



# 6 Garbage Collection in JVM

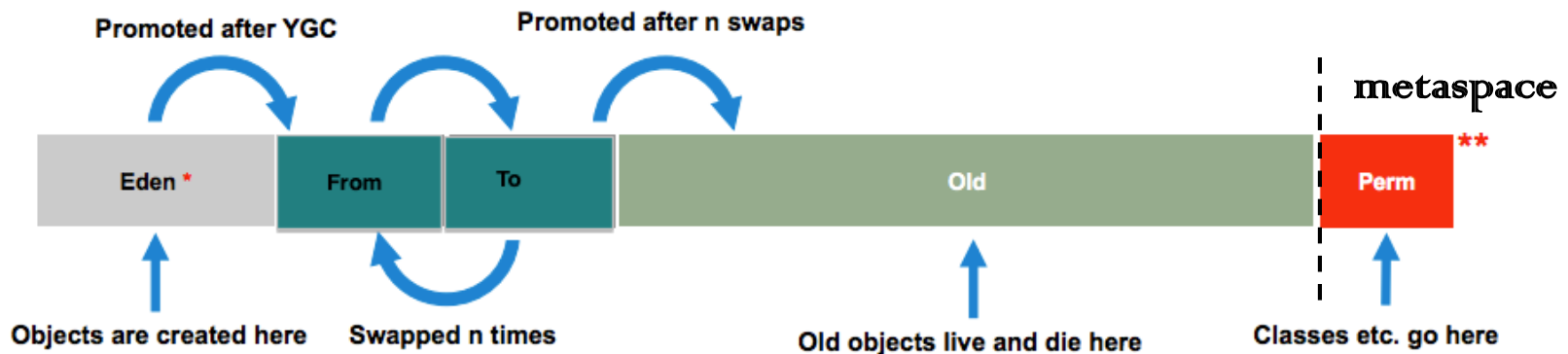


# Java Garbage Collector

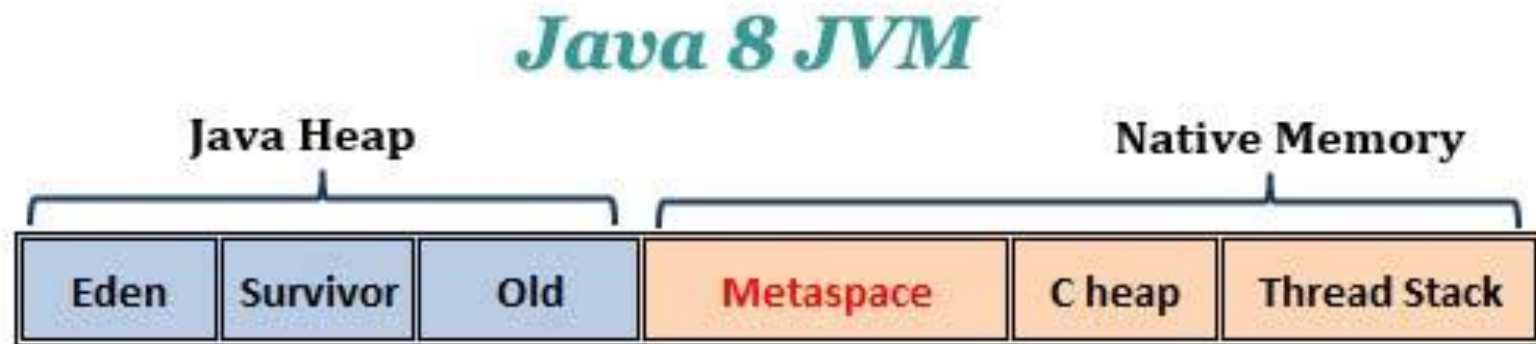
- The Java garbage collector can logically separate the heap into different areas, so that the GC can faster identify objects which can get removed. **Java垃圾回收将堆划分成不同的区域(generation代), 以便GC可以更快地识别可以删除的对象**
- The JVM automatically re-collects the memory which is not used any more.
  - The memory for objects which are not referred any more will be automatically released by the garbage collector.
  - To see that the garbage collector starts working add the command line argument "**-verbose:gc**" to your virtual machine.

# Garbage Collection in JVM

- The HotSpot VM (Sun JVM) has three major spaces: **young generation, old generation, and permanent/metaspase generation.**
  - When a Java application allocates Java objects, those objects are allocated in the young generation space. 新对象分配到young generation中
  - Objects that survive, that is, those that remain live, after some number of minor garbage collections are promoted into the old generation space. GC后仍然存活的对象，提升到old generation中
  - The permanent generation space holds VM and Java class metadata as well as interned Strings and class static variables PermGen/Metaspase中保存VM和class的元数据，以及类的静态变量



# Garbage Collection in JVM



# Generational Collection GC algorithms

- **Generational Collection GC algorithms**

- The various GC algorithms discussed above have their own advantages and disadvantages. It's wise to use appropriate algorithm according to the characteristics of GC objects.

- **Generational Collection divides memory space into several pieces (young generation, old (tenured) generation, and permanent generation) /metaspace and use different GC algorithms according to the characteristics of each memory pieces, in order to improve the efficiency of garbage collection. 根据不同代的不同特征，采用不同的GC算法**

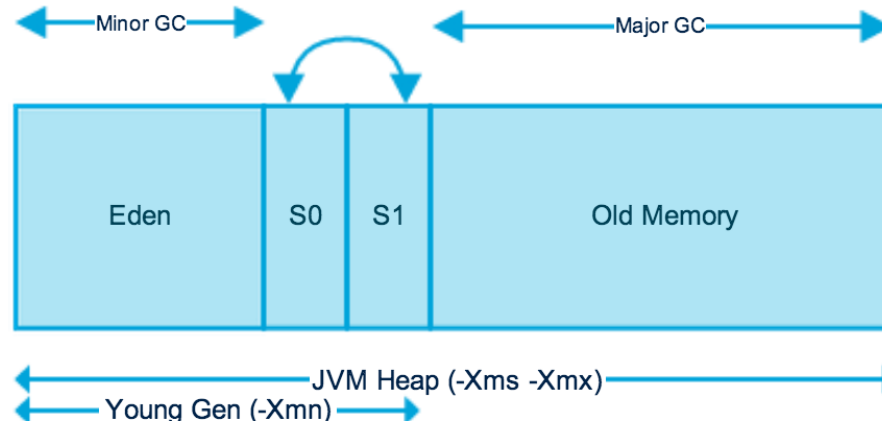
# The Yong and Old Generations

## ■ In the young generation

- Each time garbage collection will find a large number of objects die, and only a small amount survive. 每次GC会发现大量死亡对象，少量存活对象
- So the **copying algorithm is appropriate** which only needs to pay the cost of copying a small amount of living objects. 复制算法适合，代价低

## ■ In the old generation

- Because objects have a high survival rate and no additional memory space to be allocated, the **Mark-Sweep or Mark-Compact** algorithm must be used for collection. 对象存活率高，适合采用标记算法



# The PermGen and Metaspace

- PermGen (Permanent Generation) is a special heap space separated from the main memory heap.
- The JVM keeps track of loaded **class metadata** in the PermGen. Additionally, the JVM stores all the static content in this memory section. This includes **all the static methods, primitive variables, and references to the static objects**. Furthermore, it contains data about **bytecode, names and JIT information**. Before Java 7, **the String Pool was also part of this memory**. **PermGen中保存类的定义、静态方法、静态对象的引用等内容**
- With its limited memory size, PermGen is involved in generating the famous *OutOfMemoryError*. Simply put, the class loaders aren't garbage collected properly and, as a result, generated a memory leak. Therefore, we receive a memory space error; this happens mostly on development environment while creating new class loaders. **缺点：PermGen的容量是固定的(缺省或指定)，固定的容量容易导致运行时的内存溢出错误**

# The PermGen and Metaspace

- Simply put, Metaspace is a new memory space – starting from the Java 8 version; it has replaced the older PermGen memory space. The most significant difference is how it handles the memory allocation. **Java 8 开始，用Metaspace替代了PermGen，区别在内存分配方面**
- As a result, this native memory region grows automatically by default. Additionally, the garbage collection process also gains some benefits from this change. The garbage collector now automatically triggers cleaning of the dead classes once the class metadata usage reaches its maximum metaspace size. **Metaspace的容量是自动增长的，此外，当类的元数据使用达到了metaspace最大值时，会自动触发GC**
- Therefore, with this improvement, JVM reduces the chance to get the *OutOfMemory* error. **降低了OutOfMemory 的风险**
- Despite all of this improvements, we still need to monitor and tune up the metaspace to avoid memory leaks. **但仍然需要监控和调优，避免内存泄漏**



# When Garbage Collection occurs

- It is important to understand that a garbage collection occurs when any one of the three spaces, young generation, old generation, or permanent/meatspace generation, is in a state where it can no longer satisfy an allocation event.
- In other words, a garbage collection occurs when any one of those three spaces is considered full and there is some request for additional space that is not available. 三个空间任何一个已满，且存在对空间的额外需求时，会发生GC

# Minor, Major and Full Garbage Collection

- **Minor Collection for young generation**
- **Major Collection for old generation**
- **Full Collection for young、old and permanent generation**
- When the young generation space does not have enough room available to satisfy a Java object allocation, the HotSpot VM performs a minor garbage collection to free up space. Minor garbage collections tend to be short in duration relative to full garbage collections. 年轻代中内存不够时，发生**Minor GC**
- Objects that remain live for some number of minor garbage collections eventually get promoted (copied) to the old generation space. **Minor GC后存活的对象升级到老年代**
- When the old generation space no longer has available space for promoted objects, the HotSpot VM performs a full garbage collection. 老年代空间不够时，进行**full GC**

# Minor, Major and Full Garbage Collection

- It actually performs a full garbage collection when it determines there is not enough available space for object promotions from the next minor garbage collection. This is a less costly approach rather than being in the middle of a minor garbage collection and discovering that the promotion of an object will fail.
- Recovering from an object promotion failure is an expensive operation. A full garbage collection also occurs when the permanent generation space does not have enough available space to store additional VM or class metadata. **full GC**针对年轻代、老年代和永久代，当没有空间提供给**minor GC**将对象提升到老年代中，或者永久代中无空间保存**class**元数据时发生。



# 7 Garbage Collection Tuning in JVM



# Performance Considerations in GC

- Performance Considerations

- **Throughput** is the percentage of total time not spent in garbage collection considered over long periods of time. Throughput includes time spent in allocation (but tuning for speed of allocation is generally not needed). 吞吐量:总时间中非用于垃圾回收的时间
  - *Pauses* are the times when an application appears unresponsive because garbage collection is occurring. 暂停: 由于GC导致的暂停次数
- Users have different requirements of garbage collection. For example, some consider the right metric for a web server to be throughput because pauses during garbage collection may be tolerable or simply obscured by network latencies. However, in an interactive graphics program, even short pauses may negatively affect the user experience. 不同用户有不同的GC需求, E.g., Web用户追求高吞吐量, 而图形交互程序则追求低暂停。

# Performance Considerations in GC

- Some users are sensitive to other considerations.
  - *Footprint* is the working set of a process, measured in pages and cache lines. On systems with limited physical memory or many processes, footprint may dictate *scalability*(可伸缩性).
  - *Promptness*(及时性) is the time between when an object becomes dead and when the memory becomes available, an important consideration for distributed systems, including Remote Method Invocation (RMI).

# Performance Considerations in GC

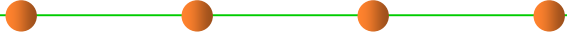
- **In general, choosing the size for a particular generation is a trade-off between these considerations.** For example, a very large young generation may maximize throughput, but does so at the expense of footprint, promptness, and pause times. Young generation pauses can be minimized by using a small young generation at the expense of throughput. **The sizing of one generation does not affect the collection frequency and pause times for another generation.** 需要根据各种情况权衡代的容量，一个代的容量不影响其他代的回收频率和暂停时间。
- **There is no one right way to choose the size of a generation. The best choice is determined by the way the application uses memory as well as user requirements.** 没有普适的容量设置准则，应根据应用的内存使用 and 用户需求进行具体问题具体分析。

# Garbage Collection Tuning in JVM

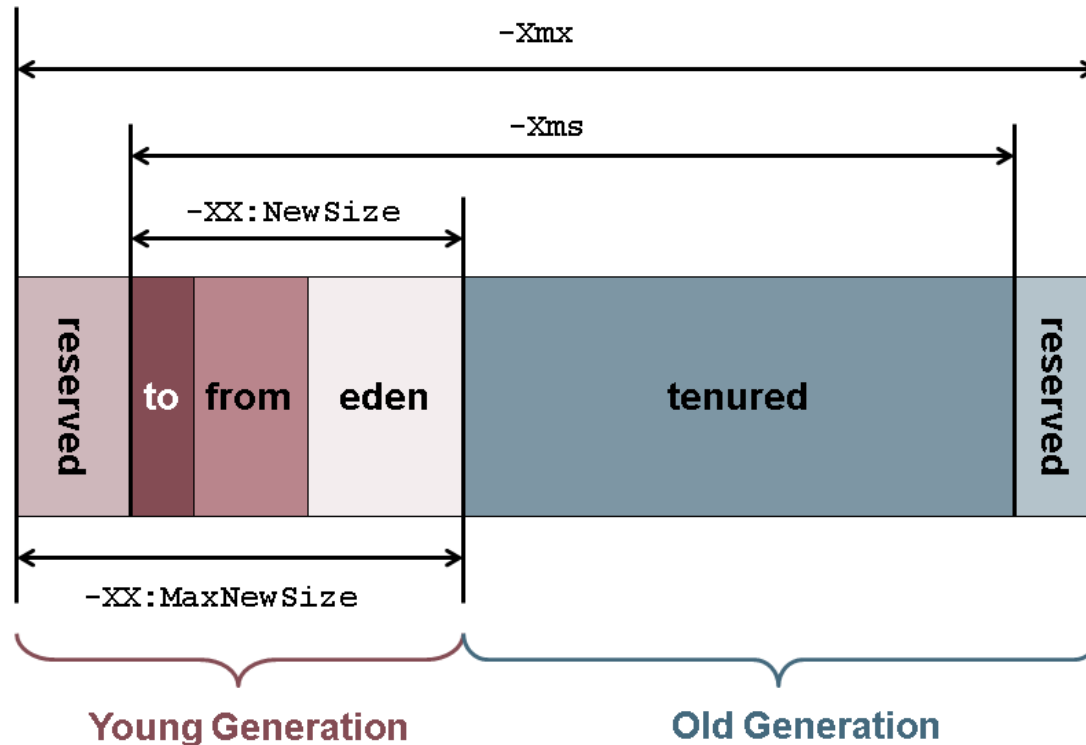
- A best practice is to tune the time spent doing garbage collection to within 5% of execution time. 最佳做法是将GC时间控制在执行时间的5%之内
- **The JVM runs with fixed available memory.** Once this memory is exceeded you will receive "java.lang.OutOfMemoryError".  
"Exception in thread java.lang.OutOfMemoryError:  
Java heap space".
- The JVM tries to make an intelligent choice about the available memory at startup but you can overwrite the default with the following settings. JVM在启动时提供了缺省的设置，可根据具体需求自行重新设定。



# Tuning JVM's garbage collection

- 
- Specifying VM heap size
  - Choosing a garbage collection scheme
  - Using *verbose* garbage collection to determine heap size
  - Automatically logging low memory conditions
  - Manually requesting garbage collection
  - Requesting thread stacks

# (1) Tuning VM Heap Size



# (1) Tuning VM Heap Size

- **The Java heap is where the objects of a Java program live. It is a repository for live objects, dead objects, and free memory.**
  - When an object can no longer be reached from any pointer in the running program, it is considered "garbage" and ready for collection.
- **The JVM heap size determines how often and how long the VM spends collecting garbage. JVM堆大小决定了虚拟机收集垃圾的频率和时间长短。**
  - An acceptable rate for garbage collection is application-specific and should be adjusted after analyzing the actual time and frequency of garbage collections. 对于特定的程序，应分析实际情况后进行调整
  - If you set a large heap size, full garbage collection is slower, but it occurs less frequently. 较大的堆，GC速度慢，GC频率低
  - If you set your heap size in accordance with your memory needs, full garbage collection is faster, but occurs more frequently. 如果根据内存需求设置堆大小，则完整垃圾回收速度会更快，但会更频繁地发生。

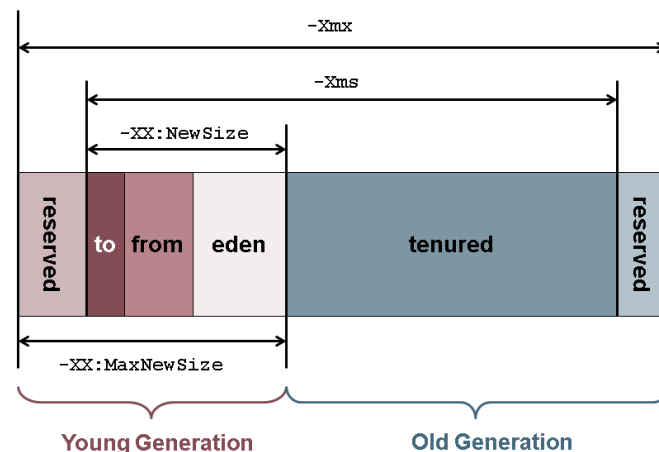
# (1) Tuning VM Heap Size

- **The -Xmx and -Xms command line options specify the initial and maximum total size of the young generation and old generation spaces. This initial and maximum size is also referred to as the Java heap size.**
  - Java -Xms 1024M      年轻代和老年代之和的初始值
  - Java -Xmx 2048M      年轻代和老年代之和的最大值
- **When -Xms is smaller than -Xmx, the amount of space consumed by young and old generation spaces is allowed to grow or contract depending on the needs of the application.**
  - The growth of the Java heap will never be larger than -Xmx, and the Java heap will never contract smaller than -Xms.
  - Growing or contracting the size of either the young generation space or old generation space requires a full garbage collection.
  - Full garbage collections can reduce throughput and induce larger than desired latencies.

# (1) Tuning VM Heap Size

## ■ The young generation space is specified using any one of the following command line options:

- `-XX: NewSize=<n>[g|m|k]` The initial and minimum size of the young generation space. <n> is the size. [g|m|k] indicates whether the size should be interpreted as gigabytes, megabytes, or kilobytes. 最小值
- `-XX: MaxNewSize=<n>[g|m|k]` The maximum size of the young generation space. 最大值
- `-Xmn<n>[g|m|k]` Sets the initial, minimum, and maximum size of the young generation space to the same value. 初始、最小、最大为同一值



# (1) Tuning VM Heap Size

- The size of the old generation space is implicitly set based on the size of the young generation space. 老年代大小同年轻代大小相关
  - The initial old generation space size is the value of `-Xms` minus `-XX:NewSize`.
  - The maximum old generation space size is the value of `-Xmx` minus `-XX:MaxNewSize`.
  - If `-Xms` and `-Xmx` are set to the same value and `-Xmn` is used, or `-XX:NewSize` is the same value as `-XX:MaxNewSize`, then the old generation size is `-Xmx` (or `-Xms`) minus `-Xmn`.

If the garbage collector has become a bottleneck, you may wish to customize the generation sizes. Check the *verbose* garbage collector output, and then explore the sensitivity of your individual performance metric to the garbage collector parameters.

# (1) Tuning VM Heap Size

## ■ The size of the Metaspace:

- `-XX:MaxMetaspaceSize`, 最大空间, 默认没有限制。
- `-XX:MetaspaceSize`, 初始空间大小, 达到该值就会触发垃圾收集。同时GC会对该值进行调整: 如果释放了大量的空间, 就适当降低该值; 如果释放了很少的空间, 那么在不超过`MaxMetaspaceSize`时, 适当提高该值。
- `-XX:MinMetaspaceFreeRatio` is the minimum percentage of class metadata capacity free after garbage collection 在GC之后, 最小的Metaspace剩余空间容量的百分比。
- `-XX:MaxMetaspaceFreeRatio` is the maximum percentage of class metadata capacity free after a garbage collection to avoid a reduction in the amount of space 在GC之后, 最大的Metaspace剩余空间容量的百分比。

# (1) Tuning VM Heap Size

- **-XX:MinHeapFreeRatio=<n>**

- 设置堆内存的最小空闲比例，在使用率小于 n 的情况下，heap 进行收缩，Xmx==Xms 的情况下无效

- **-XX:MaxHeapFreeRatio=<n>**

- 设置堆内存的最大空闲比例，在使用率大于 n 的情况下，heap 进行扩张，Xmx==Xms 的情况下无效

- **-XX:NewRatio=<n>**

- 指定Old Generation heap size 与 Young Generation 的比例

- **-XX:SurvivorRatio=<n>**

- 指定 Young Generation 中 Eden Space 与一个 Survivor Space 的 heap size 比例



## (2) Choosing a Garbage Collection Scheme

- Depending on which JVM you are using, you can choose from several garbage collection schemes to manage your system memory.
- Some garbage collection schemes are more appropriate for a given type of application.
- Once you have an understanding of the workload of the application and the different garbage collection algorithms utilized by the JVM, you can optimize the configuration of the garbage collection. 一旦理解了应用程序的工作负载以及JVM使用的不同垃圾收集算法，就可以优化垃圾收集的配置。取决于：应用的需求和JVM的版本。

## (2) Choosing a Garbage Collection Scheme

---

- JVM has four types of GC implementations:
  - Serial Garbage Collector
  - Parallel Garbage Collector
  - CMS Garbage Collector
  - G1 Garbage Collector

## (2) Choosing a Garbage Collection Scheme

- **The serial garbage collector is** the simplest GC implementation, as it basically works with a single thread. As a result, this GC implementation freezes all application threads when it runs. Hence, it is not a good idea to use it in multi-threaded applications like server environments. 串行收集器，使用一个线程进行垃圾回收，执行时会冻结所有的应用线程，不适合多线程应用。
- The Serial GC is the garbage collector of choice for most applications that do not have small pause time requirements and run on client-style machines. 适合不要求低暂停时间和单机程序。
- **-XX:+UseSerialGC**

## (2) Choosing a Garbage Collection Scheme

- **The throughput (parallel) collector** performs minor collections (on the young generation) in parallel, which can significantly reduce garbage collection overhead. (but major collections are performed using a single thread) 对年轻代的回收采用并行方式(多个线程)，对老年代的回收还是单线程
- **-XX:+UseParallelGC**
- It's the default GC of the JVM and sometimes called Throughput Collectors. But it also freezes other application threads while performing GC.

## (2) Choosing a Garbage Collection Scheme

- The *Concurrent Mark Sweep* (CMS) implementation uses multiple garbage collector threads for garbage collection. It's designed for applications that prefer shorter garbage collection pauses, and that can afford to share processor resources with the garbage collector while the application is running. 利用多垃圾回收线程，适用于短回收暂停，且能够在应用程序运行时与垃圾收集器共享处理器资源。
- Simply put, applications using this type of GC respond slower on average but do not stop responding to perform garbage collection. 应用程序平均响应较慢，但不会停止响应以执行垃圾收集。
- **-XX:+UseParNewGC**

## (2) Choosing a Garbage Collection Scheme

- *G1 (Garbage First) Garbage Collector* is designed for applications running on multi-processor machines with large memory space. It's available since *JDK7 Update 4* and in later releases. 适用于运行在多处理器大内存空间的应用程序。
- Unlike other collectors, *G1* collector partitions the heap into a set of equal-sized heap regions, each a contiguous range of virtual memory. When performing garbage collections, *G1* shows a concurrent global marking phase (i.e. phase 1 known as *Marking*) to determine the liveness of objects throughout the heap.
- After the mark phase is completed, *G1* knows which regions are mostly empty. It collects in these areas first, which usually yields a significant amount of free space (i.e. phase 2 known as *Sweeping*). It is why this method of garbage collection is called Garbage-First.
- **-XX:+UseG1GC**


### (3) Using verbose garbage collection

- The verbose garbage collection option (verbosegc) enables you to **measure exactly how much time and resources are put into garbage collection.**

-verbose:gc

- To determine the most effective heap size, turn on verbose garbage collection and redirect the output to a log file for diagnostic purposes. -Xloggc:日志文件路径
- From log file:
  - How often is garbage collection taking place?
  - How long is garbage collection taking? Full garbage collection should not take longer than 3 to 5 seconds.
  - What is your average memory footprint? In other words, what does the heap settle back down to after each full garbage collection? If the heap always settles to 85 percent free, you might set the heap size smaller.

## (4) Manually request garbage collection

- 
- You can manually request that the JVM perform garbage collection.
  - When you perform garbage collection, the JVM often examines every living object in the heap.
  - Garbage Collect calls the JVM's `System.gc()` method to perform garbage collection. The JVM implementation then decides whether or not the request actually triggers garbage collection.