How do I get the reference count of a CLR object?

devblogs.microsoft.com/oldnewthing/20100811-00

August 11, 2010

Raymond Chen

A customer asked the rather enigmatic question (with no context):

> Is there a way to get the reference count of an object in .Net?
>
> Thanks,
> Bob Smith
> Senior Developer
> Contoso

The CLR does not maintain reference counts, so there is no reference count to "get". The garbage collector only cares about whether an object has zero references or at least one reference. It doesn't care if there is one, two, twelve, or five hundred—from the point of view of the garbage collector, one is as good as five hundred.

The customer replied,

> I am aware of that, yet the mechanism is somehow implemented by the GC…
>
> What I want to know is whether at a certain point there is more then one variable pointing to the same object.

As already noted, the GC does not implement the "count the number of references to this object" algorithm. It only implements the "Is it definitely safe to reclaim the memory for his object?" algorithm. A null garbage collector always answers "No." A tracing collector looks for references, but it only cares *whether* it found one, not how many it found.

The discussion of "variables pointing to the same objects" is somewhat confused, because you can have references to an object from things other than variables. Parameters to a method contain references, the implicit `this` is also a reference, and partially-evaluated expressions also contain references. (During execution of the line `string s = o.ToString();`, at the point immediately after `o.ToString()` returns and before the result is assigned to `s`, the string has an active reference but it isn't stored in any variable.) And as we saw earlier, merely storing a reference in a variable doesn't prevent the object from being collected.

It's clear that this person solved half of his problem, and just needs help with the other half, the half that doesn't make any sense. (I like how he immediately weakened his request from "I want the exact reference count" to "I want to know if it is greater than one." Because as we all know, the best way to solve a problem is to reduce it to an even harder problem.)

Another person used some psychic powers to figure out what the real problem is:

> If I am reading properly into what you mean, you may want to check out the `WeakReference` class. This lets you determine whether an object has been collected. Note that you don't get access to a reference count; it's a zero/nonzero thing. If the `WeakReference` is empty, it means the object has been collected. You don't get a chance to act upon it (as you would if you were the last one holding a reference to it).

The customer explained that he tried `WeakReference`, but it didn't work. (By withholding this information, the customer made the mistake of not saying what he already tried and why it didn't work.)

> Well this is exactly the problem: I instantiate an object and then create a `WeakReference` to it (global variable).
>
> Then at some point the object is released (set to null, disposed, erased from the face of the earth, you name it) yet if I check the `IsAlive` property it still returns true.
>
> Only if I explicitly call to `GC.Collect(0)` or greater before the check it is disposed.

The customer still hasn't let go of the concept of reference counting, since he says that the object is "released". In a garbage-collected system, object are not released; rather, you simply stop referencing them. And disposing of an object still maintains a reference; disposing just invokes the `IDisposable.Dispose` method.

```
FileStream fs = new FileStream(fileName);
using (fs) {
 ...
}
```

At the end of this code fragment, the `FileStream` has been disposed, but there is still a reference to it in the `fs` variable. Mind you, that reference isn't very useful, since there isn't much you can do with a disposed object, Even if you rewrite the fragment as

```
using (FileStream fs = new FileStream(fileName)) {
 ...
}
```

the variable `fs` still exists after the close-brace; it simply has gone out of scope (i.e., you can't access it any more). Scope is not the same as lifetime. Of course, the optimizer can step in and make the object eligible for collection once the value becomes inaccessible, but there is

no requirement that this optimization be done.

The fact that the `IsAlive` property says `true` even after all known references have been destroyed is also no surprise. The environment does not check whether an object's last reference has been made inaccessible every time a reference changes. One of the major performance benefits of garbage collected systems comes from the de-amortization of object lifetime determination. Instead of maintaining lifetime information about an object continuously (spending a penny each time a reference is created or destroyed), it saves up those pennies and splurges on a few dollars every so often. The calculated risk (which usually pays off) is that the rate of penny-saving makes up for the occasional splurges.

It does mean that between the splurges, the garbage collector does not know whether an object has outstanding references or not. It doesn't find out until it does a collection.

The null garbage collector takes this approach to an extreme by simply hoarding pennies and never spending them. It saves a lot of money but consumes a lot of memory. The other extreme (common in unmanaged environments) is to spend the pennies as soon as possible. It spends a lot of money but reduces memory usage to the absolute minimum. The designers of a garbage collector work to find the right balance between these two extremes, saving money overall while still keeping memory usage at a reasonable level.

The customer appears to have misinterpreted what the `IsAlive` property means. The property doesn't say whether there are any references to the object. It says whether the object has been garbage collected. Since the garbage collector can run at any time, there is nothing meaningful you can conclude if `IsAlive` returns `true`, since it can transition from alive to dead while you're talking about it. On the other hand, once it's dead, it stays dead; it is valid to take action when `IsAlive` is `false`. (Note that there are two types of `WeakReference`; the difference is when they issue the death certificate.)

The name `IsAlive` for the property could be viewed as misleading if you just look at the property name without reading the accompanying documentation. Perhaps a more accurate (but much clumsier) name would have been `HasNotBeenCollected`. The theory is, presumably, that if you're using an advanced class like `WeakReference`, which works "at the GC level", you need to understand the GC.

The behavior the customer is seeing is correct. The odds that the garbage collector has run between annihilating the last live reference and checking the `IsAlive` property is pretty low, so when you ask whether the object has been collected, the answer will be No. Of course, forcing a collection will cause the garbage collector to run, and that's what does the collection and sets `IsAlive` to `false`. Mind you, forcing the collection to take place messes up the careful penny-pinching the garbage collector has been performing. You forced it to pay for a collection before it had finished saving up for it, putting the garbage collector in debt. (Is

there a garbage collector debt collector?) And the effect of a garbage collector going into debt is that your program runs slower than it would have if you had let the collector spend its money on its own terms.

Note also that forcing a generation-zero collection does not guarantee that the object in question will be collected: It may have been promoted into a higher generation. (Generational garbage collection takes advantage of typical real-world object lifetime profiles by spending only fifty cents on a partial collection rather than a whole dollar on a full collection. As a rough guide, the cost of a collection is proportional to the number of live object scanned, so the most efficient collections are those which find mostly dead objects.) Forcing an early generation-zero collection messes up the careful balance between cheap-but-partial collections and expensive-and-thorough collections, causing objects to get promoted into higher generations before they really deserve it.

Okay, that was a long discussion of a short email thread. Maybe tomorrow I'll do a better job of keeping things short.

Bonus chatter: In addition to the `WeakReference` class, there is also the `GCHandle` structure.

Bonus reading: Maoni's WebLog goes into lots of detail on the internals of the CLR garbage collector. Doug Stewart created this handy index.