What's the point of the garbage collector

Asked 16 years, 2 months ago Modified 16 years, 2 months ago Viewed 724 times

3

```
SqlConnection connection = new SqlConnection(FROM_CONFIGURATION)
SqlCommand command = new SqlCommand("SomeSQL", connection);
connection.Open();
command.ExecuteNonQuery();
command.Dispose();
connection.Dispose();
```

It is recommended that the code above should include try/catch (or using) so that if an exception is thrown, all resources are disposed properly.

But if you have to worry about disposing stuff manually, then what's the point of the GC?! Isn't GC there to take care of this for the coder?

.net garbage-collection

Share
Improve this question
Follow

edited Oct 21, 2008 at 19:41

Ray
46.5k ● 29 ● 127 ● 170

asked Oct 21, 2008 at 19:38

anon

1 Good question for those who are interested in optimizing their code. – Kon Oct 21, 2008 at 19:47

3 How does this relate to optimizing code? – Scott Dorman Oct 21, 2008 at 19:54

15 Answers

Sorted by: Highest score (default)

15

As other people here said the GC is non-deterministic, so you don't know when your object will be collected. What I want to clarify is that this is not a problem with the memory, but with the system resources (opened files, database connections) which are expensive and should be released asap. Dispose lets you do that when you know you're no longer using the connection. If they are not released in time the system might run out of those resources and the GC isn't aware of that. That's why you have to do it manually.

Also I want to add that using the 'using' statement will do it for you in a nicely way.

Share Improve this answer Follow

answered Oct 21, 2008 at 19:46

rslite
84.5k ● 4 ● 47 ● 47

7

Garbage collection is here to take care of releasing unused **memory**.

There are various reasons why extending GC to release other resources is problematic. One of them is that finalizers (methods that are executed when an object is being garbage collected) make reference cycles uncollectable, unless you strongly constrain dereferencing from finalizers, which makes them very tricky to use.

Another reason is that most resources need to be deallocated in some sort of timely manner, which is not possible when relying on garbage collection.

Yet another reason is that limiting GC to memory management deals with the *vast bulk* of resource management in any application, and almost all of the *non interesting* resource management. Other resources are usually interesting enough do deserve some additional code to be explicit about how they are released.

And another reason is that in some applications, GC makes the application go faster because it reduces the amount of copying done to cater for ownership semantics. Which is not a concern for other resources.

Other people could go on like this for hours.

Share
Improve this answer
Follow

edited Oct 21, 2008 at 20:04

answered Oct 21, 2008 at 19:48

ddaa
54.3k ● 8 ● 52 ● 59

6

> But if you have to worry about disposing stuff manually, then what's the point of the GC?! Isn't GC there to take care of this for the coder?

The problem is you have no idea **when** the GC will run. And if your application never pressures memory, it may not run at all.

Share Improve this answer Follow

answered Oct 21, 2008 at 19:40

Michael Burr
340k ● 52 ● 548 ● 769

2 ... and some resources are far more scarce than memory. – Aaron Maenpaa Oct 21, 2008 at 19:41

5

Suppose I have this code:

```
class MonkeyGrabber : IDisposable {
   public MonkeyGrabber() { /* construction grabs a real, live monkey from the
cage */
   public void Dispose() { Dispose(true); /* releases the monkey back into the
cage */ }
   // the rest of the monkey grabbing is left as an exercise to grad student
drones
}

class MonkeyMonitor {
    public void CheckMonkeys() {
        if (_monkeyPool.GettingTooRowdy()) {
            MonkeyGrabber grabber = new MonkeyGrabber();
            grabber.Spank();
        }
    }
}
```

Now, my MonkeyMonitor checks the monkeys and if they're too rowdy, it gets a valuable system resource - a single monkey grabbing claw attached to the system, and uses it to grab a monkey and spank it. Since I didn't dispose it, the monkey claw is still holding onto the monkey dangling it above the rest cage. If the rest of the monkeys continue to get rowdy, I can't make a new MonkeyGrabber as it is still help up. Oops. A contrived example, but you get the point: Objects that implement IDisposable may hold onto limited resources that should be released in a timely manner. The GC may let go eventually or not.

In addition, some resources need to be released in a timely manner. I have a set of classes that if they are not disposed by either app or GC before the application exits will cause the app to crash hard, as the unmanaged resource manager from which they came is already gone by the time the GC gets around to it.

More on IDisposable.

using is your friend - it's the closest we have so far to RAII.

Share

Improve this answer

Follow

edited May 23, 2017 at 12:09

Community Bot

1 ● 1

answered Oct 21, 2008 at 19:55

plinth

49.1k ● 11 ● 83 ● 123

4

Objects that implement IDisposable are trying to tell you that they have linkages to structures that aren't managed memory. The garbage collector runs in batches in order to improve efficiency. But that means it may be awhile before your object is disposed of, meaning you'll be holding onto resources longer than you should be which may have negative effects on performance/reliability/scalability.

Share Improve this answer Follow

answered Oct 21, 2008 at 19:44

Orion Adrian

19.5k ● 13 ● 54 ● 67

3

The GC runs occasionally and takes care of memory management keeping everything nice and tidy for you. You may think it is useless when you see fragments of code like the one you posted, but more often than not, it saves you a lot of headaches (think C/C++ manual memory management) since it greatly reduces memory leaks and lets you worry about how your application will run and not how you will manage memory. Disposing of file handles and databases connections is a way of increasing efficiency since garbage collection is not deterministic and may not happen right away, and you don't want those file handles and open database connections sapping your systems performance. Btw, your code really is ugly, I always advocate the using statement and frequently write my db code like this:

```
using (SqlConnection connection = new SqlConnection(...))
using (SqlCommand command = connection.CreateCommand())
{
   ...
}
```

This automatically calls dispose on the connection and command objects when they fall out of scope which is when execution leaves the block.

Share Improve this answer Follow

answered Oct 21, 2008 at 19:47

IAmCodeMonkey

1,568 ● 1 ● 11 ● 11

1 c++ manual memory management? Dude, that's like so 80s, have you never heard of RAII? – gbjbaanb Oct 21, 2008 at 20:52

2

Because GC isn't the most efficient - it doesn't always occur as soon as the resource is no longer used. So when you're dealing with unmanaged resources like file I/O, DB connections, etc. it's best practice to release/clean up those resources before waiting and relying on GC to take care of it.

And look into using the `using` keyword:

```
using (SqlConnection connection = new SqlConnection(FROM_CONFIGURATION))
using (SqlCommand command = new SqlCommand("SomeSQL", connection))
{
  connection.Open();
```

```
    command.ExecuteNonQuery();
    command.Dispose();
    connection.Dispose();
}
```

Also, as a general rule, whatever can be disposed, should be in a `using` block.

Share

Improve this answer

Follow

edited Oct 21, 2008 at 19:50

answered Oct 21, 2008 at 19:43

Kon

27.4k ● 12 ● 63 ● 87

1 If you are using the using statement, you do not need to explictly call the Dipose() method, that is the entire point of using the 'using'. – IAmCodeMonkey Oct 21, 2008 at 19:51

I wouldn't necessarily say that the GC isn't efficient since it doesn't occur as soon as a resource is no longer used. Part of a garbage collected runtime is that it is non-deterministic and only runs when it determines there is a need to run. – Scott Dorman Oct 21, 2008 at 19:52

@Scott: Exactly. Being able to delay garbage collection actually makes it *more* efficient. – Jon Skeet Oct 21, 2008 at 19:59

@Scott: You're right, I just didn't get my thoughts out in a well-formulated manner. @IAmCodeMonkey: I never said to explicitly call Dispose(). I said to use 'using' on whatever is disposable. – Kon Oct 21, 2008 at 20:05

The GC is less efficient overall, you simply defer deallocations. However, you then also need a new thread to collect in, and the main app needs to be locked (you can't muck about with objects while the GC is running!) but putting finalisers on objects make it truly inefficient. – gbjbaanb Oct 21, 2008 at 20:55

2

> ...The GC was never meant to manage resources; it was designed to manage memory allocation ... In the specific case of database connections, you are dealing with other resources than just memory... (Scott Dorman)

The OP didn't tag with a specific platform, though most of the answers have been .net-specific, noting that GC is mainly for avoiding memory leaks, but extensions such as `using` expressions and `IDisposable` can help a lot.

Other platforms offer other solutions. For example, in C++, there is no (built-in) garbage collection, but some forms of shared pointers can be used to help with memory management, and the RAII-style of coding can be extremely helpful in managing other types of resources.

In cPython, two different garbage collection systems are used. A referencing counting implementation immediately calls destructors when the last reference is deleted. For common "stack" objects, this means that they get cleaned up immediately, like what

happens for C++ RAII objects. The downside is that if you have a reference cycle, the reference counting collector will never dispose of the object. As a result, they have a secondary non-deterministic garbage collector that works like Java and .NET collectors. Like .NET with its using statements, cPython tries to handle the most common cases.

So, to answer the OP, non-deterministic garbage collection helps simplify memory management, it can be used to handle other resources too as long as timeliness isn't an issue, and another mechanism (such as careful programming, reference counting GC, a using statement, or real RAII objects) are needed when timely releasing of other resources is needed.

Share Improve this answer Follow

answered Oct 21, 2008 at 20:01

Mr Fooz

112k ● 7 ● 76 ● 103

1

the above-mentioned code releases acquired resources (although, i don't believe you should be calling the Dispose() method by yourself, by releasing the resources is meant closing streams and stuff like that). GC removes object from memory (deallocates memory used by the object), but it can be done only after resources have been freed by the object.

Share Improve this answer Follow

answered Oct 21, 2008 at 19:40

zappan

3,685 ● 4 ● 32 ● 24

1 You should definitely call Dispose() on objects you are done with (or use using, which does the same thing). – Chris Marasti-Georg Oct 21, 2008 at 19:52

1

I'm not so sure about c#, which is what this looks like, but typically, the garbage collector manages memory. This connection, in addition to the object memory, has server resources. The database, which is in a separate process, has to maintain a connection. The close cleans those up.

Share Improve this answer Follow

answered Oct 21, 2008 at 19:41

sblundy

61.4k ● 22 ● 123 ● 124

The garbage collector (GC) in .NET is a core part of the .NET Common Language Runtime (CLR) and is available to all .NET programming languages. The GC was never meant to manage resources; it was designed to manage memory allocation,

1

and it does an excellent job at managing memory allocated directly to native .NET objects. It was not designed to deal with unmanaged memory and operating system allocated memory, so it becomes the responsibility of the developer to manage these resources.

In the specific case of database connections, you are dealing with other resources than just memory - specifically connection pools, possible implicit transaction scopes, etc. By calling Close() and/or Dispose() you are explicitly telling the object to release those unmanaged resources immediately while the managed resources will wait for a GC cycle to occur.

Share Improve this answer Follow

answered Oct 21, 2008 at 19:47

Scott Dorman

42.5k • 12 • 81 • 112

1

Ah I see now. I confused memory management with unmanaged resource management. Thanks for the clarification!

Share

Improve this answer

Follow

edited Oct 22, 2008 at 3:55

answered Oct 22, 2008 at 3:35

anon

0

GC does take care of disposing objects, but the disposal may not happen right away. Manually disposing the objects will free up memory faster.

Share Improve this answer Follow

answered Oct 21, 2008 at 19:41

JFV

1,783 • 6 • 24 • 38

0

The GC is limited when it comes to freeing external resources such as DB connections or file handles. However, for allocating memory within the .NET world, it takes care of a lot of the mundane memory management tasks.

Share Improve this answer Follow

answered Oct 21, 2008 at 19:42

Larsenal

51.1k • 43 • 154 • 223

The above provides an example of when natively allocated memory or resources wander into the managed world as a handle. In this scenario because the managed world did not allocate the memory it cannot "auto-tidy" it. The memory/resources has to be explicitly disposed or at the very least disposed within a finaliser.

In the vast majority of cases though, especially when talking about the code critical to most companies core aims (yeh business logicks) you don't have to worry about this kind of thing and less code means less mistakes.

Share Improve this answer Follow

answered Oct 21, 2008 at 19:49

Quibblesome

25.4k ● 10 ● 62 ● 104