Abstract—The Java garbage collector is one of the important modules of the Java Virtual Machine (JVM). When an object has no references, it is considered "garbage". The memory space occupied by this object needs to be freed. The role of the Java garbage collector is to manage memory and release the memory occupied by this "garbage". This article introduces the main function and working principle of the Java garbage collector, focusing on three mainstream garbage collection algorithms and compares serial, parallel, and concurrent design choices. This article also introduces 7 garbage collectors in Java based on different algorithms and design choices: Serial, Serial Old, ParNew, Parallel, Scavenge, Parallel Old, CMS, and G1.

Keywords—Java garbage collector, JVM, Memory management.

I. INTRODUCTION

Memory management is the process of recognizing when allocated objects are no longer needed, deallocating (freeing) the memory used by such objects, and making it available for subsequent allocations [1]. There are two ways to implement dynamic memory management: Explicit and Automatic Memory Management. The garbage collector is a solution to automatic memory management. The Java programming language implements Automatic Memory Management, which is the java garbage collector. The Java garbage collector is one of three important modules of the Java Virtual Machine (JVM) (the other two being the interpreter and multithreading mechanism) [2]. Its role is to provide applications for automatic memory allocation and automatic garbage collection.

The Java heap is the main area managed by the garbage collector, also known as the "GC heap", which is a memory area shared by all threads. The purpose of heap memory is to store object instances, that is, objects and arrays created by "new" and instance variables of objects [3]. Generally speaking, for an object stored on the heap, if multiple references are pointing to it, the object is "live", otherwise it is "dead" and considered garbage that needs to be collected.

II. GENERATIONAL GARBAGE COLLECTION STRATEGY

Because the life cycle of most objects in Java is short-lived, the generational recycling strategy is used in the Java garbage collector to manage the memory of the heap generationally. The purpose of "generational collection" is to use different management strategies (collection algorithms) for different generations of memory blocks to maximize performance [4]. Generally, the Java heap is divided into the young generation, the old generation, and the persistent generation [5]. It is worth noting that no matter how it is divided, it has nothing to do with the content. No matter which area, all object instances are stored.

A. Minor GC

In most cases, objects are allocated to the young generation. In the young generation, there is a space called Eden Space, which is mainly used to store new objects. In addition to the Eden space, there are From Survivor and To Survivor spaces. These two areas are equal in size, equivalent to the two areas in the copying algorithm, which are used to store objects that survive each garbage collection.

Garbage collection in the young generation is called Minor GC. When Eden space does not have enough space for an allocation, the newly created object cannot be placed in Eden Space, and the virtual machine will trigger a Minor GC at this time.

The main processes of Minor GC are as follows:

1) In the initial state, the newly created object is allocated to the Eden area, and the two spaces (From and To) of the survivor are empty.
2) When there is not enough space in ‘Eden’ space, the Space objects of ‘Eden’ and ‘From’ will be copied to ‘To’ space, and then ‘Eden’ and ‘From’ space will be emptied. The first garbage collection is done. At this point, ‘Eden’ space and ‘From’ space has been emptied, and “live” object are tightly stored in “To” space.
3) When the 'Eden' space is full again, the "live" objects of the 'Eden' space and the 'To' space will be copied to the 'From' space. Then, the 'Eden' space and the 'To' space are emptied. At this point, the live object is stored in the 'From' space.
4) The above steps are repeated.

Figure 1: copy algorithm

This method of garbage collection by repeatedly copying through two spaces is called the copying algorithm. This copying algorithm is simple to implement and efficient to run.
Because the memory is freed for one of the spaces each time, there is no need to consider memory fragmentation when allocating memory. Just move the top pointer of the heap and allocate memory in sequence.

**B. From the Young Generation to the Old Generation**

Objects will be moved to the old generation in the following cases:

1) When the Minor GC is executed, the live objects in the 'Eden' and 'From' places will be moved to the 'To' place. If the 'To' place cannot be loaded at this time, the objects will be directly moved to the old generation.

2) "Old" objects in the 'From' place will be moved to the old generation. Even if the 'To' place is not full, the JVM will still move objects that are old enough for the old generation. (The "age" of the object: If the object is born in the Eden space, still alive after the first Minor GC, and can be accommodated by the Survivor space, then the object will be moved to the Survivor space, and the age will be set to 1. In survivor space, an object survives once from minor GC, and its age increases by 1 year. When its age increases to a certain age (default 15), it will be transferred to the old generation. The threshold for object promotion to the old generation age can be set by the parameter -XX:MaxTenuringThreshold.)

3) In the Survivor space, if the sum of the size of the objects with the same age is bigger than half of the Survivor space, the objects whose age is bigger than or equal to this age are directly transferred to the old generation, without waiting for the age to required in MaxTenuringThreshold.

4) Large objects are allocated directly to the old generation. Large objects are a nuisance to virtual machine memory allocation. Large objects tend to trigger garbage collection early when there is still a lot of space in memory, because enough contiguous space needs to be arranged to accommodate them. The virtual machine provides a -XX:PretenureSizeThreshold parameter, which allows objects larger than this setting value to be allocated directly in the old generation. The purpose is to avoid a large number of memory copies between the Eden space and the two Survivor spaces.

**C. Garbage collection mechanism in old generation**

The old generation mainly stores memory objects with a long life cycle. If the old generation is full of objects and new objects cannot be moved from the young generation, a major collection (major GC) will be triggered. In the old generation garbage collection generally uses mark-compact or mark-and-sweep algorithm. The "mark" algorithm is slower and less efficient than the copy algorithm, but it reduces memory requirements.

**Algorithm mark-and-sweep.** The mark-sweep algorithm can be divided into two stages: mark and clear. First, mark all reachable objects starting from the root node. Unmarked objects are unreferenced garbage objects. Second clear all unmarked objects. The main problem with this algorithm is that it will generate a large number of discontinuous memory fragments. Too much fragmentation will result in not being able to find enough contiguous memory when allocating larger objects, in which case another garbage collection will have to be triggered early.

**Algorithm mark-compact.** The algorithm mark-compact is optimized based on the mark-and-sweep algorithm. It solves the memory fragmentation problem of the Mark-and-Sweep algorithm. The mark-compact algorithm can be divided into three processes: mark, sweep, and compact. In the mark phase, still the same as the mark-sweep algorithm, through the root node, all reachable objects starting from the root node are marked. Then there is the compact phase, which compacts all live objects into a part of memory (the first part of the old generation). Finally, there is the sweep phase, which cleans up memory beyond the end boundary and frees all dead objects.

Both the mark-and-sweep algorithm and the mark-compact algorithm sacrifice efficiency in exchange for memory utilization. Therefore, they are more suitable for situations where there are many surviving objects such as in the old generation.

Compare three garbage collection algorithms:

1) In terms of efficiency (time complexity): the copying algorithm is the highest, followed by mark-compact, and Mark-and-Sweep is the lowest.

2) For memory utilization, Mark-Compact has the highest, followed by mark-and-sweep, and the copying algorithm has the lowest.

3) From the point of view of memory neatness, both the copying algorithm and the mark-compact algorithm can effectively solve the problem of memory fragmentation, while mark-and-sweep will produce a large amount of memory fragmentation.

In fact, in java, in addition to the young generation and the old generation, there is a third-generation - the permanent generation. The permanent generation is mainly used to put the JVM's objects, such as class objects, method objects, member variable objects, and constructor objects. The permanent generation does not participate in recycling.

**III. SINGLE-THREADED SERIAL RECYCLING STRATEGY AND MULTI-THREADED PARALLEL RECYCLING STRATEGY**

Single-threaded serial means that only one CPU is used to perform garbage collection tasks, even if there are multiple CPUs available at the time. Its advantages are simple, easy to implement, and less fragmented, suitable for single-core computers.

The strategy of the parallel collection is to divide the garbage collection task into multiple parts and let these subparts execute simultaneously on different CPUs. The multi-threaded parallel recycling strategy can make full use of the CPU resources of multi-core machines, reduce garbage collection time and increase efficiency [6]. Its disadvantage is that it is complex and may cause some fragments not to be recovered.
IV. CONCURRENT AND STOP-THE-WORLD

Concurrent means that garbage collection and application are performed concurrently. Stop-the-world means to stop the application thread when recycling. The advantages of stop-the-world are simplicity, precision, cleaner, and less time for garbage collection because the garbage collector can monopolize CPU resources. However, its disadvantage is that after stopping the application thread, the response time of the application during the garbage collection cycle will be longer, so it is not suitable for systems with very high real-time requirements. The advantage of concurrent is that the response time of the application is smoother and more stable [7]. The disadvantage is that it is difficult to implement, frequent cleaning, and possible fragmentation.

V. SEVEN GARBAGE COLLECTORS IN JAVA

The Serial GC. The serial collector is the default in Java SE 5 and 6 [8]. There are two types of serial garbage collectors, the young generation and the old generation. Serial is the default young generation garbage collector for client-style machines. It uses a single CPU or one thread to collect garbage. All other worker threads are stopped during collection until garbage collection is complete. Serial is simple and efficient and is suitable for computers with a single core or a few cores. Serial Old is the default old generation garbage collector in client mode. Like "Serial", it applies to the case of a single-core CPU.

The Parallel GC. There are three kinds of parallel garbage collectors in Java: ParNew, Parallel Scavenge, and Parallel Old. ParNew is a young generation parallel garbage collector that uses a copy algorithm. Think of it as a multithreaded version of "Serial". The default number of threads in "ParNew" is the number of CPU cores. The number of threads can be configured through -XX:ParallelGCThreads. In Server mode, ParNew is the default garbage collector for the young generation. It is efficient and simple and is suitable for multi-core CPU conditions. But like "Serial", "ParNew" stops other worker threads during garbage collection [9].

Parallel Scavenge, like ParNew, is also a young generation parallel garbage collector, which also uses a replication algorithm. But it differs from ParNew in that Parallel Scavenge aims to achieve a manageable throughput. High throughput can make the most efficient use of CPU time and complete the program's computing tasks as quickly as possible [10]. Therefore, Parallel Scavenge is suitable for situations with less interaction and more computation, such as background operations.

\[
\text{Throughput} = \frac{T_{\text{code}}}{T_{\text{code}} + T_{GC}} \quad (1)
\]

Where \(T_{\text{code}}\) is the runtime of the user's code, and \(T_{GC}\) is the time of garbage collection.

Parallel Old is an old-generation parallel garbage collector provided in JDK 1.6, which uses the mark-compact algorithm. Parallel Old to cope with high throughput requirements in the old generation. When the system requires high throughput, a combination of the young generation Parallel Scavenge and the old generation Parallel Scavenge can be used to obtain the most efficient CPU multi-core utilization.

The Concurrent Mark Sweep (CMS) Collector. CMS is an old-generation garbage collector [11]. CMS minimizes pause time during garbage collection. Marking and sweeping are concurrent, and garbage collection works with user threads, so there is no need to stop worker threads. CMS is suitable for programs with high interaction requirements and can significantly improve user experience. However, its disadvantage is that it will generate memory fragmentation and floating garbage. In addition, CMS is very sensitive to CPU resources.

The G1 Garbage Collector. The G1 garbage collector is a concurrent garbage collector introduced in jdk1.7 [12]. The G1 garbage collector combines young and old generations. The working principle of G1 is to divide the heap memory into multiple areas and prioritize them side by side, and the area with a lot of garbage will be reclaimed first [13]. By collecting part of the area each time, the stop time generated by GC is reduced, ensuring the highest garbage collection efficiency in a limited time. The advantage of the G1 garbage collector is that it can precisely control the pause time and achieve low-pause garbage collection without sacrificing throughput. And G1 is based on the Mark-Compact algorithm, so there will be no memory fragmentation [14].

Table 1: Comparison of the 7 garbage collectors in java

<table>
<thead>
<tr>
<th>Garbage collector</th>
<th>Serial/Parallel/Concurrent</th>
<th>Young/Old generation</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>Serial</td>
<td>Young generation</td>
<td>Copy algorithm</td>
</tr>
<tr>
<td>Serial Old</td>
<td>Serial</td>
<td>Old generation</td>
<td>Mark-compact</td>
</tr>
<tr>
<td>ParNew</td>
<td>Parallel</td>
<td>Young generation</td>
<td>Copy algorithm</td>
</tr>
<tr>
<td>Parallel Scavenge</td>
<td>Parallel</td>
<td>Young generation</td>
<td>Copy algorithm</td>
</tr>
<tr>
<td>Parallel Old</td>
<td>Parallel</td>
<td>Old generation</td>
<td>Mark-compact</td>
</tr>
<tr>
<td>CMS</td>
<td>Concurrent</td>
<td>Old generation</td>
<td>Mark-and-sweep</td>
</tr>
<tr>
<td>G1</td>
<td>Concurrent</td>
<td>Young generation and old generation</td>
<td>Mark-compact and copy algorithm</td>
</tr>
</tbody>
</table>

VI. PERFORMANCE METRICS

To evaluate the performance of the garbage collector, there are six main performance metrics.

1) Throughput: The proportion of non-garbage collection time to the total time in a long cycle. This metric measures the operational efficiency of the system.

2) Garbage Collection overhead: The percentage of garbage collection time to total time in a long cycle. Garbage Collection overhead and Throughput sum to
100%.
3) Pause time: The time when the program's worker thread is suspended while the Java Virtual Machine is collecting garbage.
4) Frequency of collection: the frequency of garbage collection operations occur.
5) Footprint: Java heap area occupied by the size of memory.
6) Promptness: the time from the death of an object to the memory occupied by the object is released.

The most important of these metrics are throughput, pause time, and footprint [15].

Now, with the development of hardware memory footprint is no longer a major issue.

High throughput gives the user a better experience because it makes the end-user of the application feel like only the application thread is doing the work. Also, low pause times are better because it is always bad for an application to be hung and may interrupt the end-user experience. Some programs are sensitive to pause times, such as interactive applications.

Unfortunately, "high throughput" and "low pause time" are contradictory. If we choose to prioritize throughput, then we must reduce the frequency of garbage collection, which leads to a longer pause time for garbage collection. On the contrary, if we choose to prioritize low latency, we have to execute garbage collection more frequently to reduce the pause time of each garbage collection, but this will cause memory reduction in younger generations and a decrease in program throughput.

Therefore, a garbage collection algorithm either targets one of the two goals (i.e., focuses only on larger throughput or minimum pause time) or tries to find a compromise between the two. Nowadays, the general choice is to reduce the stall time while prioritizing maximum throughput.

It is worth noting that the importance of pause time is becoming more and more important with hardware development. On the one hand, hardware performance improvements have helped to reduce the impact of the collector on the application while it is running, i.e. increasing throughput. On the other hand, the expansion of memory has harmed pause time.

Commonly used Java garbage collector performance evaluation tools are:
1) -XX:+PrintGCDetails and -XX:+PrintGCTimeStamps;
2) jmap [options] pid;
3) jstat [options] pid.
The first method can print the start time, duration, and free memory of each generation of garbage. jmap can view class loading and memory usage. jstat can view GC execution.

VII. TESTING OF JAVA GARBAGE COLLECTION

Test objects: Specify Serial+Serial Old as JVM garbage collector.

A. Check Memory Consumption

First create the following class:

```java
public class GCTest {
    public static void main(String[] args) {
        ...}
``` 

Then use javac to compile and execute:
```
java -verbose:gc -Xms20M -Xmx20M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:+UseSerialGC
``` 

The test results are logged as follows:

```java
[0.184s][info ][gc,heap,exit] Heap
[0.184s][info ][gc,heap,exit] def new generation total 9216K, used 1311K
[0x00000007ffec0000, 0x00000007ff600000, 0x00000007ff600000]
[0.184s][info ][gc,heap,exit] eden space 8192K, 16% used [0x00000007fed4e38, 0x00000007fed4f0000]
[0.184s][info ][gc,heap,exit] from space 1024K, 0% used [0x00000007ff400000, 0x00000007ff400000, 0x00000007ff500000]
[0.185s][info ][gc,heap,exit] to space 1024K, 0% used [0x00000007ff500000, 0x00000007ff600000, 0x00000007ff600000]
[0.185s][info ][gc,heap,exit] tenured generation total 10240K, used 0K
[0x00000007ff600000, 0x0000000800000000, 0x0000000800000000]
[0.185s][info ][gc,heap,exit] the space 10240K, 0% used [0x00000007ff600000, 0x00000007ff600000, 0x00000007ff600000]
[0.185s][info ][gc,heap,exit] Metaspace
used 309K, committed 384K, reserved 1056768K
[0.185s][info ][gc,heap,exit] class space
used 15K, committed 64K, reserved 1048576K
```

From the log we can see that the young generation size is 9216K, which is the sum of Eden space 8192K and from space 1024 K. Also the base consumption of the Eden area is 1311K.

B. Trigger process of Minor GC

In most cases, objects are allocated memory in the young generation Eden space. A Minor CG will occur when there is not enough space in the Eden space. The JVM provides the -XX:+PrintGCDetails option to print garbage collection logs.

To test how the garbage collector works, first create four byte array objects and their sizes are 2MB, 2MB, 3MB, 2MB:

```java
byte[] allocation1 = new byte[2 * 1024 * 1024];
byte[] allocation2 = new byte[2 * 1024 * 1024];
byte[] allocation3 = new byte[3 * 1024 * 1024];
byte[] allocation4 = new byte[2 * 1024 * 1024];
```

The execution parameters are set to: java -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 GCTest.

The test results are printed as follows:
We can see from the log that the size of the Eden space is 8192K, and the memory size of the from and to spaces is both 1024K. At the same time, it can be seen that the reason for this Minor GC is that when the object “allocation3” allocates 1024K. At the same time, it can be seen that the reason for this Minor GC is that when the object “allocation3” allocates 1024K. At the same time, it can be seen that the reason for this

VIII. CONCLUSION

The development of the Java garbage collector is accompanied by the development of computers and the development of user needs. The era of single-core machines gave rise to “Serial”, and later with multi-core machines, there was "ParNew". With the further improvement of usage requirements, people want both multi-threading and high throughput, and there was Parallel Scavenge. Then came G1, which takes into account both the young and old generations, can control the pause time caused by garbage collection without sacrificing throughput, and also solves the problem of memory fragmentation.

REFERENCES


