How Does Garbage Collection Work in Java?

The Alibaba tech team look into garbage collection in Java, something rarely discussed in the typical introduction to Java.

Alibaba Tech Follow
Dec 27, 2019 · 13 min read

Hello everyone, John is cleaning up the garbage here. Can you go out?

When I was at school, there was this one popular meme (similar to the one above) that said if people cleaned up the tableware after finishing a meal in the cafeteria, they had to be C++ programmers. The meme further goes on to say that those who left directly after finishing a meal had to be Java programmers.

Indeed, this meme has a bit of truth to it. Generally, it seems to be the case that we do not need to focus so much on garbage collection (abbreviated as GC) in Java, as we would in C++. Many beginners can still develop a program or system that works, or even works good, without the understanding of GC. However, this does not mean that GC in Java is not important. On the contrary, it is very important but it is also complex, and therefore often different for the beginners.

However, it's not all bad. Let's dive into garbage collection (GC) in Java today and see what's all involved in it.

What Is Garbage Collection?

Garbage collection (GC), as its name implies, is a means of freeing space occupied by waste materials, or garbage, and avoid memory leaks. Through performing the GC mechanism, available memory can be effectively used. Moreover, through this process, objects that are dead or unused for a long time in the memory heap will be deleted and the memory space used by these objects will be reclaimed.

Before the Java language appeared, programmers devoted themselves to writing C or C++ programs. At such a time, a seriously contradictory phenomenon existed. When creating objects in C++ and other languages, you had to constantly allocate space. When you did not use these objects, you then had to also release space. Therefore, you had to write both constructors and destructors. In many cases, both functions were repeated for memory allocation and reclamation. Then, someone suggested that if we could write code to realize this purpose. When you allocate and release space, you can reuse the code without repeatedly writing both functions.

In 1960, the concept of GC was first proposed in the MIT-based Lisp, and Java was not yet invented at that time. In fact, GC is not a Java patent. The history of GC is much longer than that of Java.

How to Define Garbage

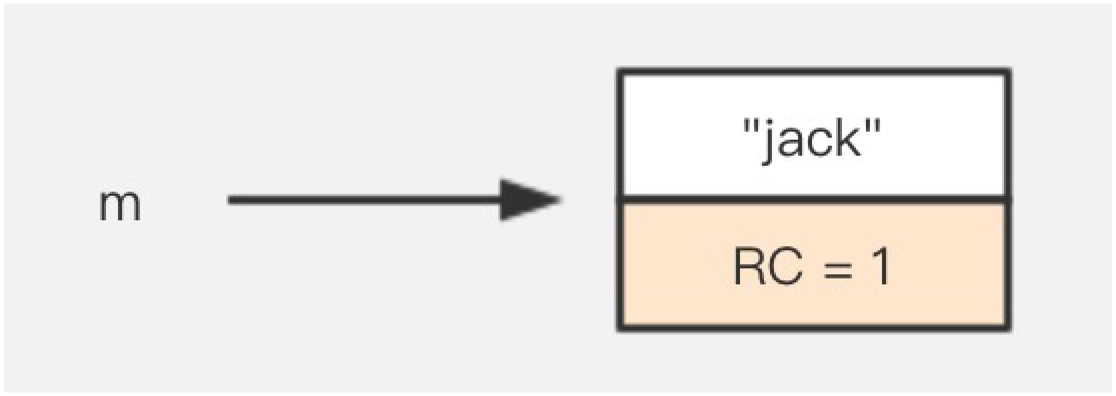
To perform GC, we must first understand what garbage is and figure out which memory space needs to be reclaimed.

Reference Counting Algorithm

The Reference Counting Algorithm allocates a field in the object header to store the reference count of the object. If this object is referenced by another object, its reference count increments by one. If the reference to this object is deleted, the reference count decrements by one. When the reference count of this object drops to zero, the object will be garbage-collected.

```
String m = new String("jack");
```

First, let’s create a string in which “jack” is referenced by m.



Then, set m to null. The reference count of “jack” is zero. In the Reference Counting algorithm, the memory for “jack” is to be reclaimed.

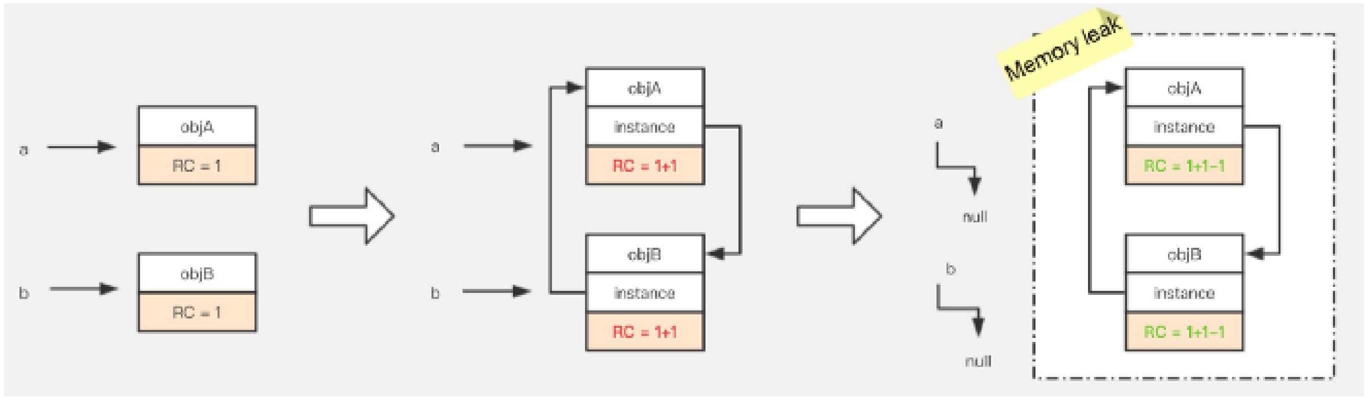
```
m = null;
```

The Reference Counting Algorithm performs GC in the execution of the program. This algorithm does not trigger Stop-The-World events. Stop-The-World means that the execution of the program is suspended for GC till all objects in the heap are processed. Therefore, this algorithm does not strictly follow the Stop-The-World GC mechanism.

It looks pretty applicable to GC. However, we know that GC on the Java virtual machine (JVM) follows the Stop-The-World mechanism. Why did we give up the Reference Counting algorithm? Let’s look at the following example:

```
public class ReferenceCountingGC {

    public Object instance;

    public ReferenceCountingGC(String name){}
}

public static void testGC(){

    ReferenceCountingGC a = new ReferenceCountingGC("objA");
    ReferenceCountingGC b = new ReferenceCountingGC("objB");

    a.instance = b;
    b.instance = a;

    a = null;
    b = null;
}
```

We first define two objects, then make mutual reference to the objects, and lastly set references for each object to null.

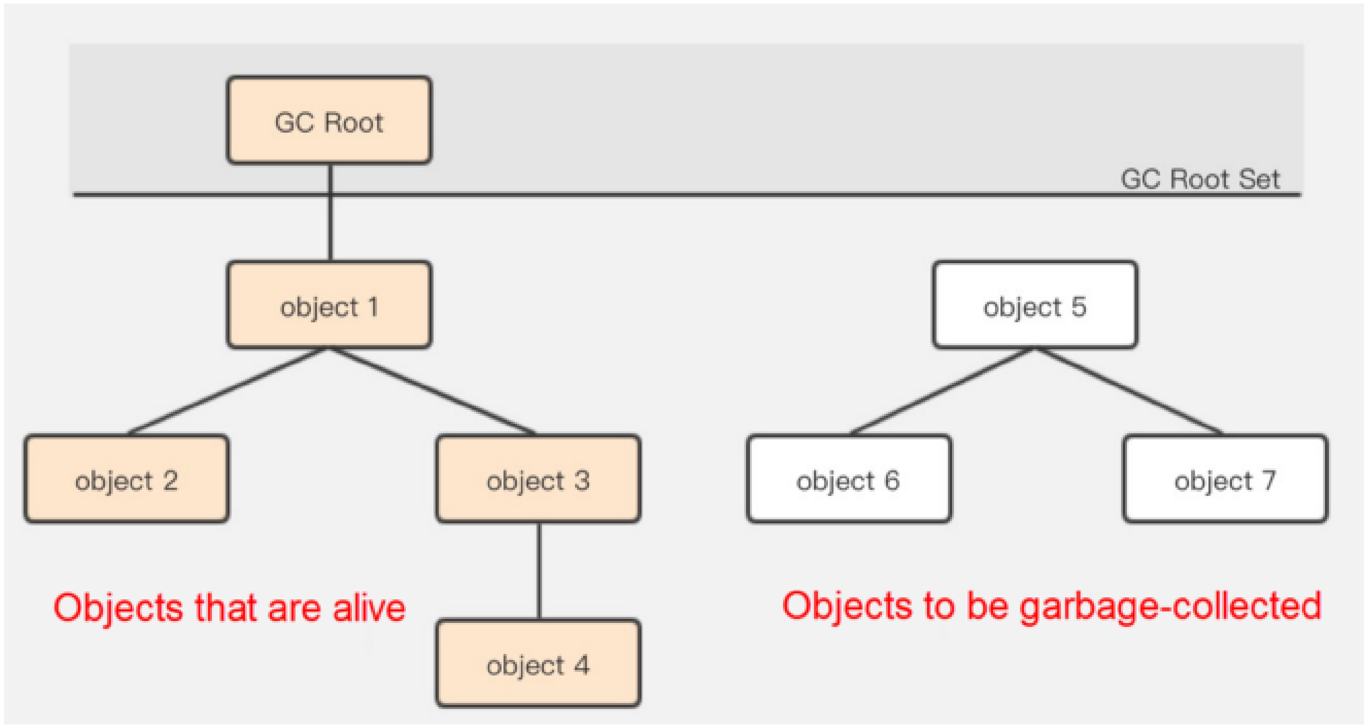


We can see that both objects can no longer be accessed. However, they are referenced by each other, and thus their reference count will never be zero. Consequently, the GC collector will never be notified to garbage collect them by using the Reference Counting algorithm.

Reachability Analysis Algorithm

The basic idea of the Reachability Analysis Algorithm is to start from GC roots. GC traverses the whole object graph in the memory, starting from these roots and following references from the roots to other objects. The path is called the reference chain. If an object has no reference chain to the

GC roots, that is the object cannot be reached from the GC roots, the object is unavailable.

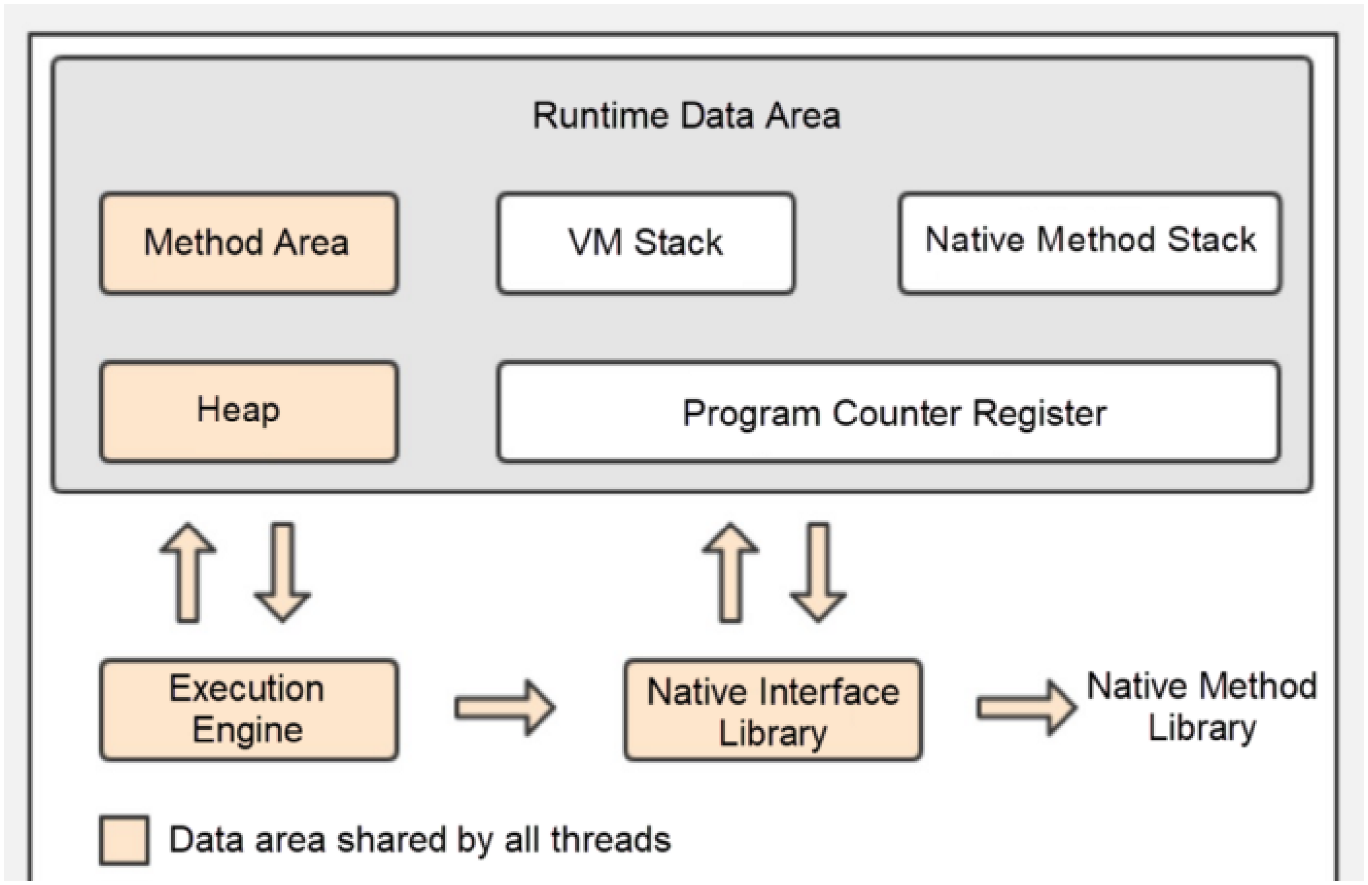


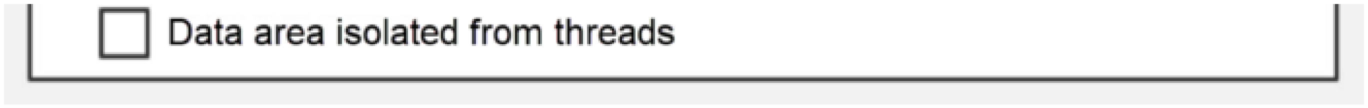
The Reachability Analysis algorithm successfully solves the problem of cyclic references in the Reference Counting algorithm. As long as an object cannot establish a direct or indirect connection with the GC roots, the system determines that the object is to be garbage-collected. Then, another question arises. What are GC roots?

Java Memory Space

In Java, GC roots can be four types of objects:

- Objects referenced in the virtual machine (VM) stack, that is the local variable table in the stack frame
- Objects referenced by class static attributes in the method area
- Objects referenced by constants in the method area
- Objects referenced by JNI (the Native method) in the native method stack





Objects referenced in the VM stack, that is the local variable table in the stack frame

In this case, s is the GC root. When s is set to null, the localParameter object has its reference chain with the GC root broken, and the object will be garbage-collected.

```
public class StackLocalParameter {
    public StackLocalParameter(String name){}
}

public static void testGC(){
    StackLocalParameter s = new
StackLocalParameter("localParameter");
    s = null;
}
```

Objects referenced by class static attributes in the method area

When s is the GC root and s is set to null, after GC, the properties object to which s points is garbage-collected because it cannot establish a connection with the GC root. As a class static attribute, m is also a GC root. The parameter object is still connected to the GC root, so the parameter object will not be garbage-collected in this case.

```
public class MethodAreaStaicProperties {
    public static MethodAreaStaicProperties m;
    public MethodAreaStaicProperties(String name){}
}

public static void testGC(){
    MethodAreaStaicProperties s = new
MethodAreaStaicProperties("properties");
    s.m = new MethodAreaStaicProperties("parameter");
    s = null;
}
```

Objects referenced by constants in the method area

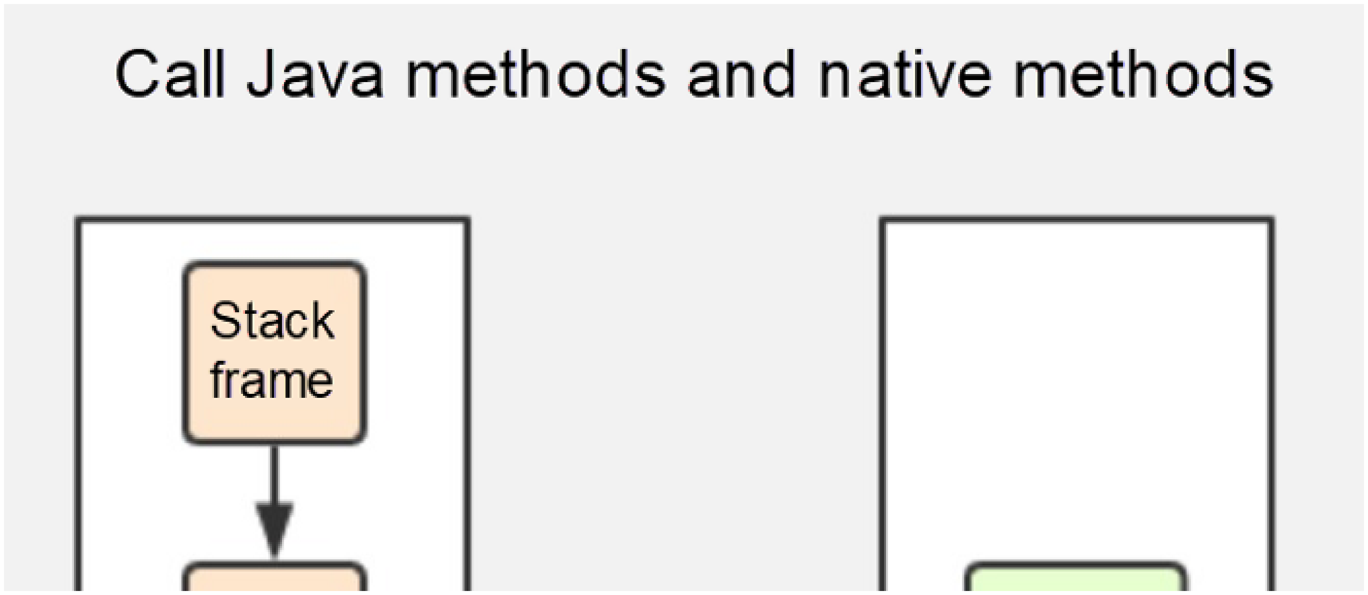
As a constant reference in the method area, m is also the GC root. After s is set to null, the final object will not be garbage-collected though it has no reference chain with the GC root.

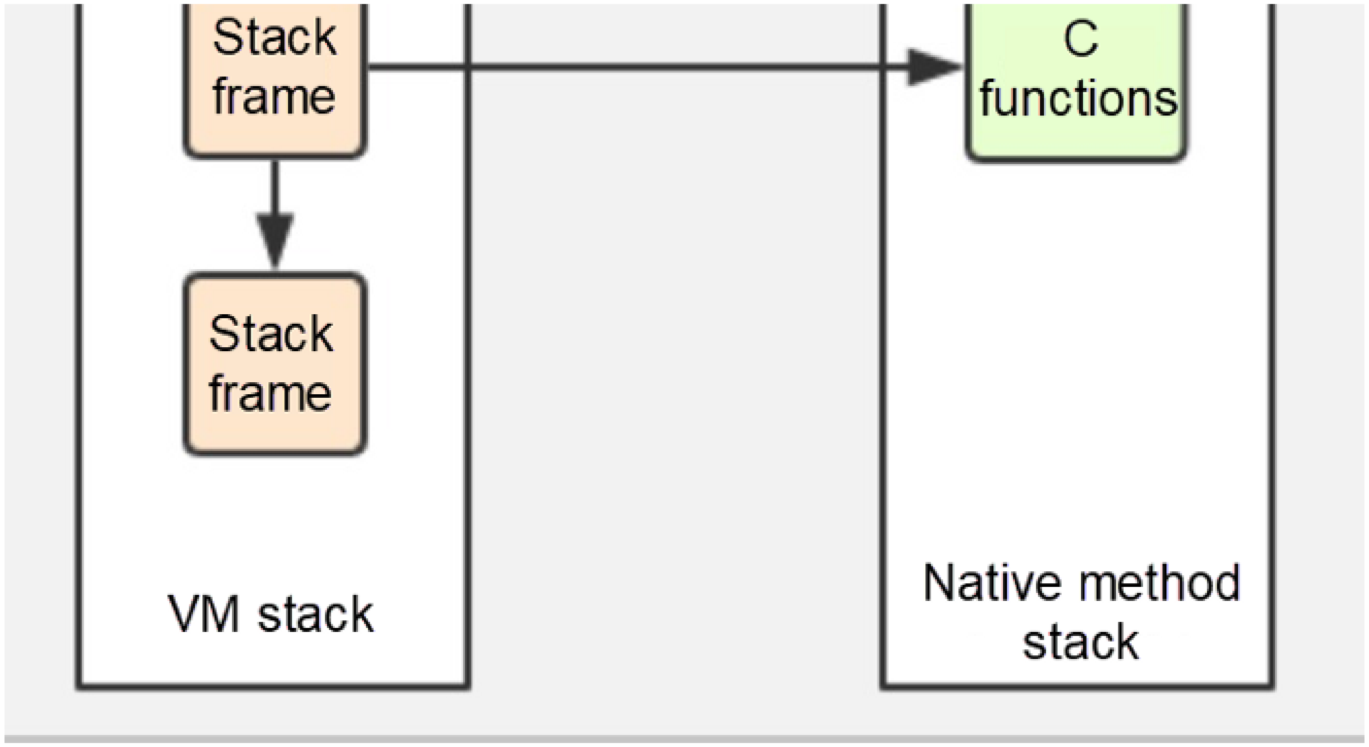
```
public class MethodAreaStaicProperties {
    public static final MethodAreaStaicProperties m =
MethodAreaStaicProperties("final");
    public MethodAreaStaicProperties(String name){}
}
```

```
public static void testGC(){
    MethodAreaStaicProperties s = new
MethodAreaStaicProperties("staticProperties");
    s = null;
}
```

Objects referenced in the native method stack

A native interface always uses a native method stack. If the native method interface is implemented by using the C connection model, its native method stack is the C stack. When a thread calls the Java method, the VM creates a new stack frame and puts it in the Java stack. However, when it calls the native method, the VM keeps the Java stack unchanged and no longer puts new frames in the thread's Java stack. Instead, the VM dynamically connects to and directly calls the specified native method.

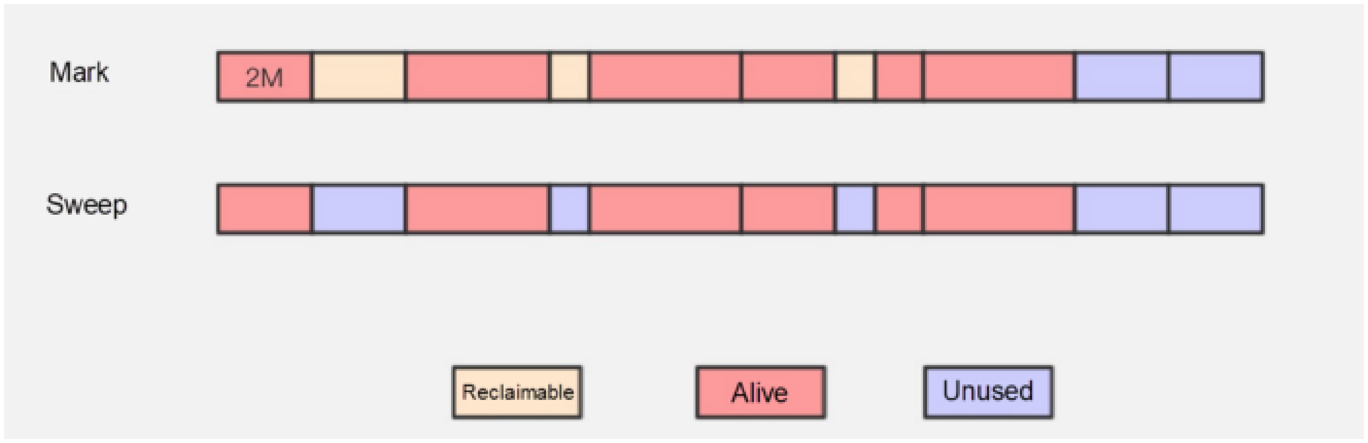




How Does GC Work

After determining the garbage to be collected, the garbage collector starts its work. However, this involves a question: How can we efficiently perform GC? JVM specifications do not clearly define how to implement the garbage collector. Therefore, VMs from different manufacturers can implement the garbage collector in different ways. The following talks about the core ideas of several common GC algorithms.

Mark-Sweep Algorithm

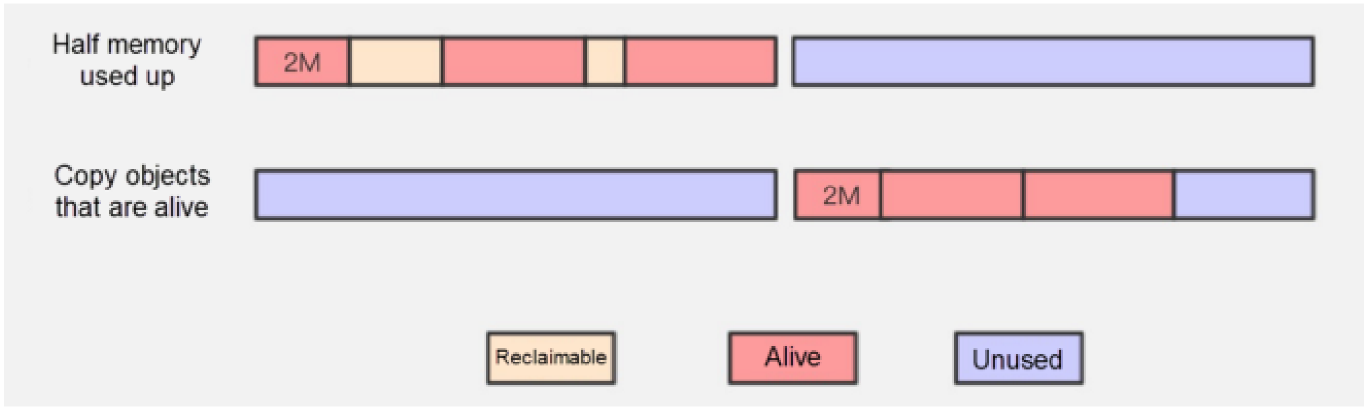


The Mark-Sweep algorithm is the most common garbage collection algorithm, which performs two operations. It first marks the objects to be garbage-collected in the memory space and then clears the marked objects up from the heap. As shown in the preceding figure, the memory space occupied by the garbage becomes unoccupied after collection and is ready for reuse.

The algorithm has a clear logic and is easy for operation. However, it has a big problem of memory fragmentation.

In the preceding figure, the medium square is assumed to be 2 MB, the smaller one is 1 MB, and the larger one is 4 MB. After these memory spaces are reclaimed, memory is available in many segments. We know that memory can be allocated only in contiguous form of blocks. If we need a 2 MB memory space in this case, the two 1 MB spaces cannot be utilized. As a result, many memory segments are wasted.

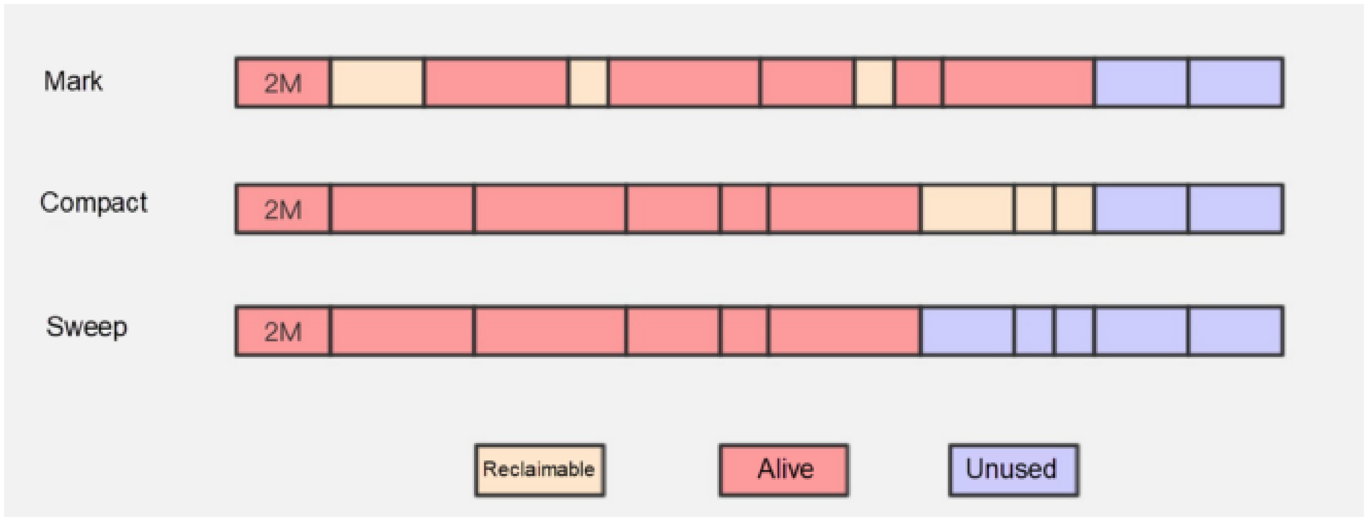
Copying Algorithm



The Copying algorithm evolves from the Mark-Sweep algorithm to solve the problem of memory fragmentation. This algorithm divides available memory into two equally sized semi-spaces. Only one semi-space is active at a time. When the active semi-space becomes full, living objects are copied to the other semi-space. Then, the active but full memory space is cleared up. This ensures that memory can be contiguously allocated. As a result, complex situations such as memory fragmentation do not occur during memory allocation. Meanwhile, the logic is clear and the operation is efficient.

The preceding figure also exposes another problem. That is, only half of the memory can be actually used. This cost is too high.

Mark-Compact Algorithm



The Mark-Compact algorithm has the same marking process as the Mark-Sweep algorithm. However, this algorithm does not directly clear up the objects that can be garbage-collected. Instead, it moves all living objects to one end, and then reclaims the memory space beyond the end boundary.

Upgraded from the Mark-Sweep algorithm, the Mark-Compact algorithm solves the problem of memory fragmentation. In addition, the algorithm avoids the demerit that only half of the memory space can be used in the Copying algorithm. This algorithm seems pretty good. Unfortunately, as shown in the preceding figure, it makes more frequent changes to the memory and needs to sort out the reference addresses of all living objects, which is much less efficient than the Copying algorithm.

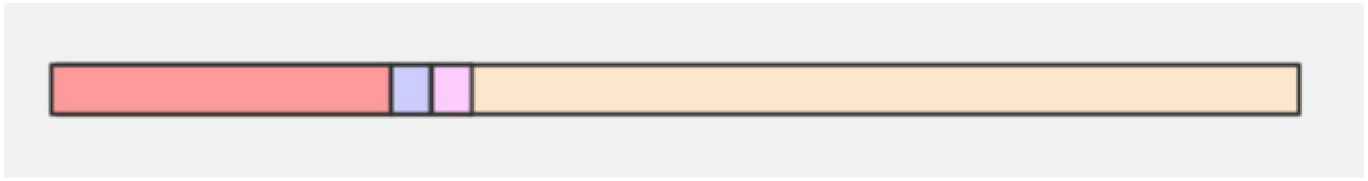
Generational Collection Algorithm

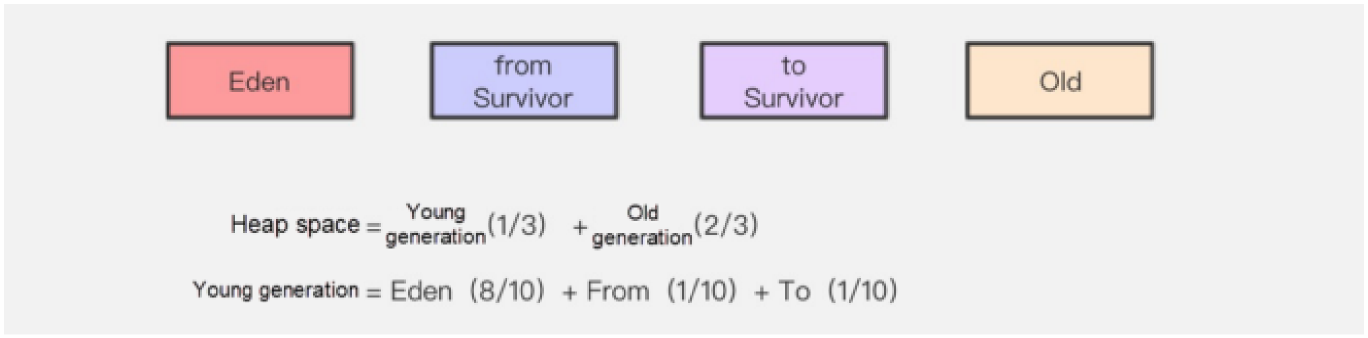
Strictly speaking, the Generational Collection algorithm is not an idea or a theory, but a combination of the first three algorithms. It provides the combination of different algorithms for different scenarios.

Memory is divided into blocks according to different lifespans of objects. Generally speaking, the Java heap is divided into the young generation and the old generation. You can use the most appropriate GC algorithm that is applicable to each generation. In the young generation, a large number of dead objects always can be found after a GC cycle and only a few objects survive. In this case, the Copying algorithm is adopted to complete the collection by copying only a few living objects. In the old generation, the Mark-Sweep algorithm or the Mark-Compact algorithm is adopted because the objects have a high survival rate and no extra memory space is reserved specially for allocation.

So, more questions arise: What parts is the memory divided into, and which algorithm is applicable to each part?

Memory Model and Collection Policy




Eden
from Survivor
to Survivor
Old
Heap space = Young generation (1/3) + Old generation (2/3)
Young generation = Eden (8/10) + From (1/10) + To (1/10)

The Java heap is the largest memory managed by JVM, and the heap is the main space that the garbage collector manages. Here, we mainly analyze the structure of the Java heap.

The Java heap is mainly divided into two spaces: the young generation and the old generation. The young generation is divided into the Eden space and the Survivor space, while the Survivor space is further divided into the From space and the To space. We may have the following questions: Why is the Survivor space required? Why is the Survivor space subdivided into two more spaces? Take it easy. Let's have a detailed look at how an object is created and deleted.

Eden Space

As shown in a professional research conducted by IBM, nearly 98% of the objects are short-lived. As a result, objects are mainly allocated in the Eden space of the young generation. When the Eden space is not large enough for memory allocation, the VM initiates a minor GC. Minor GC occurs more frequently than major GC and has a faster collection speed. After a minor GC, the Eden space will be cleared up, and most objects in the Eden space will be garbage-collected. The objects that survive the minor GC will be moved to the From space. If the From space is not large enough, these objects will be moved directly to the old generation.

Survivor Space

Resembling the yellow lights in traffic, the Survivor space acts like a buffer between the Eden space and the old generation. The Survivor space is subdivided into two more spaces: One is the From space and the other is the To space. Objects that survive each minor GC in the Eden space and the From space are moved to the To space. If the To space is not large enough, they are directly promoted to the old generation.

Why Is the Survivor Space Required?

It seem that an object goes from the Eden space of the new generation to the old generation. Why is it so complicated? Assume that there is no Survivor space, living objects in the Eden space will then be sent to the old generation after each minor GC. Therefore, the old generation will soon be filled up. Although many objects cannot be deleted after a minor GC, they do not live long. They may be cleared up by the second or third minor GC. It is not a wise decision to move them to the old generation after a minor GC.

Therefore, the Survivor space exists to reduce the number of objects sent to the old generation and thus reduces the occurrence of major GC. The pre-screening by the Survivor space ensures that only objects that can survive 16 minor GCs will be promoted to the old generation.

Why Is the Survivor Space Further Divided into Two Spaces?

The biggest benefit of the division is to solve the problem of memory fragmentation.

Let’s assume that there is only one Survivor space. After a minor GC, the Eden space is cleared up and living objects are moved to the Survivor space. Objects that previously exist in the Survivor space may also need to be deleted. Then, a question arises: How can we clear them up? In this scenario, we can only use the Mark-Sweep algorithm, but we know that the biggest problem with this algorithm is memory fragmentation. In the young generation where objects are short-lived, the Mark-Sweep algorithm will inevitably cause serious memory fragmentation. Due to the two survivor spaces, surviving objects in the previous Eden and From spaces will be copied to the To space after a minor GC. In the second Minor GC, the roles of the From and To spaces are exchanged. The objects that survive in the Eden and To spaces are copied to the From space. This process of copying the living objects between both survivor spaces is repeated several times.

What can benefit from this mechanism most is that there will always be an empty survivor space, and the other survivor space will always be free of fragmentation. So, why is the Survivor space subdivided into more spaces? For example, is it feasible to divide the Survivor space into three, four, or five spaces? Well, if the Survivor space is further subdivided, the space of each partition will be relatively small and can be easily filled up. Therefore, two survivor spaces may be the best specification of the solution after trade-offs.

Old Generation

The old generation occupies two thirds of the heap memory space, which is cleared up only when a major GC is performed. Each GC triggers Stop-The-World events. The larger the memory, the longer the Stop-The-World time. So, it is not always better to have a larger memory. The Coping algorithm performs many copying operations in the old generation that has a high object survival rate, which results in low efficiency. Consequently, the Mark-Compact algorithm is adopted in the old generation.

In addition, with the help of the promotion failure handling mechanism, objects that cannot be placed in the Survivor space will be directly promoted to the old generation. The following objects will also be placed in the old generation.

Large Objects

A large object refers to an object that requires a large amount of contiguous memory space. A large object will go directly to the old generation regardless of its lifespan. Large objects are placed in the old generation so that a large amount of copying operations can be avoided between the Eden space and both survivor spaces. Pay more attention if there are plenty of short-lived large objects in your system.

Long-lived Objects

The VM sets an age counter for each object. In normal conditions, objects are constantly moving between the From and To survivor spaces. After objects survive a minor GC, they have their age incremented by one. When the age of an object is increased to 15, it will be promoted to the old generation. Of course, JVM also supports setting the age threshold.

Dynamic Object Age

The JVM does not require that the age of an object must be 15 so that it can be promoted to the old generation. If objects of the same age occupy more than half of the Survivor space, objects of greater than or equal to this age can be directly promoted to the old generation without waiting be "old enough."

This actually acts a bit like a load balancer. Polling is a type of load balancer that ensures each machine can receive the same request. It seems like balanced, but the hardware of each machine is disconnected and their status is different. We can adjust our load balancing algorithm based on the number of requests received by each machine or the response time of each machine.

(Original article by Nie Xiaolong聂晓龙)

. . .

Alibaba Tech

First hand and in-depth information about Alibaba’s latest technology → Facebook: **“Alibaba Tech”**. Twitter: **“AlibabaTech”**.

Programming Java Coding C Technology