the entire machine? – Adam R. Grey May 19, 2016 at 15:11

16

There are situations where it's useful, but in general it should be avoided. You could compare it to GOTO, or riding a moped: you do it when you need to, but you don't tell your friends about it.

Share Improve this answer Follow

answered Sep 23, 2008 at 1:39
rjohnston
7,343 ● 8 ● 31 ● 37

12

From my experience it has never been advisable to make a call to GC.Collect() in production code. In debugging, yes, it has it's advantages to help clarify potential memory leaks. I guess my fundamental reason is that the GC has been written and optimized by programmers much smarter then I, and if I get to a point that I feel I need to call GC.Collect() it is a clue that I have gone off path somewhere. In your situation it doesn't sound like you actually have memory issues, just that you are concerned what instability the collection will bring to your process. Seeing that it will not clean out objects still in use, and that it adapts very quickly to both rising and lowering demands, I would think you will not have to worry about it.

Share Improve this answer Follow

answered Sep 23, 2008 at 1:40

TheZenker
1,730 ● 1 ● 16 ● 29

11

One of the biggest reasons to call GC.Collect() is when you have just performed a significant event which creates lots of garbage, such as what you describe. Calling GC.Collect() can be a good idea here; otherwise, the GC may not understand that it was a 'one time' event.

Of course, you should profile it, and see for yourself.

Share Improve this answer

answered Sep 23, 2008 at 2:07

Follow

TraumaPony
10.8k ● 13 ● 57 ● 75

9

Well, obviously you should not write code with real-time requirements in languages with non-real-time garbage collection.

In a case with well-defined stages, there is no problem with triggering the garbage-collector. But this case is extremely rare. The problem is that many developers are going to try to use this to paper-over problems in a cargo-cult style, and adding it indiscriminately will cause performance problems.

Share Improve this answer

Follow

answered Sep 23, 2008 at 1:36

wnoise
9,902 ● 39 ● 47

True. But automated tests capable of catching the error condition "object not eligible for garbage collection, but should be" would be valuable. I this it might be achieved via a combination of factory logic, destructor logic, and GC.Collect. E.g. your Entity class has a IObjectTracker property, ordinarily null but assigned by the test-purpose entity factory. The factory also notifies the tracker of object birth, while the destructor notifies it (when present) of death. If you can know "destructor has executed for all garbage-collectible objects" you could check tracker state to detect leaks. – The Dag Jan 23, 2013 at 17:04

7

Calling GC.Collect() forces the CLR to do a stack walk to see if each object can be truely be released by checking references. This will affect scalability if the number of objects is high, and has also been known to trigger garbage collection too often. Trust the CLR and let the garbage collector run itself when appropriate.

Share Improve this answer Follow

answered Sep 23, 2008 at 1:55

Ta01
31.6k ● 13 ● 73 ● 100

2 Not only do you cause the stack walk, but your applications main thread (and any child threads it created) are frozen so the GC **can** walk the stack. The more time your app spends in GC, the more time it spends frozen. – Scott Dorman Sep 23, 2008 at 3:17

6 I am more concerned about an App Crash due to an Out of Memory exception than slow performance because the app / GC was throwing out things that are no longer needed. Does anybody know why Microsoft appears to throw an OOM exception without FIRST throwing out the garbage? (Without this OBVIOUS step - or at least an explanation of why this step does not appear to be attempted before throwing OOM exception I'm not sure I have any faith in things happening "automatically" the "way they are supposed to." – Wonderbird Jun 10, 2015 at 17:29

6

Nothing is wrong with explicitly calling for a collection. Some people just really want to believe that if it is a service provided by the vendor, don't question it. Oh, and all of those random freezes at the wrong moments of your interactive application? The next version will make it better!

Letting a background process deal with memory manipulation means not having to deal with it ourselves, true. But this does not logically mean that it is best for us to not deal with it ourselves under all circumstances. The GC is optimized for most cases. But this does not logically mean that it is optimized in all cases.

Have you ever answered an open question such as 'which is the best sorting algorithm' with a definitive answer? If so, don't touch the GC. For those of you who asked for the conditions, or gave 'in this case' type answers, you may proceed to learn about the GC and when to activate it.

Gotta say, I've had application freezes in Chrome and Firefox that frustrate the hell out of me, and even then for some cases the memory grows unhindered -- If only they'd learn to call the garbage collector -- or given me a button so that as I begin to read the text of a page I can hit it and thus be free of freezes for the next 20 minutes.

Share Improve this answer Follow

answered Apr 3, 2017 at 22:18

community wiki
Gerard ONeill

6

Infact, I don't think it is a very bad practice to call GC.Collect.
There may be cases when we need that. Just for instance, I have a form which runs a thread, which inturn opens differnt tables in a database, extracts the contents in a BLOB field to a temp file, encrypt the file, then read the file into a binarystream and back into a BLOB field in another table.

The whole operation takes quite a lot of memory, and it is not certain about the number of rows and size of file content in the tables.

I used to get OutofMemory Exception often and I thought it would be wise to periodically run GC.Collect based on a counter variable. I increment a counter and when a specified level is reached, GC is called to collect any garbage that may have formed, and to reclaim any memory lost due to unforeseen memory leaks.

After this, I think it is working well, atleast no exception!!!
I call in the following way:

```
var obj = /* object utilizing the memory, in my case F
GC.Collect(GC.GetGeneration(obj ,GCCollectionMode.Opti
```

Share Improve this answer Follow

edited Feb 5, 2019 at 11:35

community wiki

3 revs, 3 users 48%
Venugopal M

5

Under .net, the time required to perform a garbage collection is much more strongly related to the amount of stuff that isn't garbage, than to the amount of stuff that is. Indeed, unless an object overrides `Finalize` (either explicitly, or via C# destructor), is the target of a `WeakReference`, sits on the Large Object Heap, or is special in some other gc-related way, the only thing identifying the memory in which it sits as being an object is the existence of rooted references to it. Otherwise, the GC's operation is analogous to taking from a building everything of value, and dynamiting the building, building a new one on the site of the old one, and putting all the valuable items in it. The effort required to dynamite the building is totally independent of the amount of garbage within it.

Consequently, calling `GC.Collect` is apt to increase the overall amount of work the system has to do. It will delay the occurrence of the next collection, but will probably do just as much work immediately as the next collection would have required when it occurred; at the point when the next collection would have occurred, the total amount of time spent collecting will have been about the same as had `GC.Collect` not been called, but the system will have accumulated some garbage, causing the succeeding collection to be required sooner than had `GC.Collect` not been called.

The times I can see `GC.Collect` really being useful are when one needs to either measure the memory usage of some code (since memory usage figures are only really meaningful following a collection), or profile which of several algorithms is better (calling GC.Collect() before running each of several pieces of code can help ensure a consistent baseline state). There are a few other cases where one might know things the GC doesn't, but unless one is writing a single-threaded program, there's no way one can know that a `GC.Collect` call which would help one thread's data structures avoid "mid-life crisis" wouldn't cause other threads' data to have a "mid-life crises" which would otherwise have been avoided.

Share Improve this answer Follow

answered May 13, 2012 at 16:31

community wiki
supercat

5

Creating images in a loop - even if you call dispose, the memory is not recovered. Garbage collect every time. I went from 1.7GB memory on my photo processing app to 24MB and performance is excellent.

There are absolutely time that you need to call GC.Collect.

Share Improve this answer Follow

answered Oct 22, 2012 at 20:52

dale
77 ● 1 ● 1

3 Calling `Dispose` is *not supposed to* release managed memory. You don't seem to know how the memory model in .NET works. – Andrew Barber Oct 22, 2012 at 20:55

5

We had a similar issue with the garbage collector not collecting garbage and freeing up memory.

In our program, we were processing some modest sized Excel Spreadsheets with OpenXML. The spreadsheets contained anywhere from 5 to 10 "sheets" with about 1000 rows of 14 columns.

The program in a 32 bit environment (x86) would crash with an "out of memory" error. We did get it to run in an x64 environment, but we wanted a better solution.

We found one.

Here are some simplified code fragments of what didn't work and what did work when it comes to explicitly calling the Garbage Collector to free up memory from disposed objects.

Calling the GC from inside the subroutine didn't work. Memory was never reclaimed...

```
For Each Sheet in Spreadsheets
    ProcessSheet(FileName,sheet)
Next
```

```
Private Sub ProcessSheet(ByVal Filename as string, ByV
    ' open the spreadsheet
    Using SLDoc as SLDocument = New SLDocument(Filenam
        ' do some work....
        SLDoc.Save
    End Using
    GC.Collect()
    GC.WaitForPendingFinalizers()
    GC.Collect()
    GC.WaitForPendingFinalizers()
End Sub
```

By Moving the GC call to outside the scope of the subroutine, the garbage was collected and the memory was freed up.

```
For Each Sheet in Spreadsheets
    ProcessSheet(FileName,sheet)
    GC.Collect()
    GC.WaitForPendingFinalizers()
    GC.Collect()
    GC.WaitForPendingFinalizers()
Next

Private Sub ProcessSheet(ByVal Filename as string, ByV
    ' open the spreadsheet
    Using SLDoc as SLDocument = New SLDocument(Filenam
        ' do some work....
        SLDoc.Save
    End Using
End Sub
```

I hope this helps others that are frustrated with the .NET garbage collection when it appears to ignore the calls to `GC.Collect()`.

Paul Smith

Share Improve this answer Follow

edited Jan 18, 2017 at 15:35

community wiki
2 revs, 2 users 90%
Paul Smith

4

The worst it will do is make your program freeze for a bit. So if that's OK with you, do it. Usually it's not needed for thick client or web apps with mostly user interaction.

I have found that sometimes programs with long-running threads, or batch programs, will get OutOfMemory exception even though they are disposing objects properly. One I recall was a line-of-business database transaction processing; the other was an indexing routine on a background thread in a thick client app.

In both cases, the result was simple: No GC.Collect, out of memory, consistently; GC.Collect, flawless performance.

I've tried it to solve memory problems several other times, to no avail. I took it out.

In short, don't put it in unless you're getting errors. If you put it in and it doesn't fix the memory problem, take it back out. Remember to test in Release mode and compare apples to apples.

The only time things can go wrong with this is when you get moralistic about it. It's not a values issue; many programmers have died and gone straight to heaven with many unneccessary GC.Collects in their code, which outlives them.

Share Improve this answer Follow

answered Aug 17, 2012 at 1:19

community wiki
FastAl

2

I think you are right about the scenario, but I'm not sure about the API.

Microsoft says that in such cases you should add memory pressure as a hint to the GC that it should soon perform a collection.

Share Improve this answer Follow

answered Sep 23, 2008 at 1:33

Sergio Acosta
11.4k ● 12 ● 63 ● 91

2 Interesting, but the documentation says that AddMemoryPressure should be used when 'a small managed object allocates a large amount of UNmanaged memory'. (emphasis mine) – Robert Paulson Oct 20, 2008 at 1:04

What's wrong with it? The fact that you're second-guessing the garbage collector and memory allocator,

2

which between them have a much greater idea about your application's actual memory usage at runtime than you do.

Share Improve this answer Follow

answered Sep 24, 2008 at 23:14

Rob
48.4k ● 5 ● 75 ● 93

2 The heuristic nature of the garbage collector and the fact that they exposed this functionality to the outside world make me think of it as something that is useful if used where it needs to be. The problem is not using it but knowing how, where and when to use it. – Trap Sep 25, 2008 at 10:29

Not to mention the GCs better knowledge about every *other* application and their memory needs. The GC negotiates memory with the OS and as such is impacted by available physical memory and all other processes on the machine both managed and unmanaged. While I doubt the GC really knows "when is a good time to collect" on a "case-by-case" basis, it's very likely to have a better strategy overall than... ANY single application. ;) – The Dag Jan 23, 2013 at 17:12

2

The desire to call GC.Collect() usually is trying to cover up for mistakes you made somewhere else!

It would be better if you find where you forgot to dispose stuff you didn't need anymore.

Share Improve this answer Follow

answered Oct 19, 2008 at 17:23

Sam
29k ● 50 ● 170 ● 250

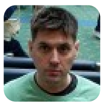
8 thats perhaps a generalisation – user585968 Jan 30, 2012 at 5:35

1

Bottom line, you can profile the application and see how these additional collections affect things. I'd suggest staying away from it though unless you are going to profile. The GC is designed to take care of itself and as the runtime evolves, they may increase efficiency. You don't want a bunch of code hanging around that may muck up the works and not be able to take advantage of these improvements. There is a similar argument for using foreach instead of for, that being, that future improvements under the covers can be added to foreach and your code doesn't have to change to take advantage.

Share Improve this answer Follow

answered Sep 23, 2008 at 1:41



Christopher Elliott
87 ● 2 ● 7

1

The .NET Framework itself was never designed to run in a realtime environment. If you truly need realtime processing you would either use an embedded realtime language that isn't based on .NET or use the .NET Compact Framework running on a Windows CE device.

Share Improve this answer Follow

answered Sep 23, 2008 at 3:25

Scott Dorman
42.5k ● 12 ● 81 ● 112

He could be using the .Net Micro Framework, which WAS designed for real-time environments. – TraumaPony Sep 23, 2008 at 5:45

@TraumaPony: Check the chart at the bottom of this page msdn.microsoft.com/en-us/embedded/bb278106.aspx: Clearly the Micro Framework was not designed for real-time environments. It was, however, designed for embedded environments (like WinCE) but with lower power requirements. – Scott Dorman Sep 23, 2008 at 9:39