An insight into Python garbage collection

Fabio Falzoi

$ whoami

- full stack developer @ **SpazioDati s.r.l.**
- happy Python user since 2014
- ~3 (scattered) years of professional Python experience

Outline

- CPython reference counting
- CPython generational garbage collection
- GC in real life
- PyPy and Go gc approaches

CPython reference counting

Reference Counting

For each object, use a counter to keep the number of references to the object

When the reference count is increased?

- assignment operator
- argument passing
- appending an object to a list
- …

When the reference count is decreased?

- reference goes out of scope (function return)
- …

sys.getrefcount

```python
foo = []

# 2 references, 1 from the foo var and 1 from getrefcount
sys.getrefcount(foo)

def bar(a):
    # 4 references
    # from the foo var, function argument,
    # getrefcount and Python's function stack
    sys.getrefcount(a)

bar(foo)
# 2 references, the function scope is destroyed
sys.getrefcount(foo)
```

Quick Quiz #1: sys.getrefcount oddities

If you run the following snippet with CPython 3.7.3 in a REPL

```python
import sys

a = 1

sys.getrefcount(a)
```

you'll get `118`

Can you figure out why this happens?

Quick Quiz #1: sys.getrefcount oddities

If you run the following snippet with CPython 3.7.3 in a REPL

```
import sys

a = 1

sys.getrefcount(a)
```

you'll get `118`

Can you figure out why this happens?

answer

CPython **interns** the number from -5 to 256 (including) to save memory and optimize performance. Same thing goes for strings with just one character.

```
#define NSMALLPOSINTS           257
#define NSMALLNEGINTS           5

static PyLongObject small_ints[NSMALLNEGINTS + NSMALLPOSINTS];
```

refcount - PyObject

In CPython, each object stores the number of references to it in the `ob_refcnt` field

```
typedef struct _object {
    _PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;
    struct _typeobject *ob_type;
} PyObject;
```

refcount - Py_INCREF and Py_DECREF

How `ob_refcnt` is incremented and decremented in CPython?

```
static inline void _Py_INCREF(PyObject *op)
{
    _Py_INC_REFTOTAL;
    op->ob_refcnt++;
}

#define Py_INCREF(op) _Py_INCREF(_PyObject_CAST(op))
```

```
static inline void _Py_DECREF(const char *filename, int lineno, PyObject *op)
{
    ...

    if (--op->ob_refcnt != 0) {
#ifdef Py_REF_DEBUG
        ...
#endif
    }
    else {
        _Py_Dealloc(op);
    }
}

#define Py_DECREF(op) _Py_DECREF(__FILE__, __LINE__, _PyObject_CAST(op))
```

The good, the bad and the ugly about reference counting

Good

- easy to implement
- when refcount hits 0, objs are immediately deleted

Bad

- space overhead
- execution overhead
  - extra check whenever an object gets allocated, referenced, or dereferenced
- unbounded latency
  - *cascading effect* for large and convoluted heaps
- memory fragmentation

Ugly

- not thread safe
- doesn't detect cycles

The ugly 1/2: not thread safe

What if two threads concurrently decrease an object reference counter?

Suppose `refcnt == 2`

Thread 1:

```
CPU register <- refcnt
CPU register -= 1
refcnt <- CPU register
```

Thread 2:

```
CPU register <- refcnt
CPU register -= 1
refcnt <- CPU register
```

These operations are not atomic, so there's a race condition…

refcnt may end up with a value of 0 or 1 !!!

If refcnt turns out to be 1, we may experience a **memory leak** :-(

Quick Quiz #2: Concurrently increasing refcnt

What happens if two threads try to concurrently increase the same refcnt?

Quick Quiz #2: Concurrently increasing refcnt

What happens if two threads try to concurrently increase the same refcnt?

answer

Even worse than before!

We have again a race condition: the ref counter may be increased just by 1. Since it will permanently be lower than its real value, the object may be collected when it is still actively referenced.

Therefore, we may end up **referencing memory already freed** and all sorts of (bad) things can happen.

Solution: the Python's *infamous* GIL

The ugly 2/2: cyclical references

Reference cycles can only occur in container objects (lists, dicts, classes, tuples)

Here an example:

```
obj_1, obj_2 = {}, {}
obj_1['next'], obj_2['next'] = obj_2, obj_1
```

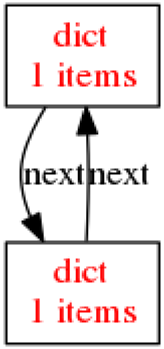
The ugly 2/2: cyclical references

Reference cycles can only occur in container objects (lists, dicts, classes, tuples)

Here an example:

```
obj_1, obj_2 = {}, {}
obj_1['next'], obj_2['next'] = obj_2, obj_1
```

`objgraph` module confirms the we are dealing with a cyclic reference:



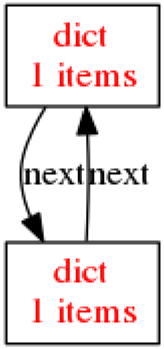
The ugly 2/2: cyclical references

Reference cycles can only occur in container objects (lists, dicts, classes, tuples)

Here an example:

```
obj_1, obj_2 = {}, {}
obj_1['next'], obj_2['next'] = obj_2, obj_1
```

`objgraph` module confirms the we are dealing with a cyclic reference:



How to solve this?

CPython generational garbage collection

Introducing generational garbage collection

CPython adds to the reference counting, another garbage collector to deal with cyclic references

That additional GC is a form of *tracing garbage collection*. As such, it consists of threee phases:

1. Scan
2. Mark
3. Sweep

Introducing generational garbage collection

CPython adds to the reference counting, another garbage collector to deal with cyclic references

That additional GC is a form of *tracing garbage collection*. As such, it consists of threee phases:

1. Scan
2. Mark
3. Sweep

In a nutshell: the gc **scan** the entire heap to **mark** reachable objects, then **sweep** away the unreachable ones

Introducing generational garbage collection

CPython adds to the reference counting, another garbage collector to deal with cyclic references

That additional GC is a form of *tracing garbage collection*. As such, it consists of threee phases:

1. Scan
2. Mark
3. Sweep

In a nutshell: the gc **scan** the entire heap to **mark** reachable objects, then **sweep** away the unreachable ones

These are costly operations, but CPython manges to improve the overall performance using a *heuristic* approach

Introducing generational garbage collection

CPython adds to the reference counting, another garbage collector to deal with cyclic references

That additional GC is a form of *tracing garbage collection*. As such, it consists of threee phases:

1. Scan
2. Mark
3. Sweep

In a nutshell: the gc **scan** the entire heap to **mark** reachable objects, then **sweep** away the unreachable ones

These are costly operations, but CPython manges to improve the overall performance using a *heuristic* approach

The **generational hypothesis** says that *the most recently created objects are also those most likely to become unreachable quickly*

Short-lived objects allocation

We define the **age** of an object as the number of times it survived a garbage collection cycle

Short-lived objects allocation

We define the **age** of an object as the number of times it survived a garbage collection cycle

Does the generational hypotesis holds true in Python?

```
# allocates temporary floats
avg = (a + b + c)/3

# allocates a temporary iterator for sequence
[x for x in sequence]

# allocates many temporary strings
python.replace('CPython', 'PyPy').replace('PyPy', 'Jython')
  .replace('Jython', 'IronPython').replace('IronPython', 'PythonNet')

# and so on
```

Short-lived objects allocation

We define the **age** of an object as the number of times it survived a garbage collection cycle

Does the generational hypotesis holds true in Python?

```
# allocates temporary floats
avg = (a + b + c)/3

# allocates a temporary iterator for sequence
[x for x in sequence]

# allocates many temporary strings
python.replace('CPython', 'PyPy').replace('PyPy', 'Jython')
  .replace('Jython', 'IronPython').replace('IronPython', 'PythonNet')

# and so on
```

Take home message

We can separate objects by their age, enqueing them in separate lists (**generations**). Then, it is convenient to collect younger generations more frequently than older ones.

CPython generational gc

CPython keeps three lists of every (**container**) object allocated as a program is run:

- *generation 0*
- *generation 1*
- *generation 2*

Younger objects are stored in generation 0 and they are *promoted* to the older generation if they survive a garbage collection cycle.

CPython generational gc internals

GC container objects

Container objects that should be tracked by generational gc are identified by a flag

```
/* Objects support garbage collection (see objimp.h) */
#define Py_TPFLAGS_HAVE_GC (1UL << 14)
```

a C macro is defined to help test for the presence of the flag

```
/*
 * Garbage Collection Support
 * ==========================
 */

...

/* Test if a type has a GC head */
#define PyType_IS_GC(t) PyType_HasFeature((t), Py_TPFLAGS_HAVE_GC)
```

An example of a GC container object

As an example, consider the `defaultdict` type from `collections` module

from `Modules/_collectionsmodule.c`:

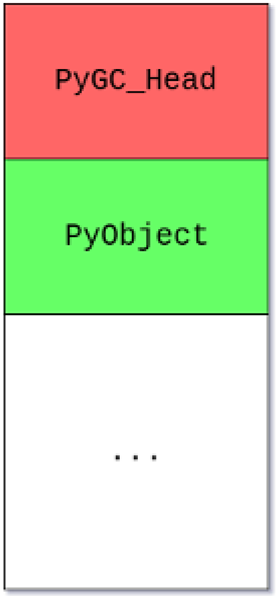
```
static PyTypeObject defdict_type = {
    PyVarObject_HEAD_INIT(DEFERRED_ADDRESS(&PyType_Type), 0)
    "collections.defaultdict",          /* tp_name */

    ...

    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
                                /* tp_flags */

    ...

};
```

GC internal structs: PyGC_Head

GC information is stored on top of every `PyObject`

Specifically, every gc tracked object is linked in a doubly-linked list through a struct `PyGC_Head`

```
typedef struct {
    uintptr_t _gc_next;
    uintptr_t _gc_prev;
} PyGC_Head;
```



_gc_prev field

When not collecting, `_gc_prev` is to link container objects in a doubly linked list

Lowest two bits of `_gc_prev` are used for flags. The most important one is `PREV_MASK_COLLECTING`, which is set when the object is undergoing a collection and cleared once the cycle detection algorithm ends.

During a collection, `_gc_prev` is temporary used for storing `gc_refs`, that is the current value of `ob_refcnt`.

Quick quiz #3:

Why CPython uses the same field `_gc_prev` both as a pointer and as a reference counter?

Quick quiz #3:

Why CPython uses the same field `_gc_prev` both as a pointer and as a reference counter?

answer

To reduce memory overhead: every single container object has a structure `PyGCHead` on top of it!

GC internal structs: gc_generation

Descriptor of a GC generation

```
struct gc_generation {
    PyGC_Head head;
    int threshold;
    int count;
};
```

- `head`
  all `gc_generation` structs are doubly-linked

- `threshold`
  garbage collection threshold

- `count`
  for generation 0: difference between allocations and deallocations
  for older generations: count of collections completed

GC internal structures: gc_runtime_state

Struct describing the current runtime status of the generational garbage collection

- `long_lived_total`
  number of objects that survived the last *full collection*

- `long_lived_pending`
  number of objects that survived all *non-full collection*

N.B. A *full collection* is a collection on all generations

To further limit the overhead, the generational gc starts a full collections only if
`long_lived_pending` > `long_lived_total` / 4

```
struct _gc_runtime_state {
    ...
    struct gc_generation generations[NUM_GENERATIONS];
    PyGC_Head *generation0;
    struct gc_generation permanent_generation;
    struct gc_generation_stats generation_stats[NUM_GENERATIONS];
    ...
    Py_ssize_t long_lived_total;
    Py_ssize_t long_lived_pending;
};
```

More on that `permanent_generation` later: hold your horses! :-)

Generational GC lifecycle

GC lifecycle

When an object with `Py_TPFLAGS_HAVE_GC` flag is allocated, two things happen:

1. The allocation may start a gc collection cycle
2. The gc starts to track the newly allocated object

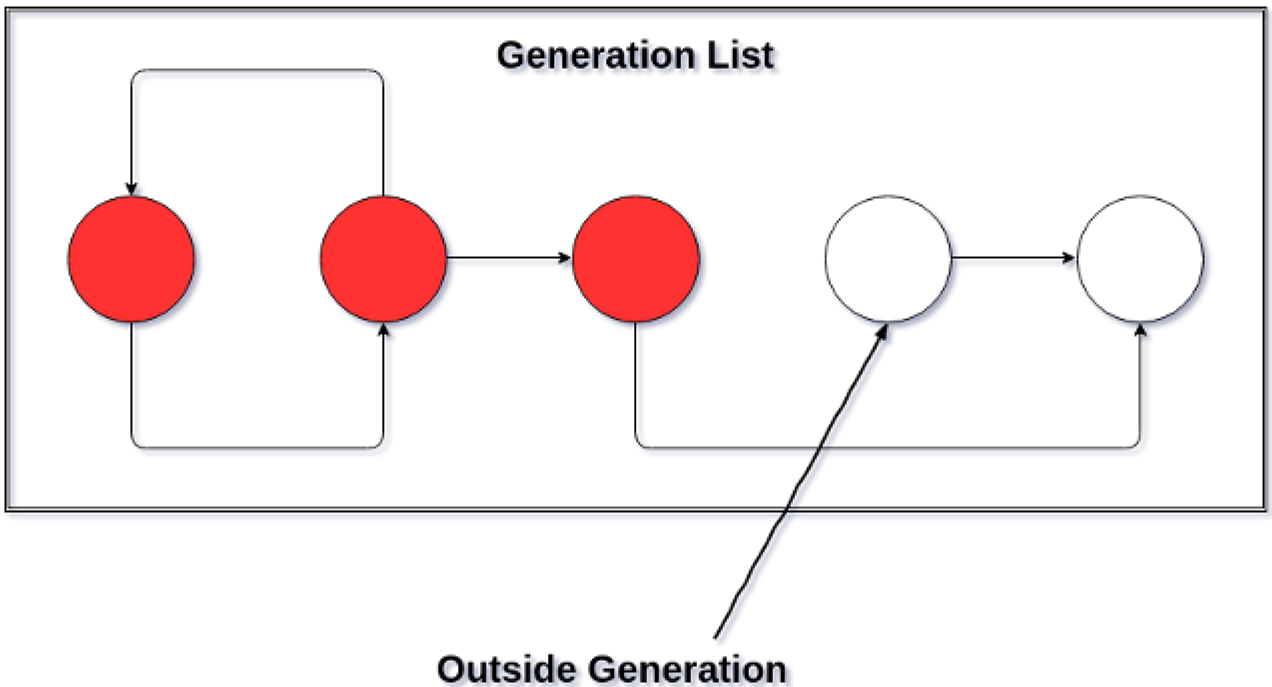
```
PyObject *
PyType_GenericAlloc(PyTypeObject *type, Py_ssize_t nitems)
{
    PyObject *obj;

    ...

    if (PyType_IS_GC(type))
        obj = _PyObject_GC_Malloc(size);

    ...

    if (PyType_IS_GC(type))
        _PyObject_GC_TRACK(obj);
    return obj;
}
```

Inner details of the collection process

Cycle detection algorithm: the 30k foot view

To break reference cycles on the i-th generation, the algorithm acts this way

- iterate over all objects in the collection under scanning
- for each object, traverse all its reference
- for each referenced object that is in the list, decrease its `gc_refs` by 1



collect_generations

- find the oldest generation where the count exceeds the threshold
- in case of use *full collections* apply the heuristic on `long_lived_pending` and `long_lived_total` values

```
static Py_ssize_t
collect_generations(void)
{
    int i;
    Py_ssize_t n = 0;

    for (i = NUM_GENERATIONS-1; i >= 0; i--) {
        if (_PyRuntime.gc.generations[i].count >
            _PyRuntime.gc.generations[i].threshold) {

            if (i == NUM_GENERATIONS - 1
                && _PyRuntime.gc.long_lived_pending <
                _PyRuntime.gc.long_lived_total / 4)
                continue;

            n = collect_with_callback(i);
            break;
        }
    }
    return n;
}
```

gc module workhorse: the collect function

- merge target generation with all younger ones to form the `young` list
- executes the cycle detection algorithm
  - leaving reachable object in `young` list
  - moving **possibly unreachable** objects into the `unreachable` list
- merge `young` list into next generation
- executes all finalizers of the `unreachable` objects
- executes the appropriate `clear` function on the `unreachable` objects

gc module workhorse: the collect function

```
static Py_ssize_t
collect(int generation, Py_ssize_t *n_collected, Py_ssize_t *n_uncollectable,
        int nofail)
{
    ...
    /* merge younger generations with one we are currently collecting */
    ...
    update_refs(young);  // gc_prev is used for gc_refs
    subtract_refs(young);
    ...
    move_unreachable(young, &unreachable);  // gc_prev is pointer again

    /* Move reachable objects to next generation. */
    ...
    gc_list_merge(young, old);

    if (check_garbage(&unreachable)) { // clear PREV_MASK_COLLECTING here
        gc_list_merge(&unreachable, old);
    }
    else {
        delete_garbage(&unreachable, old);
    }
    ...
}
```

visit_decref: the subtract_refs visitor function

Decrement `gc_refs` using the **visitor pattern**

```
static int visit_decref(PyObject *op, void *data)
{
    if (PyObject_IS_GC(op)) {
        PyGC_Head *gc = AS_GC(op);
        if (gc_is_collecting(gc)) {
            gc_decref(gc);
        }
    }
    return 0;
}
```

When the traversal is complete for all objs, in `young` list we'll end up with two kind of objects:

- objs with reference count > 0
  these objs are surely reachable from outside the `young` list, so they will be promoted to the older generation
- objs with reference count == 0 these objects **may** be unreachable from outside, so they are now eligible to be garbage collected

Quick Quiz #4: still reachable objs

During the visit of each object we decrease `gc_refs`, not `ob_refcnt` to avoid triggering an immediate memory reclaim.

Can you figure out why some objects with reference count == 0 may still end up as reachable?

Quick Quiz #4: still reachable objs

During the visit of each object we decrease `gc_refs`, not `ob_refcnt` to avoid triggering an immediate memory reclaim.

Can you figure out why some objects with reference count == 0 may still end up as reachable?

answer

The object can be referenced by another object that belong to the same generation, but with a reference count > 0. Since this object is still reachable, the first one is reachable as well!

Take-home message

We'll know which objs can be collected only after a full scan of the `young` list

Cycle detection algorithm: check_garbage

Before reclaiming memory, `check_garbage` walks again the collectable objs list and check that they are really unreachable

Why this, again? Because some objs could have been *resurrected* by a finalizer!

Thanks to PEP 442, since CPython 3.4, the generational gc can safely support finalizers.

End of the journey: delete_garbage

- break reference cycles by calling the appropriate clear function of the container.
- when `ob_refcnt` falls to 0, the object memory is immediately released

```
static void delete_garbage(PyGC_Head *collectable, PyGC_Head *old)
{
    ...

    while (!gc_list_is_empty(collectable)) {
        PyGC_Head *gc = GC_NEXT(collectable);
        PyObject *op = FROM_GC(gc);

        ...

        if ((clear = Py_TYPE(op)->tp_clear) != NULL) {
            Py_INCREF(op);
            (void) clear(op);

            ...

            Py_DECREF(op);
        }

        ...

    }
}
```

Feel confused?

If you feel confused by now… well, you're not alone!

Just read the comment on top of `delete_garbage` :-)

```
/*
 * ... It is possible I screwed something up here.
 */
```

Garbage collection is hard, indeed.

Generational gc final performance notes

CPython generational garbage collection is a **stop-the-world** collector: during the entire collection process the program is not making progress

Despite this, the overall performance are acceptable:

- Reference counting plays nicely with generational garbage collector
- The generations make the GC **incremental**
- CPython limits full collections with a further heuristic

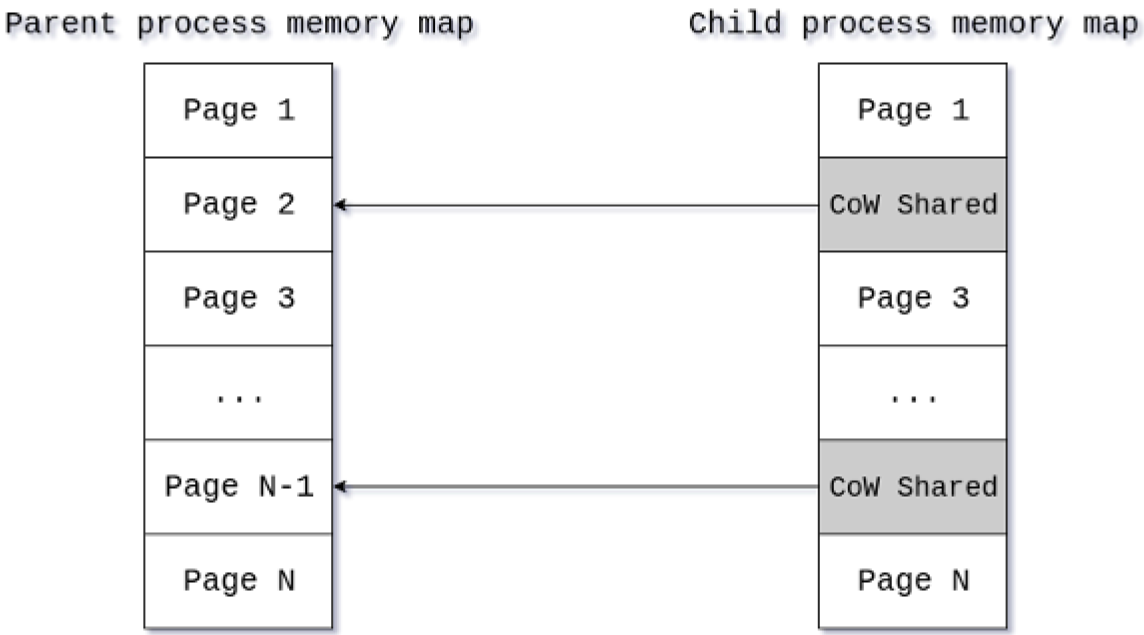
Garbage Collection in real life: an Instagram story

Linux Copy-on-Write

It is a kernel optimization to avoid unnecessary memory copy

- you have a process that owns a memory page
- that process forks a child process, this one will share the same memory page, marked as read-only
- when the child process tries to write to that page, it gets a page fault
- the kernel then duplicates the page before restarting the write

As a result, the memory page is copied by the kernel only when needed



Instagram tech stack

Python + Django + uWSGI

uWSGI allows a multi processing model based on fork and leverage the CoW relying on master process initialization

Linux CoW + uWSGI + Python gc

Do you remember the `PyGC_Head` struct we saw earlier?

```
/* GC information is stored BEFORE the object structure. */
typedef struct {
    uintptr_t _gc_next;
    uintptr_t _gc_prev;
} PyGC_Head;
```

The `collect` function uses `_gc_prev` to store a copy of the `ob_refcnt` for **each** container object, and run its cycle detection algorithm

Whenever a gc collection starts after a fork, the algorithm causes **a lot** of memory writes, thus a lot of page faults and, finally, a lot of memory copying after uWSGI fork!

first attempt: disable GC

```
gc.set_threshold(0)
```

Pros

- each process now shares 100 MB more than before
- CPU utilization higher than 10% due to reduced page faults

Cons

- 600 MB leaked memory with 3.000 requests

Unfortunately, it seems that writing reference cycles free code is not so easy for complex application

The growing leaked memory forced them to restart the server periodically, washing out the performance improvements gained after disabling gc

second attempt: gc.freeze()

Objective:

- do not poke objects allocated before fork (objects in parent process)
- continue to collect all objects allocated after fork (objects in child processes)

Zekun Li, a sw eng from Instagram, contributed a patch to Python 3.7 to introduce `gc.freeze()`

```
static PyObject * gc_freeze_impl(PyObject *module)
{
    for (int i = 0; i < NUM_GENERATIONS; ++i) {
        gc_list_merge(GEN_HEAD(i), &_PyRuntime.gc.permanent_generation.head);
        _PyRuntime.gc.generations[i].count = 0;
    }
    Py_RETURN_NONE;
}
```

That's what that `permanent_generation` is for: it holds objects *hidden* from the garbage collection process!

for further details, see Zekun Li's talk at Pycon 2018

PyPy garbage collection

What about PyPy?

PyPy does not use reference counting, but relies entirely on a *generational stop-the-world garbage collector*

The collections are divided into two categories:

- minor collections, regarding a limited number of newly allocated objects
- major collections, regarding all the other objects in the heap

major collections are the ones that cause longer GC pauses

incminimark

To limit the extension of these pauses, the major collections are splitted into pieces, to be executed **incrementally**. That's why the PyPy garbage collector has been called **incminimark**.

Each gc piece (be it a mark or a sweep one) is executed after a minor collection, until the major collection process is complete.

There is one problem, though…

incminimark

To limit the extension of these pauses, the major collections are splitted into pieces, to be executed **incrementally**. That's why the PyPy garbage collector has been called **incminimark**.

Each gc piece (be it a mark or a sweep one) is executed after a minor collection, until the major collection process is complete.

There is one problem, though…

Since the collection process is splitted, the program now runs between the steps, potentially changing the references to the objects

incminimark

To limit the extension of these pauses, the major collections are splitted into pieces, to be executed **incrementally**. That's why the PyPy garbage collector has been called **incminimark**.

Each gc piece (be it a mark or a sweep one) is executed after a minor collection, until the major collection process is complete.

There is one problem, though...

Since the collection process is splitted, the program now runs between the steps, potentially changing the references to the objects

How this can be solved?

Marking and sweeping

Regarding the sweep phase, unreachable objects will surely remain as such, so no issues arise

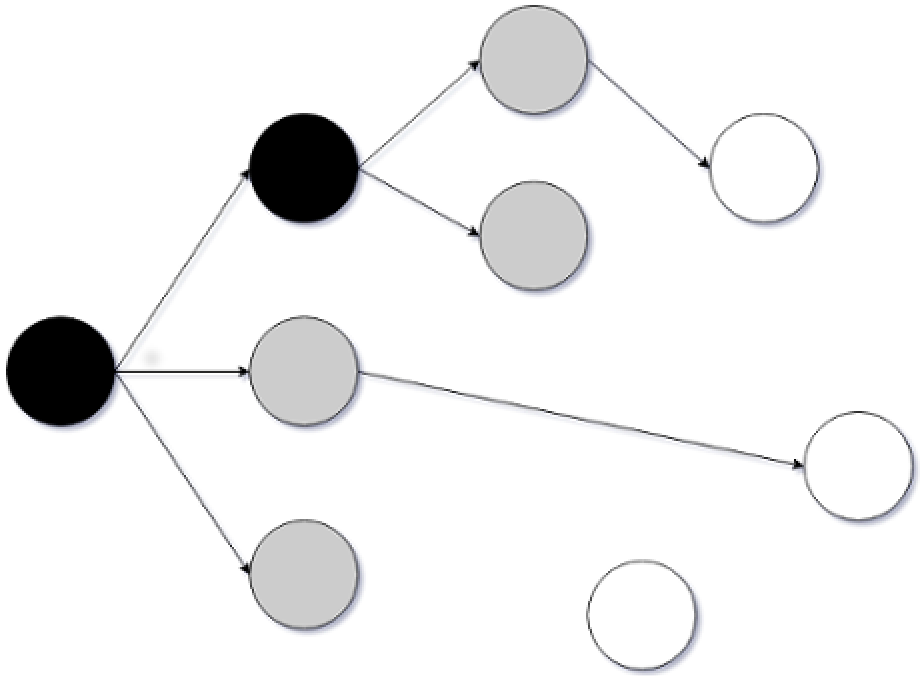
The marking phase is what is really causing troubles

To understand why we need to further inspect the inner details of the marking algorithm

Tri-color marking

During the **tri-color** marking phase, objects:

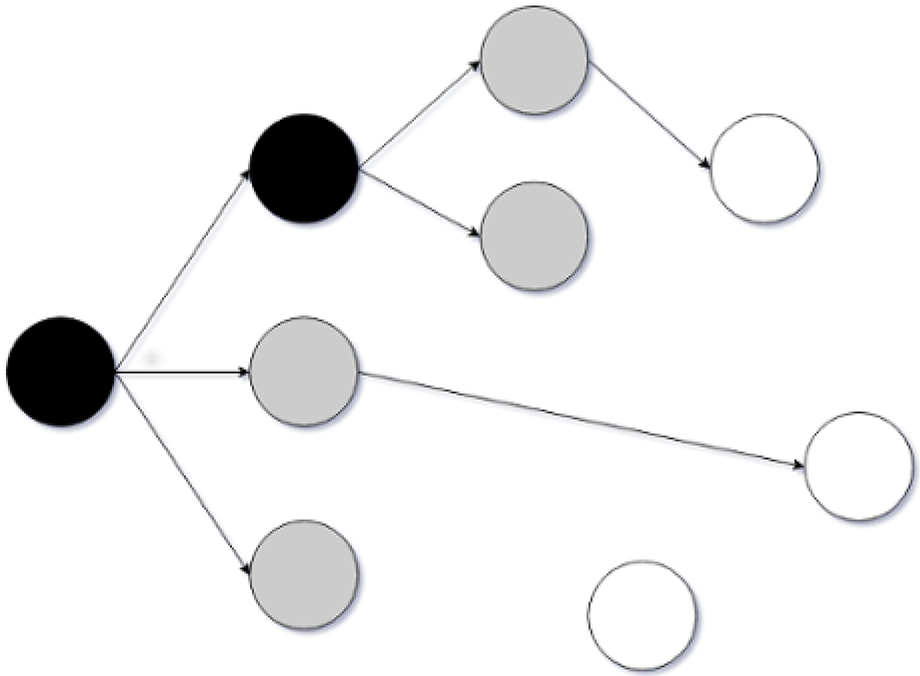
- start as white at the beginning
- become grey when they are found to be alive
- finally become black when all their references have been traversed



Tri-color marking

During the **tri-color** marking phase, objects:

- start as white at the beginning
- become grey when they are found to be alive
- finally become black when all their references have been traversed



In other words, the coloring of an object always follows this order: **white → grey → black**

Tri-color marking invariant

Note that the described algorithm must hold the **tri-color inviariant** to work properly:

at any moment, no black objects should reference white objects

Otherwise, the white object would result as reachable through the reference inside the black object

Tri-color marking invariant

Note that the described algorithm must hold the **tri-color inviariant** to work properly:

at any moment, no black objects should reference white objects

Otherwise, the white object would result as reachable through the reference inside the black object

But since our program is running between the incremental marking steps, it may modify an already scanned object (a black one) to point to a temporarily unreachable object (a white one)!!!

Write barrier

The PyPy GC solves this problem introducing a **write barrier**

The write barrier simply keeps track of all objects being modified, to enqueue them for further marking

In particular, this kind of write barrier is a **backward write barrier**, since it colors the modified objects from black to grey, the opposite of the usual direction

Write barrier

The PyPy GC solves this problem introducing a **write barrier**

The write barrier simply keeps track of all objects being modified, to enqueue them for further marking

In particular, this kind of write barrier is a **backward write barrier**, since it colors the modified objects from black to grey, the opposite of the usual direction

N.B. the write barrier introduces a performance overhead due to the additional check while modifying any reference.

incminimark performance notes

incminimark uses a classical stop-the-world approach, but it *spreads* the gc work in incremental phases, to bound the introduced latency

The typical introduced latency is ~20-30 ms

incminimark performance notes

incminimark uses a classical stop-the-world approach, but it *spreads* the gc work in incremental phases, to bound the introduced latency

The typical introduced latency is ~20-30 ms

When having a predictable latency is a major constraint, PyPy allows to switch to a **semi-manual GC management**, in order to move the long GC pauses where feasible

incminimark performance notes

incminimark uses a classical stop-the-world approach, but it *spreads* the gc work in incremental phases, to bound the introduced latency

The typical introduced latency is ~20-30 ms

When having a predictable latency is a major constraint, PyPy allows to switch to a **semi-manual GC management**, in order to move the long GC pauses where feasible

Two new features have been introduced in PyPy v7.0.0:

- `gc.disable()` to totally disable the major collections
- `gc.collect_step()` to manually run a single step of the major collection process

Go garbage collection

What about Go?

The gc used by Go is a **concurrent, tri-color, mark & sweep garbage collector**, strongly optimized for low latency performance

Differently from PyPy and CPython, Go gc is not generational

Google's engineers considered a generational approach, but finally gave up because:

The write barrier was fast but it simply wasn’t fast enough

– Rick Hudson –

Concurrent garbage collection

The current algorithm does more work than a generational one, but can be executed (mostly) **concurrently** to the goroutines that modify the references.

This kind of architecture allowed Go to fulfill the (impressive) *Service Level Objective* of ~500 microseconds **Stop The World** pause per GC cycle

Still curious?

If you want to know more about Go garbage collection…

Still curious?

If you want to know more about Go garbage collection…

Don't miss my talk on the topic at Golab 2019! ;-)

Thank you for your time!

I'd love to hear from you

- fabio.falzoi84 {gmail.com}
- github: https://github.com/Pippolo84
- **extended version** of this talk: https://github.com/Pippolo84/an-insight-into-python-garbage-collection

Question Time