

A re-announce on GC's defects

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2007-01/msg02637.html>

- *From:* "Born" <wandertop@xxxxxxxxxx>
 - *Date:* Thu, 18 Jan 2007 14:07:24 +0800
-

GC is really garbage itself

Reason 1:

There is delay between the wanted destruction and the actual destruction.

Negative effects by the destruction delay:

1) Efficiency issue

It's bad for CPU/Resource intensive but memory cheap objects.

CPU intensive objects refer to objects who own internal threads.

Resource intensive objects refer to objects who own unmanaged resources like file handle, network connections, etc.

Don't tell me these objects are rare. Everything is possible to happen and a general purpose language should not find any excuse not to apply to some situations.

2) Logic issue

A re-announce on GC's defects

The need for weak reference makes the destruction delay logically incorrect.

Weak references (or you can call them handles) refer to references who do not require the referenced targets to keep alive, while strong references do.

When all strong references to a target go out of their lifetime, the target also comes to the end of its lifetime. Right at this point, weak references to this target become invalid. However, the destruction delay caused by the GC violates this logic. Weak references will continue to think the target alive until the GC really collects the target object.

Don't tell me the `IDisposable` pattern is for that. There may be more than one strong references and you don't know when and where to call `Dispose`.

Don't tell me `WeakReference` (C#) is for that. If you don't have `Dispose` called properly, `WeakReference` still gives a wrong logic.

Don't tell me this can be solved by adding method like `IsDestroyed` to the target. It holds a candle to the sun and it only adds to the complexity of logics.

An example:

Suppose we're doing a 3D game. A radar is monitoring a target. Obviously, the radar should hold a weak reference to the target. When the target is killed, logical confusion is immediately brought to the radar watcher (the gamer). Is the target destroyed or not? You can not tell him, hey, it's killed but still shown on the radar because you've got to wait for the GC to make it.

Reason 2:

A re-announce on GC's defects

Poor scalability

This is a Theory issue.

GC is global, which means it scales as the application's memory use scales. Theoretically, this indicates a bad scalability.

Don't tell me an application should not use too many concurrent objects. Again, everything is possible. Restrictions only prove the defects of the language.

Fairly speaking, GC itself is not garbage. However, when java and C# integrate it and prevent the user from manually managing memory, it becomes garbage. Note GC in java and C# is not really an additive as someone would argue since there is no way to do real memory management like delete obj in C++.

Better memory management in my mind is reference counting + smart pointer, which makes things automatic and correct. You have deterministic destructions while no need to manually call Dispose. You need not to manually change the reference count as smart pointers help you achieve it. The only problem with this approach is cyclic reference. However, even if not theoretically proven, the problem generally can be solved by replacing some strong references with weak references.

I believe the restriction by GC is one of the main reasons why in some field (the gaming industry, for example), java or C# is rarely used in serious products who face real computing challenges.

Solution

A re-announce on GC's defects

A re-announce on GC's defects

1) The ideal solution is to convince the language providers to give us back the ability of managing memory by our own. GC can still be there, and it becomes a real addictive in that situation.

2) Transfer the burden to the user. We can ask the user to always take special cautions (for example, always use "using" in C# to have Dispose correctly called even exception occurs). Things can work around if the user do them right. However, that's at risk in nature and not a robust solution.