gc-cycles

June 7, 2021

1 Garbage Collection with Isolated Strong-Reference Cycles

One automatic memory management feature is Garbage Collection (GC), a stripped-down form of which is compile-time Automatic Reference Counting (ARC). A major practical difference to the programmer is handling of isolated strong-reference cycles (ISRC's).

As suggested by its name, an ISRC can arise for example when an object is a self-loop (e.g. a singleton pointer array that simply contains its own memory address), when two objects contain references to each other (e.g. a parent object that points to its child, with that child object also pointing back up to its parent), or when three or more objects form a "ouroboros" (e.g. a linked list where the tail node points back to the head node). The "isolated" part arises when such a cycle has no incoming pointers from the program's usable scope.

1.1 The Trouble with ISRC's

ISRC's can be problematic because they violate the intuition that an object whose reference count (refcount, the number of incoming pointers to its memory address) is greater than 0 cannot be garbage-collected.

Usually, an object whose refcount is positive is in usable scope, and therefore accessible (in a non-nasal-demons-invoking way) by the program. For example, it could be assigned as a property of another object that is itself referenced by a stack variable. Such an object cannot be safely garbage-collected.

But objects in an ISRC can have positive refcounts without being in usable scope. For example, consider `x = []; x.append(x); del x`. The list itself still contains a pointer to its own memory address, so its refcount is +1. But the list is no longer in usable scope, because the only way we could have accessed it is through the `x` variable name, which has gone out of scope. If we naively follow the refcount intuition from above, the list will keep itself alive for no reason. This is a kind of memory leak.

1.2 Steps Between Source Code and Execution

Before we explore this further, it's worth clarifying some of the steps from source code to run-time.

1.2.1 Source Code

Source code is code written in a human-readable language, usually a high-level language like Python or C. [1]

1.2.2 Object Code

Source code gets compiled into object code, which is either bytecode designed to run on a virtual machine (VM, for example the Python interpreter contains a VM and the Java runtime is itself a VM), or machine code designed to be invoked by the operating system's shell or kernel and run natively on the physical machine (for example, Swift code and C code compile directly to machine code). [2], [3]

1.2.3 Run-time

Object code itself becomes useful when we "execute" or "run" it. The word "run-time" (or "run-time") can mean "the time while a program is running" (referring either to the length of time that elapses, or to the entire period of elapsed time itself), or it can be used as shorthand for "runtime environment" meaning "all the machinery required to actually execute the instructions that a piece of object code specifies". Thus, we can say "the Java runtime performs GC at run-time", meaning "the Java VM performs GC while the program is running". [4]

1.3 Full-Featured GC

Full-featured GC (for example "tracing" GC as implemented by VM's for languages like Python[5] or Java[6]) is not vulnerable to ISRC's. A simple form of tracing GC continually runs graph search on the graph of known references (starting with root references, for example local variables on the stack or static variables), traversing the entire graph from reference to reference [7].

At the end of the search, any object that was not reached is by definition not in usable scope (since the only way for an object to be in usable scope is for its memory address to be a root reference, or for its memory address to be pointed to by some object which is itself reachable from a root reference), and its memory can be freed.

The tradeoff is that the GC thread stays alive in the background at run-time, consuming memory (including both memory used by the GC thread itself, and the "extra" memory each of the target

[1] Sometimes people refer to assembly code as "source code", and indeed assembly is a human-readable language, but it's quite low-level.

[2] For example, Oracle's reference implementation of the Java compiler is `javac` on most platforms, and Apple's reference implementation of the Swift compiler is `swiftc` on macOS.

[3] An assembly-code compiler is called an "assembler", and assembly-code compilation is called "assembly". "Disassembly" is the reverse translation, from machine code to human-readable assembly code.

[4] For example, the Python Software Foundation's reference implementation of the Python interpreter, CPython, is `python` on most platforms, and Oracle's reference implementation of the Java VM is `java` on most platforms. Apple also provides a reference implementation of the Swift interactive REPL UI, `swift` on macOS. That REPL can be considered a "Swift runtime environment". But for most applications, there will be no dedicated "Swift runtime environment": As we said, Swift is generally compiled directly to machine code which is then distributed with the intention that it will be run natively on the end user's physical machine.

[5] An interpreted language whose compiled object code is Python bytecode. For more on CPython's GC, see https://devguide.python.org/garbage_collector/.

[6] A compiled language whose compiled object code is Java bytecode.

[7] See https://softwareengineering.stackexchange.com/questions/377197/how-a-garbage-collector-follows-pointers-to-discover-live-objects.

process's individual object allocations needs to reserve to store that object's GC metadata) and CPU time. And in practice, in order to avoid race conditions, many GC's employ a "stop the world" tactic, pausing or "blocking" the target process mid-execution every so often in order to run their operations without fear of interference.

1.3.1 Generational GC

Both Python and Java improve on this tradeoff a bit by implementing "generational" GC, which avoids the overhead of a full trace by maintaining a list of object "generations". New objects are born into generation 0, and if they survive a round of GC, they get promoted to generation 1. When the GC needs to free up more space, it will start with generation 0, proceeding to sweep generation 1 only if the generation-0 sweep does not free up enough space. CPython's GC, for example, keeps track of three distinct generations.

The idea is that most objects are "temporary", and go out of scope very quickly, hence most objects can be safely garbage-collected soon after they are created. Think of a line like `heads = RNG(seed=42).random() > 0.80`, which models the flip of a biased coin: the `RNG` instance is seeded and used to generate a standard Uniform pseudorandom variable, then immediately goes out of scope. Its refcount was never even positive. Focusing GC on young objects, therefore, is likely to yield a high "hit rate" of successful garbage collections, freeing up enough memory to proceed without having to sweep too many objects.

On the other hand, an object that survives a round of GC is probably "permanent". Think of an object allocated within a function, which object's reference is then assigned to a global `pointer_t` variable which is used throughout the rest of the program. Promoting such an object to the next generation keeps track of the fact that attempting to garbage-collect it in the future is probably a waste of time, to be avoided unless absolutely necessary.

Actual Efficiency Gains Much as the actual efficiency gains of clever caching schemes depend critically on a program's actual data read/write patterns, the actual efficiency gains of a generational GC scheme depends critically on a program's actual allocation/deallocation patterns.

For example, generational GC on a program that allocates many new objects, allocates even more new objects, allows the original batch to go out of scope, allocates further new objects, allows the second batch to go out of scope, and so on, could end up *slower* than non-generational GC, since there is operational overhead associated with keeping track of the generations. In this program, that overhead could be wasted, because it is always the youngest objects that remain in scope and the older objects that are allowed to go out of scope, so the GC could end up always needlessly sweeping the young un-collectable objects before moving on to the old collectable objects. In this case, a more efficient GC scheme might have been FIFO, whereby the oldest generations are swept first.

1.4 ARC

On the other hand, ARC as implemented by e.g. the Swift compiler is a relatively space- and time-efficient form of GC, because it is a compile-time operation that simply injects explicit memory management instructions into the compiled machine code [8]. These machine-code instructions are

[8] Perhaps the only thing more time-efficient would be to decline to manage memory altogether. The ensuing memory leaks would of course be inelegant and totally unscalable. But there are some small-scale applications where this is an effective tradeoff.

essentially the same as the instructions that a C compiler would have injected, but much more convenient to the programmer because they are automatic.

For example, unlike C programmers, in most cases Swift programmers don't need to remember to manually write a `free()` call in their source code when they're done with some object: The Swift compiler will "see" that the object's refcount has fallen to zero, hence its memory is no longer reachable [9], and inject the machine instructions corresponding to `free()` in the appropriate place, automatically.

The tradeoff is that ARC needs the programmer to "nudge" it in some cases.

1.4.1 ISRC's

For example, Swift programmers defining a closure that captures `self`, then assigning that closure as a property of `self` itself[10], are familiar with the need to specify that the reference is either weakly captured or unowned [11]: If they don't do this, even when the `self` instance permanently leaves usable scope, the instance (which refers to the closure as a property) and the closure (which has "closed over" or "captured" the instance) will form an ISRC keeping each other alive for no reason. Unless the operating system is configured to detect and handle this at run-time, this will be a memory leak.

1.5 Toy Example

Below, I show a toy example of GC at work detecting and handling an ISRC in Python. [12]

[1]:
```
import gc
```

[2]:
```
class Foo:

    def __init__(self, name, other=None):
        """Construct a `Foo` instance whose name is `name` and partner is␣
↪`other`."""
        self.name = name
        self.other = other

    def __repr__(self):
        """Identify this `Foo` instance in a human-readable way."""
```

[9] At least, not in any well-defined way. A programmer could of course insist on invoking undefined behavior by attempting to access the memory at that address notwithstanding. These are the so-called "nasal demons".

[10] This is the proverbial "elf on a `self`". Just kidding.

[11] You'll see them write e.g. `[weak self]`, which is a capture list.

[12] Prior to Python 3.4 which implemented PEP 442, the below toy example would not have worked, because we define a custom `__del__()` destructor for the `Foo` type to help announce what's going on under the hood. Although it could detect them, the CPython GC would not garbage-collect an ISRC of objects with custom `__del__()` destructors, because it could not decide in what order it should destroy the objects. Python programmers had to use `weakref` for at least one reference in a cycle of objects with custom destructors. This is the same as the Swift ARC's solution: By replacing one of the *strong* references with a *weak* reference, you turn an isolated strong-reference *cycle* into an isolated strong-reference *chain*, at which juncture it's clear that you should start at the beginning of the chain—the object with no incoming references—and work your way out. PEP 442 got around this issue by simply "copping out" and making the order of destruction undefined, which means that the interpreter makes no promises and can choose whichever arbitrary order is most convenient.

```
        return f"{self.name}Foo"

    def __del__(self):
        """Announce that this `Foo` instance is being destroyed e.g. by the GC.
 ↪"""
        print(f"{repr(self)} is being destroyed!")
```

If we construct an object, assign it to a named variable, then immediately `del` that variable, the GC will destroy the object. Note that `del` command itself does NOT destroy the object referenced by a variable name: It simply takes the variable name out of scope and decrements the referenced object's refcount by one.

[3]:
```
eve_var = Foo(name="Eve")

# now EveFoo's refcount is +1

del eve_var  # take `eve_var` out of scope and decrement EveFoo's refcount by -1

# now EveFoo's refcount has reached 0, so she can be safely garbage-collected
```

```
EveFoo is being destroyed!
```

This works even if we don't `del` the variable, but simply reassign it:

[4]:
```
reve_var = Foo(name="ReincarnatedEve")

# now ReincarnatedEveFoo's refcount is +1

reve_var = None  # reassign `reve_var` to point to `None` and decrement␣
 ↪ReincarnatedEveFoo's refcount by -1

# now ReincarnatedEveFoo's refcount has reached 0, so she can be safely␣
 ↪garbage-collected
```

```
ReincarnatedEveFoo is being destroyed!
```

This will *not* work if we store the object somewhere else first, then `del` the variable..

[5]:
```
rreve_var = Foo(name="ReincarnatedReincarnatedEve")

# now ReincarnatedReincarnatedEveFoo's refcount is +1

anonymous_list = [rreve_var,]

# now ReincarnatedReincarnatedEveFoo's refcount is +2

del rreve_var  # take `rreve_var` out of scope and decrement␣
 ↪ReincarnatedReincarnatedEveFoo's refcount by -1
```

```
# now ReincarnatedReincarnatedEveFoo's refcount is still +1, so she cannot be␣
↪safely garbage-collected
```

.. but it will work if we then also take that "somewhere else" out of scope (and, to avoid having to wait around for the sake of this demo, manually prompt the GC to run a full sweep)!

[6]:
```
del anonymous_list  # decrement ReincarnatedReincarnatedEveFoo's refcount by -1

# now ReincarnatedReincarnatedEveFoo's refcount has reached 0, so she can be␣
↪safely garbage-collected

_ = gc.collect()  # manually prompt the GC to run a full sweep, suppressing its␣
↪output
```

```
ReincarnatedReincarnatedEveFoo is being destroyed!
```

1.5.1 ISRC

Objects that are unreachable will be garbage-collected, even if they are involved in an ISRC:

[7]:
```
alice_var = Foo(name="Alice")
bob_var = Foo(name="Bob")

# now AliceFoo's and BobFoo's refcounts are each +1

alice_var.other = bob_var
bob_var.other = alice_var

# now AliceFoo's and BobFoo's refcounts are each +2

del alice_var, bob_var

# now AliceFoo's and BobFoo's refcounts are each still +1, so even though they␣
↪are unreachable and can therefore be safely garbage-collected, they will not␣
↪be garbage-collected.. or will they??!

_ = gc.collect()  # manually prompt the GC to run a full sweep, suppressing its␣
↪output
```

```
AliceFoo is being destroyed!
BobFoo is being destroyed!
```