

Summary of the most important JVM parameters

👤 [Guide](#) 📄 Java 💎 JVM 🕒 About 3272 words ⌚ About 11 minutes

This article is translated by JavaGuide from <https://www.baeldung.com/jvm-parameters> and has been extensively supplemented.

Documentation parameters :

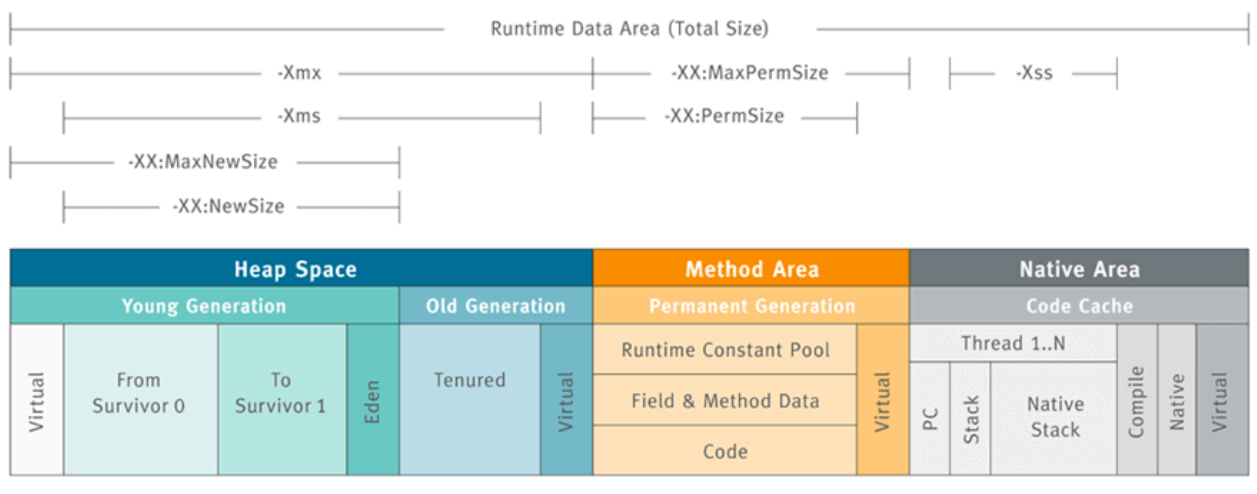
<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html>

JDK version: Mainly 1.8, common parameters of new versions will also be added

In this article, we will learn about some of the most commonly used parameter configurations in the Java Virtual Machine (JVM) to help you better understand and tune the operating environment of Java applications.

Heap memory related

The Java heap is the largest area of memory managed by the JVM, **shared by all threads** and created when the virtual machine starts. **This memory area's sole purpose is to store object instances. Almost all object instances and arrays require memory allocated on the heap.**



Setting the heap memory size (-Xms and -Xmx)

Setting the initial and maximum heap memory sizes based on the actual needs of the application is one of the most common practices in performance tuning. **It is recommended to explicitly set these two parameters, and it is generally recommended to set them to the same value** to avoid the performance overhead caused by dynamic adjustment of the heap memory at runtime.

Use the following parameters to set it up:

```
1 -Xms<heap size>[unit] # 设置 JVM 初始堆大小
2 -Xmx<heap size>[unit] # 设置 JVM 最大堆大小
```

bash

- <heap size> : Specifies the specific value of memory.
- [unit] : Specifies the unit of memory, such as g (GB), m (MB), k (KB).

Example: Set the JVM's initial and maximum heap sizes to 4GB:

```
1 -Xms4G -Xmx4G
```

bash

Set the Young Generation Memory Size

According to [Oracle's official documentation](#) , after configuring the total available heap memory, the second most influential factor is Young Generation the proportion of heap memory used. By default, the minimum size of YG is **1310 MB** and the maximum size is **unlimited** .

The new generation memory size can be set in two ways:

1. Adoption -XX:NewSize and -XX:MaxNewSize designation

```
1 -XX:NewSize=<young size>[unit] # 设置新生代初始大小
2 -XX:MaxNewSize=<young size>[unit] # 设置新生代最大大小
```

bash

Example: Set the minimum size of the new generation to 512MB and the maximum size to 1024MB:

```
1 -XX:NewSize=512m -XX:MaxNewSize=1024m
```

bash

2. By `-Xmn<young size>[unit]` specifying

Example: To fix the new generation size to 512MB:

```
1 -Xmn512m bash
```

An important rule of thumb in GC tuning strategies is as follows:

Try to allocate and reclaim memory in the Young Generation for newly created objects, as Minor GC is typically much less expensive than Full GC. Analyze GC logs to determine whether Young Generation space allocation is appropriate. If a large number of new objects are prematurely promoted to the Old Generation (promotion), adjust the Young Generation size appropriately using [`-Xmn`] or [`-XX:NewSize/-XX:MaxNewSize`] to minimize the number of objects that enter the Old Generation directly.

In addition, you can also use `-XX:NewRatio=<int>` the parameter to set **the memory size ratio of the old generation and the new generation (excluding the Survivor area)**.

For example, `-XX:NewRatio=2` (default value) means the ratio of old generation to new generation is 2:1. That is, the new generation occupies 1/3 of the total heap size.

```
1 -XX:NewRatio=2 bash
```

Set the PermGen/Metaspace size

Starting from Java 8, if we do not specify the size of Metaspace, as more classes are created, the VM will exhaust all available system memory (this does not happen with the PermGen).

Before JDK 1.8, when the permanent generation had not yet been completely removed, the following parameters were usually used to adjust the method area size:

```
1 -XX:PermSize=N #方法区（永久代）初始大小
2 -XX:MaxPermSize=N #方法区（永久代）最大大小,超过这个值将会抛出
  OutOfMemoryError 异常:java.lang.OutOfMemoryError: PermGen
```

Relatively speaking, garbage collection rarely occurs in this area, but it does not mean the data will "exist forever" after entering the method area.



In JDK 1.8, the method area (HotSpot's permanent generation) was completely removed (this started in JDK 1.7) and replaced by the metaspace, which uses local memory.

Here are some common parameters:

```
1 -XX:MetaspaceSize=N #设置 Metaspace 的初始大小（是一个常见的误区，后面会解  
2 释）  
-XX:MaxMetaspaceSize=N #设置 Metaspace 的最大大小
```

🐛 **Fix (see: [issue#1947](#)):**

1. -XX:MetaspaceSize Not the initial capacity: The initial capacity of the Metaspace is not -XX:MetaspaceSize a setting. Regardless of -XX:MetaspaceSize the configured value, for a 64-bit JVM, the initial capacity of the Metaspace is usually a fixed small value (Oracle documentation mentions about 12MB to 20MB, and the actual observation is about 20.8MB).

You can refer to the Oracle official document [Other Considerations](#) mentioned:

Specify a higher value for the option MetaspaceSize to avoid early garbage collections induced for class metadata. The amount of class metadata allocated for an application is application-dependent and general guidelines do not exist for the selection of MetaspaceSize. The default size of MetaspaceSize is platform-dependent and ranges from 12 MB to about 20 MB.

The default size of MetaspaceSize depends on the platform and ranges from 12 MB to approximately 20 MB.

Also, take a look at this experiment: [Misunderstanding of the JVM parameter MetaspaceSize](#) .

2. Expansion and Full GC: When Metaspace usage increases and reaches -XX:MetaspaceSize a specified threshold for the first time, a Full GC is triggered. Thereafter, the JVM dynamically adjusts the threshold for triggering GC. If Metaspace continues to grow, a Full GC may still be triggered each time a new threshold is reached and expansion is required (the specific behavior depends on the garbage collector and version). The garbage collector internally `_capacity_until_GC` determines whether the Metaspace area has reached the threshold based on variables. The initialization code is as follows:



```

1  void MetaspaceGC::initialize() {
2      // Set the high-water mark to MaxMetaspaceSize during VM
3      initialization since
4      // we can't do a GC during initialization.
5      _capacity_until_GC = MaxMetaspaceSize;
    }

```

3. -XX:MaxMetaspaceSize Importance: If you don't explicitly set -XX:MaxMetaspaceSize , the maximum size of the Metaspace is theoretically limited by available native memory. In extreme cases (such as a class loader leak causing constant class loading), this can indeed exhaust a significant amount of native memory . Therefore, setting a reasonable -XX:MaxMetaspaceSize upper limit is strongly recommended to prevent system impact.


Related reading: [Issue Correction: MaxMetaspaceSize will not exhaust memory if size is not specified #1204](#) .

Garbage collection related

Selecting a garbage collector

Choosing the right garbage collector (GC) is crucial for your application's throughput and latency. For a detailed introduction to garbage collection algorithms and collectors, see my article: [JVM Garbage Collection Explained \(Key Points\)](#) .

JVM provides multiple GC implementations suitable for different scenarios:

- **Serial GC (Serial Garbage Collector):** Single-threaded GC, suitable for client mode or single-core CPU environment. Parameters: `-XX:+UseSerialGC` .
- **Parallel GC (Parallel Garbage Collector):** Multi-threaded execution of Minor GC for the young generation and, optionally, Full GC for the old generation - `XX:+UseParallelOldGC` . Focused on throughput, it is the default GC for JDK 8. Parameters: `-XX:+UseParallelGC` .
- **CMS GC (Concurrent Mark Sweep):** Targets minimal collection pause times, allowing most GC phases to execute concurrently with user threads. Suitable for applications with high response time requirements. Deprecated in JDK 9 and removed in JDK 14. Parameters: `-XX:+UseConcMarkSweepGC` .
- **G1 GC (Garbage-First Garbage Collector):** The default GC for JDK 9 and later.  It divides the heap into multiple regions, balancing throughput and pause times, and attempts to complete GC within predictable pause times. Parameters: `-XX:+UseG1GC` .

- **ZGC:** A newer, low-latency GC that aims to reduce GC pause times to milliseconds or even sub-milliseconds. It requires support from a newer JDK version. Parameters (specific parameters may vary with version): `-XX:+UseZGC` , `-XX:+UseShenandoahGC` .

GC logging

In a production environment or when troubleshooting GC issues, **be sure to enable GC logging** . Detailed GC logs are key to analyzing and resolving GC issues.

The following are some recommended GC log parameters (applicable to common versions such as JDK 8/11):

```
1  # --- 推荐的基础配置 ---
2  # 打印详细 GC 信息
3  -XX:+PrintGCDetails
4  # 打印 GC 发生的时间戳（相对于 JVM 启动时间）
5  # -XX:+PrintGCTimeStamps
6  # 打印 GC 发生的日期和时间（更常用）
7  -XX:+PrintGCDateStamps
8  # 指定 GC 日志文件的输出路径，%t 可以输出日期时间戳
9  -Xloggc:/path/to/gc-%t.log
10
11 # --- 推荐的进阶配置 ---
12 # 打印对象年龄分布（有助于判断对象晋升老年代的情况）
13 -XX:+PrintTenuringDistribution
14 # 在 GC 前后打印堆信息
15 -XX:+PrintHeapAtGC
16 # 打印各种类型引用（强/软/弱/虚）的处理信息
17 -XX:+PrintReferenceGC
18 # 打印应用暂停时间（Stop-The-World, STW）
19 -XX:+PrintGCApplicationStoppedTime
20
21 # --- GC 日志文件滚动配置 ---
22 # 启用 GC 日志文件滚动
23 -XX:+UseGCLogFileRotation
24 # 设置滚动日志文件的数量（例如，保留最近 14 个）
25 -XX:NumberOfGCLogFiles=14
26 # 设置每个日志文件的最大大小（例如，50MB）
27 -XX:GCLogFileSize=50M
28
29
30
```

bash



```
31
32 # --- 可选的辅助诊断配置 ---
# 打印安全点（Safepoint）统计信息（有助于分析 STW 原因）
# -XX:+PrintSafepointStatistics
# -XX:PrintSafepointStatisticsCount=1
```

Note: JDK 9 and later versions introduced a unified JVM logging framework (`-Xlog`), which has different configuration methods. However, the above `-Xloggc` and rolling parameters are generally still compatible or have corresponding new parameters.

Dealing with OOM

It is very common for large applications to face out of memory errors which in turn leads to application crash. This is a very critical scenario and it is difficult to reproduce and resolve the issue.

That's why the JVM provides some parameters that dump the heap memory into a physical file that can be used later to find leaks:

```
1 # 在发生 OOM 时生成堆转储文件
2 -XX:+HeapDumpOnOutOfMemoryError
3
4 # 指定堆转储文件的输出路径。<pid> 会被替换为进程 ID
5 -XX:HeapDumpPath=/path/to/heapdump/java_pid<pid>.hprof
6 # 示例: -XX:HeapDumpPath=/data/dumps/
7
8 # (可选) 在发生 OOM 时执行指定的命令或脚本
9 # 例如, 发送告警通知或尝试重启服务 (需谨慎使用)
10 # -XX:OnOutOfMemoryError="<command> <args>"
11 # 示例: -XX:OnOutOfMemoryError="sh /path/to/notify.sh"
12
13 # (可选) 启用 GC 开销限制检查
14 # 如果 GC 时间占总时间比例过高 (默认 98%) 且回收效果甚微 (默认小于 2% 堆内存),
15 # 会提前抛出 OOM, 防止应用长时间卡死在 GC 中。
16 -XX:+UseGCOverheadLimit
```

bash

Other common parameters

- `-server` : Explicitly enable the HotSpot VM in Server mode. (Usually the default on 64-bit JVMs).
- `-XX:+UseStringDeduplication` (JDK 8u20+) Attempts to identify and share `char[]` String objects with the same underlying array to reduce memory usage. This is useful for scenarios with a large number of repeated strings.
- `-XX:SurvivorRatio=<ratio>` : Sets the size ratio of the Eden area to a single Survivor area. For example `-XX:SurvivorRatio=8` , Eden:Survivor = 8:1.
- `-XX:MaxTenuringThreshold=<threshold>` : Sets the maximum age threshold for objects to be promoted from the young generation to the old generation (the age of an object increases by 1 each time it survives a Minor GC). The default value is usually 15.
- `-XX:+DisableExplicitGC` : Disable explicit calls in code `System.gc()` . It is recommended to enable this feature to avoid triggering unnecessary Full GCs.
- `-XX:+UseLargePages` : (Requires OS support) Try using large memory pages (e.g. 2MB instead of 4KB). This may improve performance for memory-intensive applications, but requires careful testing.
- `-XX:MinHeapFreeRatio=<percent>` / `-XX:MaxHeapFreeRatio=<percent>` : Controls the minimum/maximum percentage of heap memory that remains free after GC, used for dynamic heap resizing (if `-Xms` and `-Xmx` are not equal). It is generally recommended to set `-Xms` and `-Xmx` to be the same to avoid resizing overhead.

Note: The following parameters may be deprecated, removed, or enabled by default in modern JVM versions and do not need to be set manually :

- `-XX:+UseLWPSynchronization` : An older synchronization strategy option, modern JVMs usually have more optimized implementations.
- `-XX:LargePageSizeInBytes` : Usually `-XX:+UseLargePages` determined automatically by or through OS configuration.
- `-XX:+UseStringCache` : Removed.
- `-XX:+UseCompressedStrings` : Superseded by the Compact Strings feature, which is enabled by default in Java 9 and later.
- `-XX:+OptimizeStringConcat` : String concatenation optimization (invokedynamic) is the default behavior in Java 9 and later.



Summarize

This article provides a practical guide to common JVM parameter configurations for Java developers, aiming to help readers understand and optimize the performance and stability of Java applications. It highlights the following aspects:

- 1. Heap memory configuration:** It is recommended to explicitly set the initial and maximum heap memory (`-Xms` , `-Xmx` , usually set to the same) and the new generation size (`-Xmn` or `-XX:NewSize/-XX:MaxNewSize`), which is crucial for GC performance.
- 2. Metaspace Management (Java 8+):** Clarified `-XX:MetaspaceSize` the actual role of (the threshold that triggers the first Full GC, not the initial capacity), and strongly recommends setting `-XX:MaxMetaspaceSize` to prevent potential native memory exhaustion.
- 3. **Garbage Collector Selection and Logging:**** This section describes the applicable scenarios of different GC algorithms and emphasizes the necessity of enabling detailed GC logging (`-Xloggc` , `-XX:+PrintGCDetails` etc.) for troubleshooting in production and test environments.
- 4. OOM Troubleshooting:** This section describes how to use `-XX:+HeapDumpOnOutOfMemoryError` parameters such as to automatically generate a heap dump file when an OOM occurs, so that you can perform subsequent memory leak analysis.
- 5. Other Parameters:** This section briefly introduces other useful parameters such as string deduplication, and points out the current status of some old parameters.

For specific troubleshooting and tuning cases, please refer to this article compiled by the author: [JVM online problem troubleshooting and performance tuning cases](https://javaguide.cn/java/jvm/jvm-parameters-intro.html#设置堆内存大小-xms-和-xmx) .



JavaGuide官方公众号

(微信搜索JavaGuide)



- 1、公众号后台回复“**PDF**”获取原创PDF面试手册
- 2、公众号后台回复“**学习路线**”获取Java学习路线最新版
- 3、公众号后台回复“**开源**”获取优质Java开源项目合集
- 4、公众号后台回复“**八股文**”获取Java面试真题+面经

Recently Updated 2025/4/25 06:41

Contributors: Snailclimb , guide , drlifeL , TommyMerlin , 1036956372 , Guide , Erzbir , Mr.Hope , paigeman , aucub , seven17777777 , hulingfeng , 1312255201

Copyright © 2025 Guide

