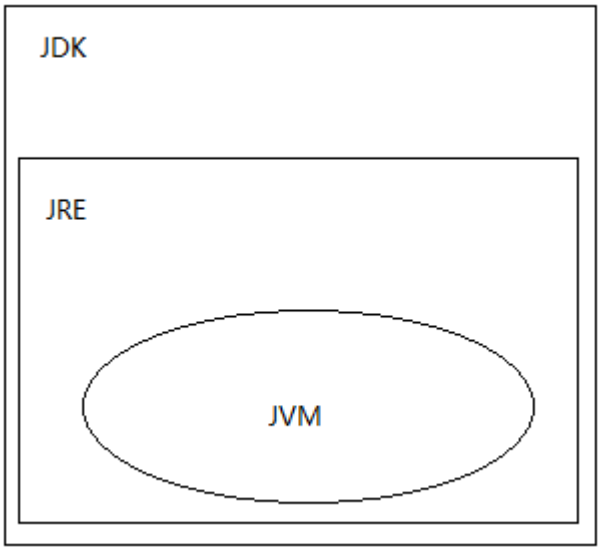
JVM

1. 引言

```
1). 什么是JVM

    JVM是Java Virtual Machine（Java虚拟机）的缩写。

    入Java语言虚拟机后，Java语言在不同平台上运行时不需要重新编译。Java语言使用Java虚拟机屏蔽了与具体平台相关的信息，使得Java语言编译程序只需生成在Java虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。

2). JDK , JRE , JVM 的关系

    JDK（Java Development Kit）是针对Java开发员的产品，是整个Java的核心，包括了运行环境JRE、Java工具和Java基础类库。
    Java Runtime Environment（JRE）是运行JAVA程序所必须的环境的集合，包含JVM标准实现及Java核心类库。
    JVM是Java Virtual Machine（Java虚拟机）的缩写，是整个java实现跨平台的最核心的部分，能够运行以Java语言写作的软件程序。


3). JVM有什么用

    一次编写，到处运行
    内存管理，垃圾回收
    数组越界，类型多态 ...

4). 有哪些JVM版本

    oracle - hotspot
    bea - jrocket
    ibm - j9
    google - harmony

```

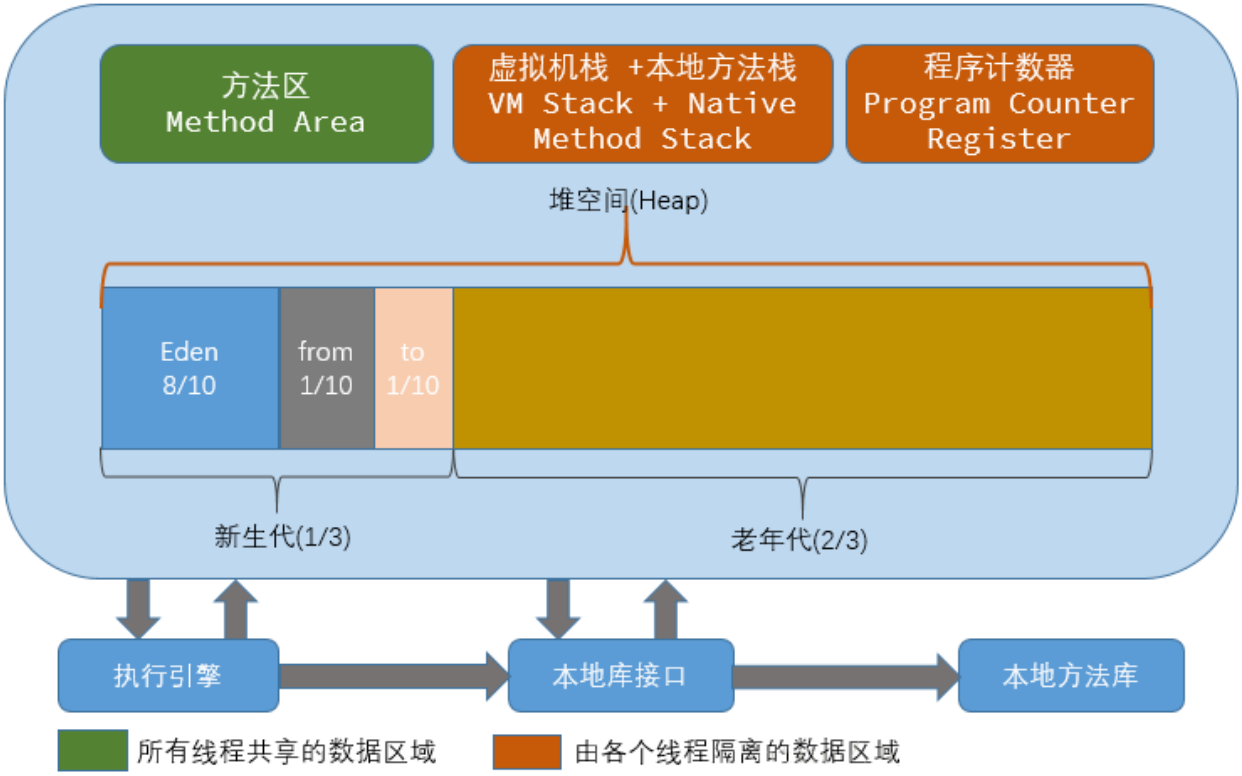


2. 内存结构

以JDK1.8 (hotspot) 运行时内存结构为例 ， 分为两大块 ： Java内存 和 非Java内存(堆外内存)：

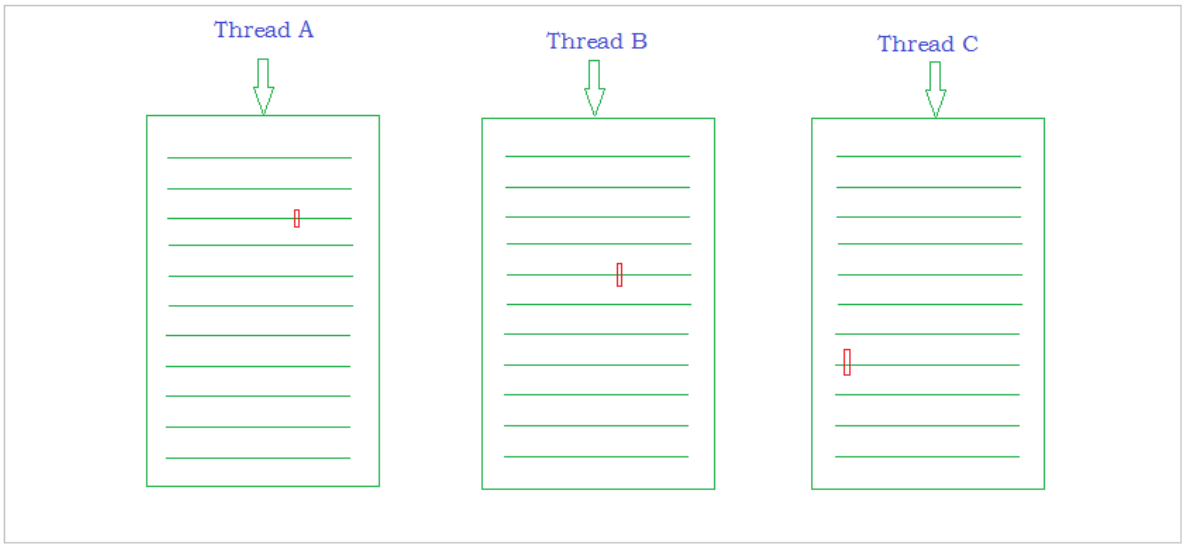
```
java 内存
    |- 线程私有
        |- 程序计数器
        |- 虚拟机栈
        |- 本地方法栈

    |- 线程共享
        |- 堆
        |- 方法区

非java内存（堆外内存）
    |- 直接内存
```

2.1 JAVA 内存



2.1.1 程序计数器

- 虚拟机需要通过『程序计数器』记录指令执行到哪了。
- 线程要轮流使用 CPU 时间片，因此需要『程序计数器』来记住正在执行的字节码的地址。例如 线程 A 的计数器记录当前执行到了第三行字节码，这时候时间片用完了，CPU 切换到其它线程运行，当 CPU 再次切换到 线程 A 时，它就会从计数器得知上次执行的代码位置，继续向下运行。



2.1.2 虚拟机栈

2.1.2.1 作用

一个线程使用的内存大小

线程内调用一次方法，就会产生一个栈帧，栈帧内包含方法内局部变量，方法参数，返回地址等。多个栈帧合称为『栈』，而正在执行的方法称为『活动栈帧』，一个线程内同一时刻只能有一个『活动栈帧』

2.1.2.2 配置

```
-Xss

The default value depends on the platform:
* Linux/x64 (64-bit): 1024 KB
* macOS (64-bit): 1024 KB
* Oracle Solaris/x64 (64-bit): 1024 KB
* Windows: The default value depends on virtual memory
```

2.1.2.3 特点

1). 方法执行完毕，栈帧内存即被释放

2). 因为线程私有，不存在共享，因此线程安全

3). 值越大，会让线程数更少

2.1.2.4 栈内存溢出情况

1). 栈太小，方法调用过深（栈帧太多）

2). 栈太小，方法内局部变量太多（栈帧太大）

2.1.2.5 案例

1). 使用 `debug` 演示栈帧，栈帧内变量

2). 演示栈内存大小设置，栈内存溢出（情况1）

3). 相关工具 `jps`， `jstack` 介绍

4). 演示使用 `jstack` 工具检查线程死锁

```java
public class Demo1 {

    private static  int count = 0;

    public static void main(String[] args) {
```

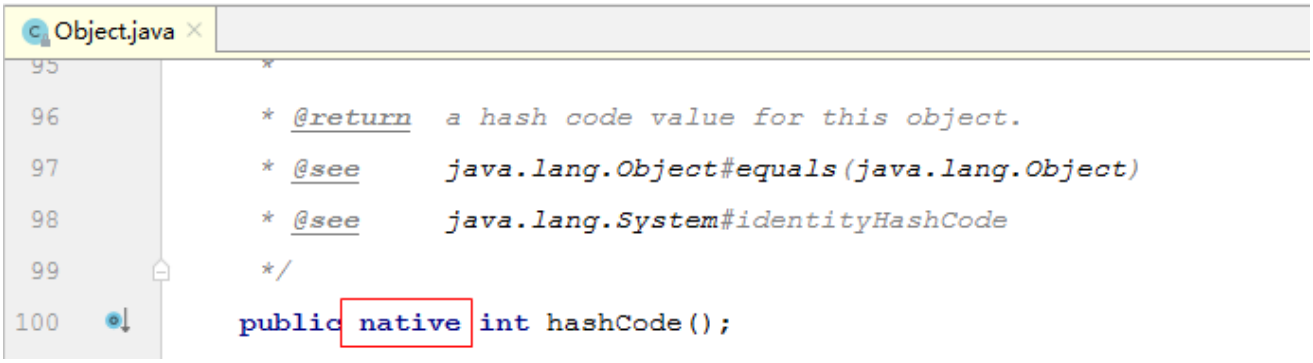
```
        method1();
    }

    private static void method1() {
        count ++ ;
        System.out.println(count);
        method1();
    }

}
```

配置 ：

```
-Xss2M
```

2.1.3 本地方法栈

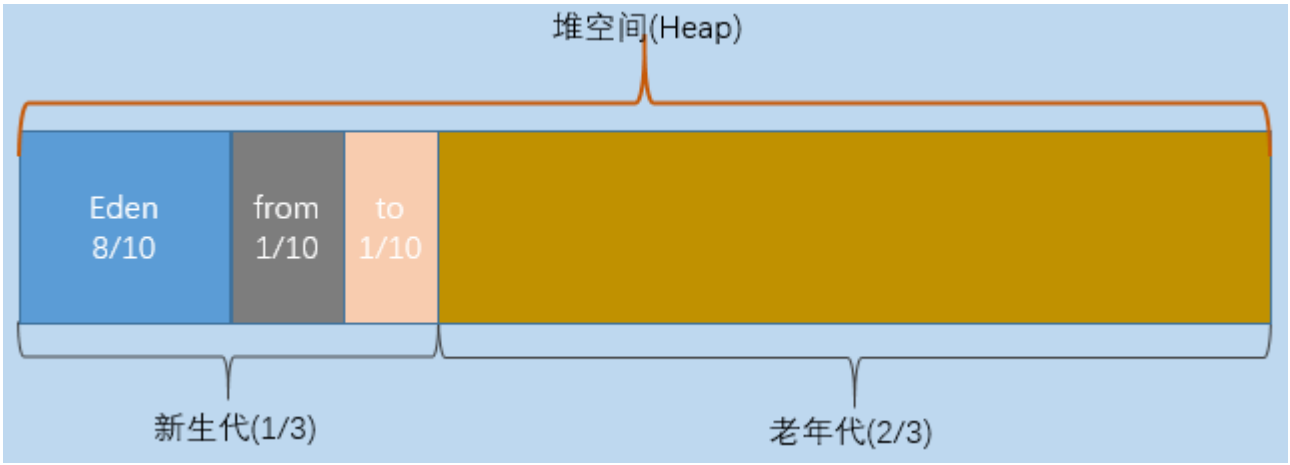
每个线程启动时，还会分配『本地方法栈』内存，来给哪些其它语言实现的方法（称为本地方法）使用。



2.1.4 堆

Java堆通常是Java虚拟机所管理的内存中最大的一块。Java堆是被锁有线程共享的一块内存区域，在虚拟机启动时创建。这块区域唯一的目的就是存放对象实例，几乎所有对象实例及数组都在该区域分配内存，从 JDK1.7 开始，StringTable等也会使用堆内存。

Java堆时垃圾收集器管理的主要区域（GC堆），从内存回收的角度（收集器一般采用分代收集算法），堆被划分为新生代和旧生代，新生代又被进一步划分为Eden（伊甸园） 和 Survivor（幸存区） 区，最后Survivor由FromSpace和ToSpace组成，结构图如下所示：



堆空间内存分配（默认情况下）

老年代 ： 三分之二的堆空间

年轻代 ： 三分之一的堆空间

`eden`区： `8/10` 的年轻代空间

`survivor From : 1/10` 的年轻代空间

`survivor To : 1/10` 的年轻代空间

2.2 非JAVA内存（堆外内存）

2.2.1 元空间

作用：

用来存储类对象，类加载器，静态变量，`StringTable`，`SymbolTable`，即时编译器生成的代码等。

历史：

1). 『方法区』是 `Java VM` 规范中定义的概念，具体实现根据各个虚拟机厂商的不同而不同。对于 `Oracle` 的 `HotSpot` 虚拟机来说，最初作为『方法区』的实现称之为『永久代』，从 `Java 8` 开始，『永久代』被替换为『元空间』。 2). 『永久代』，垃圾回收仍然会考虑『永久代』，但回收效率不高，`StringTable` 最初也使用的是『永久代』内存，容易造成 `OOM` 问题。 3). 『元空间』，使用了操作系统内存，默认没有上限。并且 `StringTable` 的空间被移至堆内存，『元空间』中仅存储类加载器、类对象等信息，垃圾回收不用考虑『元空间』，元空间自己管理内存释放。

方法区内存溢出：

动态生成的类过多

2.2.2 直接内存

定义： 在 NIO 进行 IO 操作时，用到的数据缓冲内存

特点：

典型实现由 DirectByteBuffer，它使用了堆外内存，可以用 allocateDirect 方法创建。好处有：

1)． 没有使用堆内存，减少 GC 压力 2)． I/O 读写操作直接操作堆外内存，省去了系统空间和用户空间的数据拷贝 3)． 堆外内存回收通过虚引用实现

3． 垃圾回收

3.1 判定对象是否是垃圾

3.1.1 引用计数法

有一个地方引用对象，计数加一，当计数为零表示可以回收；

缺点是难以解决对象之间的循环引用问题

3.1.2 可达性分析算法

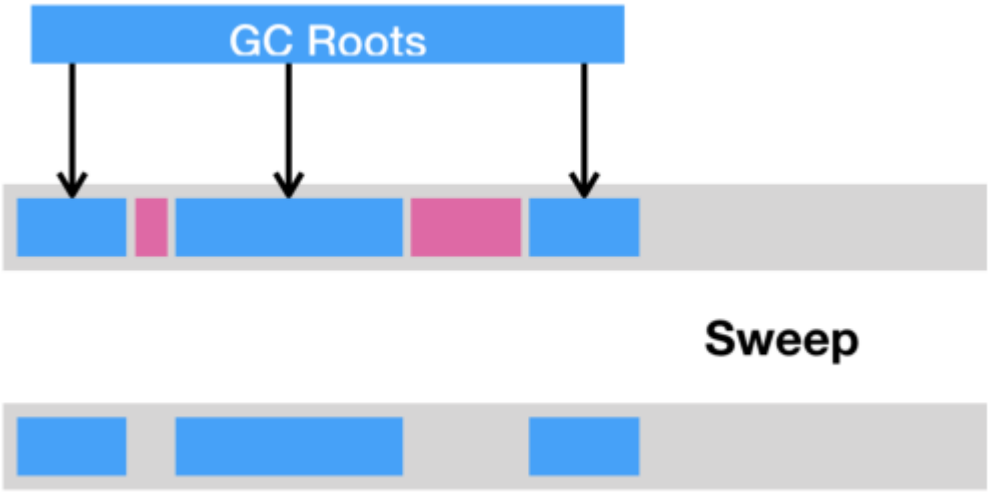
1)． java 虚拟机中的垃圾回收器采用可达性分析来探索所有存活的对象。它从一系列 GC Roots 出发，边标记边探索所有被引用的对象。

2)． 从 GC Root对象 为起点，看是否能沿着引用链找到该对象，找不到，表示可以回收。 3)． GC Root对象包括栈帧中的局部变量、方法区中的静态变量、方法区中的常量、本地方法栈中JNI引用的对象。 4)． 为了防止在标记过程中堆栈的状态发生改变，Java 虚拟机采取安全点机制来实现 Stop-the-world（应用程序的线程全部停止）操作，暂停其他非垃圾回收线程。 5)． 当然，安全点的初始目的并不是让其他线程停下，而是找到一个稳定的执行状态。在这个执行状态下，Java 虚拟机的堆栈不会发生变化。这么一来，垃圾回收器便能够"安全"地执行可达性分析。

3.2 垃圾回收算法

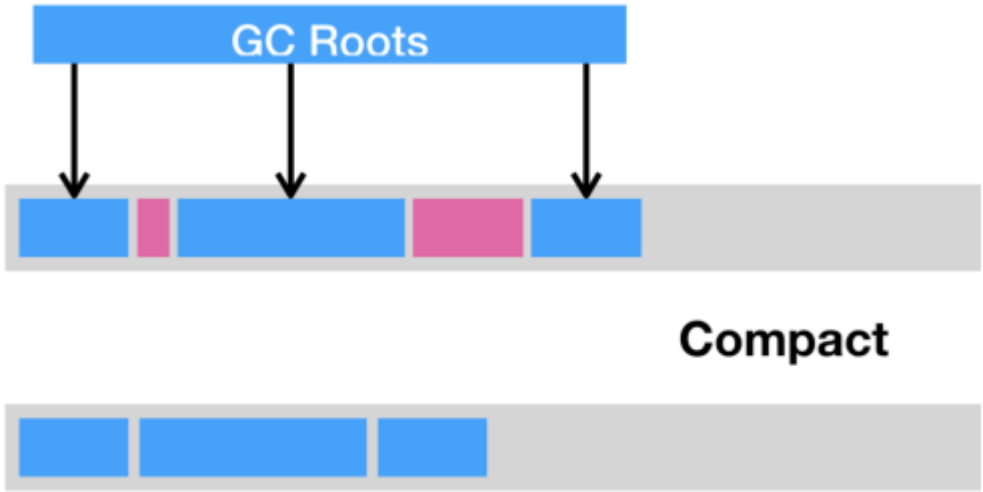
3.2.1 标记清除

第一遍标记、第二遍收集。缺点是会产生内存碎片，碎片过多，仍会使得连续空间少。



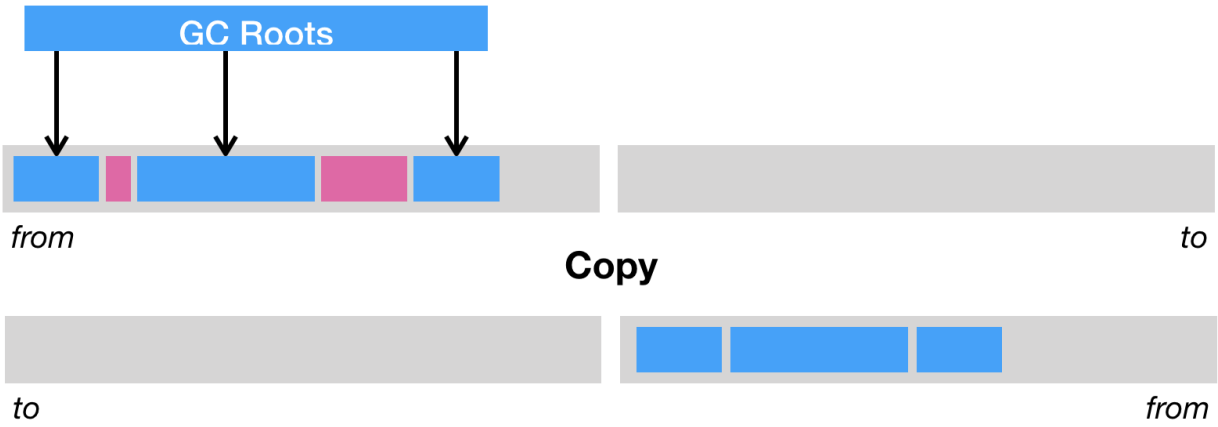
3.2.2 标记整理

第一遍标记、第二遍整理，整理是指存活对象向一端移动来减少内存碎片，相对效率较低。



3.2.3 复制

开辟两份大小相等空间，一份空间始终空着，垃圾回收时，将存活对象拷贝进入空闲空间，优点是不会有内存碎片，但占用空间多。



3.3 分代垃圾回收

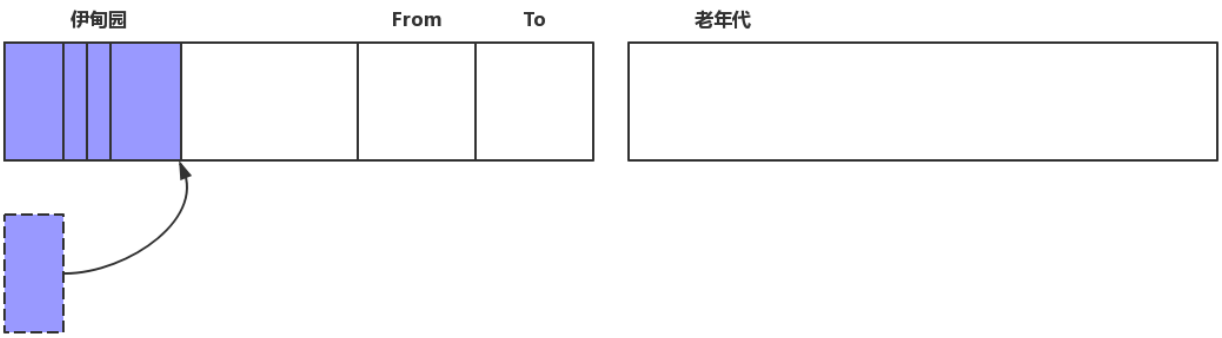
1). 大部分的 Java 对象只存活一小段时间，而存活下来的小部分 Java 对象则会存活很长一段时间。 2).根据对象的特点分代（分区域）来进行，分为新生代和老年代，新生代对象一般很少存活，采用『复制算法』、老年代对象生存时间长，适合采用『标记-清除算法』或『标记-整理算法』 3). 堆内存分为『新生代』和『老年代』，『新生代』又分为『伊甸园』和两个『幸存区』。新生代内存不足触发的 GC 称为 Minor GC ，暂停时间很短，老年代内存不足触发的 GC 称为 Full GC 暂停时间较长，一般是新生代 GC 的几十倍，它们使用的垃圾回收算法不同，见之前的介绍。

3.3.1 尝试在伊甸园分配

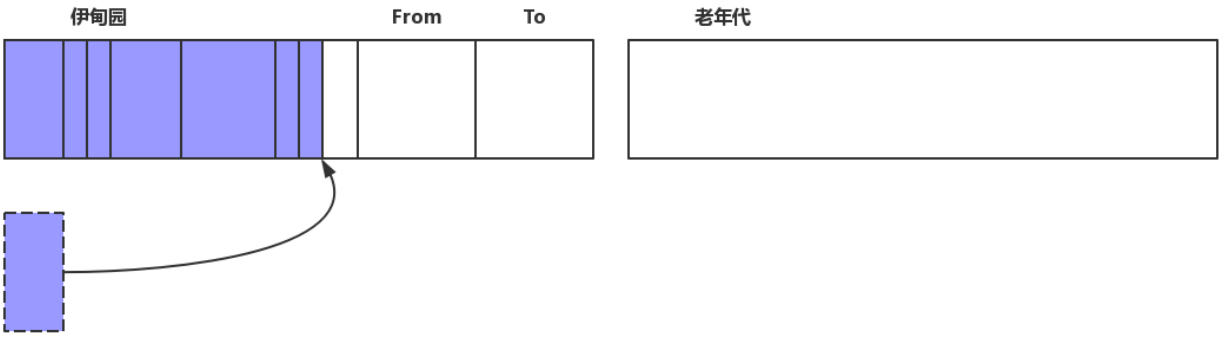
对象优先在『伊甸园』分配，当『伊甸园』没有足够的空间时，触发 Minor GC ，将『伊甸园』和『幸存区 From』中仍然存活的对象利用 复制算法 移入『幸存区 To』，然后交换『幸存区 From』和『幸存区 To』的位置。

```
1  默认情况下，Java 虚拟机采取的是一种动态分配的策略（对应 Java 虚拟机参数 -XX:+UsePSAdaptiveSurvivorSizePolicy），根据生成对象的速率，以及 Survivor区的使用情况动态调整 Eden 区和 Survivor 区的比例。当然，你也可以通过参数 -XX:SurvivorRatio 来固定这个比例。但是需要注意的是，其中一个 Survivor 区会一直为空，因此比例越低浪费的堆空间将越高。
```

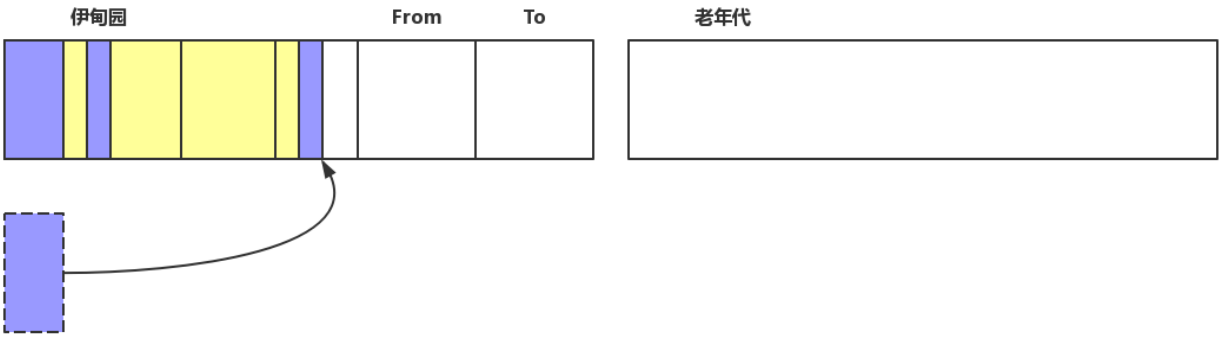
情况1：伊甸园空间还够，新对象在伊甸园能够存储的下，这时候不会发生GC。图中白色区域是空闲空间、蓝色矩形表示已创建对象。



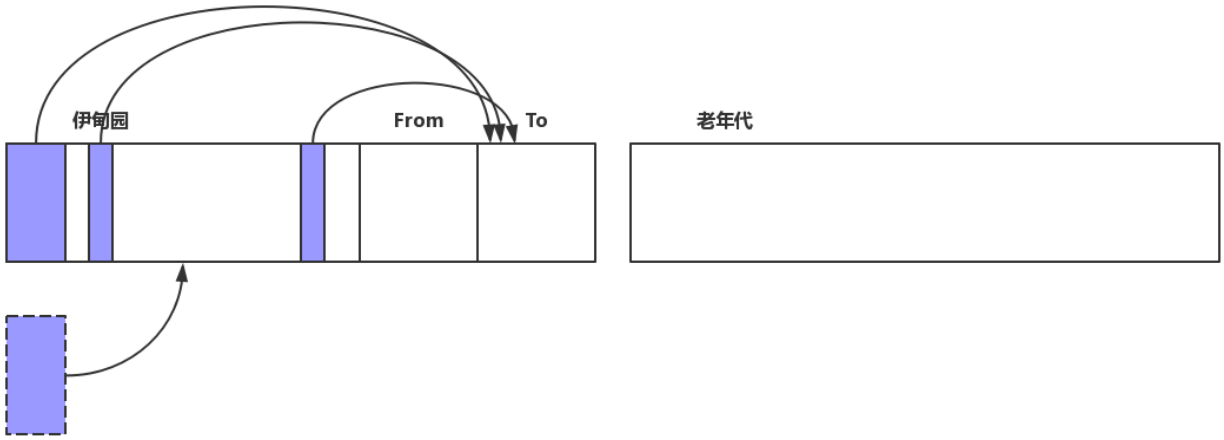
情况2：伊甸园空间不够了。



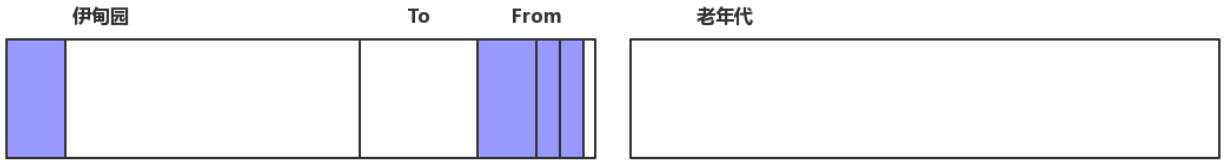
标记可回收的对象，图中用黄色表示，这时候会用户线程会被暂停（`Stop The World`）。



触发新生代的垃圾回收，称为 `Minor GC` ，幸存对象移入『幸存区 `To`』，注意这里用的是复制算法，因此在幸存区没有碎片。

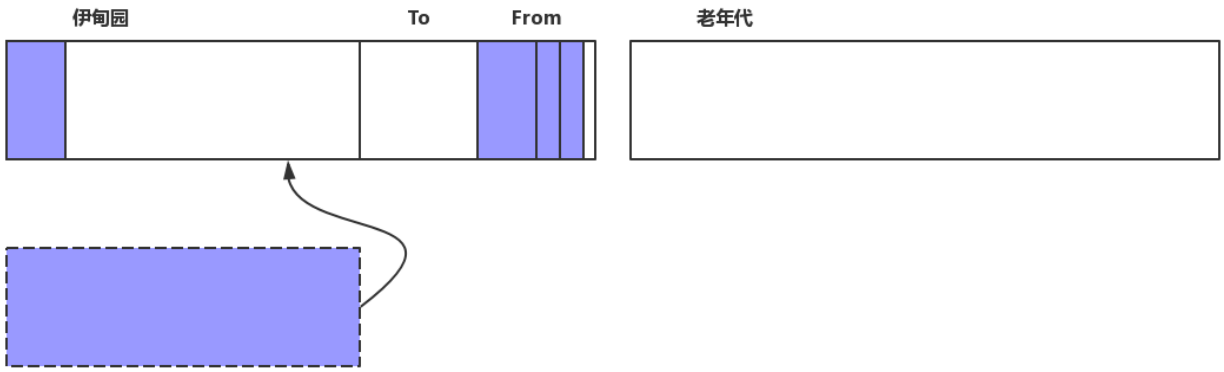


最后的结果，注意 GC 完成后，From 和 To 交换了位置，另外幸存区的对象开始记录寿命。

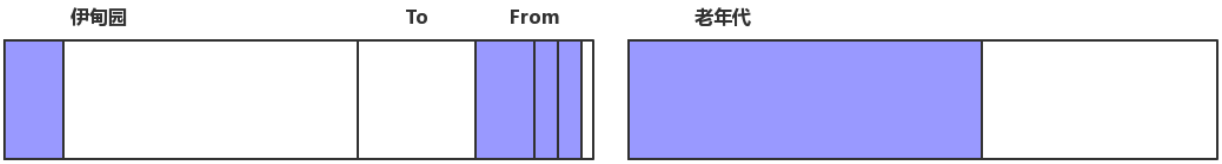


3.3.2 大对象直接晋升至老年代

1)． 当对象太大，伊甸园包括幸存区都存放不下时，这时候老年代的连续空间足够，此对象会直接晋升至老年代，不会发生 GC。



结果



测试：1)．预先定义一组大小

```
    private static final int _512KB = 512 * 1024;
    private static final int _1MB = 1024 * 1024;
    private static final int _6MB = 6 * 1024 * 1024;
    private static final int _7MB = 7 * 1024 * 1024;
    private static final int _8MB = 8 * 1024 * 1024;
```

在运行时添加如下 JVM 参数：

```
-XX:+UseSerialGC -XX:+PrintGCDetails -verbose:gc -Xms20M -Xmx20M -Xmn10M -
XX:SurvivorRatio=8
```

参数含义 ：

```
-XX:+UseSerialGC 是指使用 Serial + SerialOld 回收器组合
-XX:+PrintGCDetails -verbose:gc 是指打印 GC 详细信息
-Xms20M -Xmx20M -Xmn10M 是指分配给 JVM 的最小，最大以及新生代内存
-XX:SurvivorRatio=8 是指『伊甸园』与『幸存区 From』和『幸存区 To』比例为 8:1:1
```

2)．最开始,没什么对象：

```
public static void main(String[] args) {

}
```

测试结果

```
Heap
 def new generation   total 9216K, used 2562K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)
  eden space 8192K,  31% used [0x00000000fec00000, 0x00000000fee809f8, 0x00000000ff400000)
  from space 1024K,   0% used [0x00000000ff400000, 0x00000000ff400000, 0x00000000ff500000)
  to   space 1024K,   0% used [0x00000000ff500000, 0x00000000ff500000, 0x00000000ff600000)
 tenured generation   total 10240K, used 0K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)
   the space 10240K,   0% used [0x00000000ff600000, 0x00000000ff600000, 0x00000000ff600200, 0x0000000100000000)
 Metaspace       used 3119K, capacity 4556K, committed 4864K, reserved 1056768K
  class space    used 333K, capacity 392K, committed 512K, reserved 1048576K
```

3)．当代码改为

```
    public static void main(String[] args) {
        byte[] obj1 = new byte[_7MB];
    }
```

可以预料到，因为「eden space 8192K, 30% used」已经放不下 7MB 的对象，必然会触发新生代的 GC

```
[GC (Allocation Failure) [DefNew: 2398K->673K(9216K), 0.0023117 secs] 2398K->673K(19456K), 0.0023663 secs] [Times: user=0.00 sys=0.00,
Heap
 def new generation   total 9216K, used 8087K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)
  eden space 8192K,  90% used [0x00000000fec00000, 0x00000000ff33d8a0, 0x00000000ff400000)
  from space 1024K,  65% used [0x00000000ff500000, 0x00000000ff5a8508, 0x00000000ff600000)
  to   space 1024K,   0% used [0x00000000ff400000, 0x00000000ff400000, 0x00000000ff500000)
 tenured generation   total 10240K, used 0K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)
   the space 10240K,   0% used [0x00000000ff600000, 0x00000000ff600000, 0x00000000ff600200, 0x0000000100000000)
 Metaspace       used 3120K, capacity 4556K, committed 4864K, reserved 1056768K
  class space    used 333K, capacity 392K, committed 512K, reserved 1048576K
```

4)．可以看到，结果是一部分旧的对象进入了幸存区「from space 1024K, 73% used」，而伊甸园里放入了 7MB 的对象「eden space 8192K, 88% used」再放入一个 512KB 的对象

```
public static void main(String[] args) {
    byte[] obj1 = new byte[_7MB];
    byte[] obj2 = new byte[_512KB];
}
```

```
[GC (Allocation Failure) [DefNew: 2234K->673K(9216K), 0.0023723 secs] 2234K->673K(19456K), 0.0024431 secs] [Times:
Heap
 def new generation   total 9216K, used 8763K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)
  eden space 8192K,  98% used [0x00000000fec00000, 0x00000000ff3e6808, 0x00000000ff400000)
  from space 1024K,  65% used [0x00000000ff500000, 0x00000000ff5a8498, 0x00000000ff600000)
  to   space 1024K,   0% used [0x00000000ff400000, 0x00000000ff400000, 0x00000000ff500000)
 tenured generation   total 10240K, used 0K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)
   the space 10240K,   0% used [0x00000000ff600000, 0x00000000ff600000, 0x00000000ff600200, 0x0000000100000000)
 Metaspace       used 3118K, capacity 4556K, committed 4864K, reserved 1056768K
  class space    used 333K, capacity 392K, committed 512K, reserved 1048576K
```

可以看到，伊甸园几乎被放满了「eden space 8192K, 98% used」，但毕竟没有满，所以没有触发第二次 GC继续放入一个 512KB 的对象.

5)．果然触发了第二次 GC，其中一个 512KB 的对象进入了幸存区「from space 1024K, 52% used」而那个 7MB 的对象晋升至了老年代「tenured generation total 10240K, used 7848K」

```
public static void main(String[] args) {
    byte[] obj1 = new byte[_7MB];
    byte[] obj2 = new byte[_512KB];
    byte[] obj3 = new byte[_512KB];
}
```

```
[GC (Allocation Failure) [DefNew: 2398K->673K(9216K), 0.0020297 secs] 2398K->673K(19456K), 0.0020804 secs] [Times: user=0.00
[GC (Allocation Failure) [DefNew: 8680K->512K(9216K), 0.0052928 secs] 8680K->8279K(19456K), 0.0053244 secs] [Times: user=0.00
Heap
 def new generation   total 9216K, used 1106K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)
  eden space 8192K,   7% used [0x00000000fec00000, 0x00000000fec94930, 0x00000000ff400000)
  from space 1024K,  50% used [0x00000000ff400000, 0x00000000ff480010, 0x00000000ff500000)
  to   space 1024K,   0% used [0x00000000ff500000, 0x00000000ff500000, 0x00000000ff600000)
 tenured generation   total 10240K, used 7767K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)
   the space 10240K,  75% used [0x00000000ff600000, 0x00000000ffd95f98, 0x00000000ffd96000, 0x0000000100000000)
 Metaspace       used 3120K, capacity 4556K, committed 4864K, reserved 1056768K
  class space    used 333K, capacity 392K, committed 512K, reserved 1048576K
```

3.3.3 多次存活的对象

在幸存区历经多次 GC 还存活的对象会晋升至老年代，默认晋升的阈值是 15，也就是说只要经历 15 次回收不死，肯定晋升，但注意如果目标 survivor 空间紧张，也不必等足 15 次，可以提前晋升。

```
-XX:MaxTenuringThreshold=threshold

Sets the maximum tenuring threshold for use in adaptive GC sizing. The largest value
is 15. The default value is 15 for the parallel (throughput) collector, and 6 for the
CMS collector.

-XX:TargetSurvivorRatio=percent

Sets the desired percentage of survivor space (0 to 100) used after young garbage
collection. By default, this option is set to 50%.
```

3.3.4 老年代连续空间不足，触发 Full GC

```
public static void main(String[] args) {
    byte[] obj1 = new byte[_8MB];
    byte[] obj2 = new byte[_8MB];
}
```

第一个 8MB 直接进入老年代，第二个 8MB 对象在分配时发现老年代空间不足，只好尝试先进行一次 Minor GC ，结果发现新生代没有连续空间，只好触发一次 Full GC ，最后发现老年代也没有连续空间，这时出现 OutOfMemoryError

如果把代码改为下面的样子，则只会触发 Minor GC ，之后，老年代能够容纳 obj2，所以不会触发 Full GC ，

```
public static void main(String[] args) {
    byte[] obj1 = new byte[_8MB];
    obj1 = null;;
    byte[] obj2 = new byte[_8MB];
}
```

3.4 垃圾回收器

JVM垃圾回收性能有以下两个主要的指标：

- 吞吐量 ：工作时间（排除GC时间）占总时间的百分比， 工作时间并不仅是程序运行的时间，还包含内存分配时间。
- 暂停时间： 测试时间段内，由垃圾回收导致的应用程序停止响应次数/时间。

GC操作：

```
minor GC： 在新生代进行的GC

major GC： 在老年代进行的GC

Full GC ： 同时作用于新生代和老年代
```

3.4.1 单线程/串行收集器

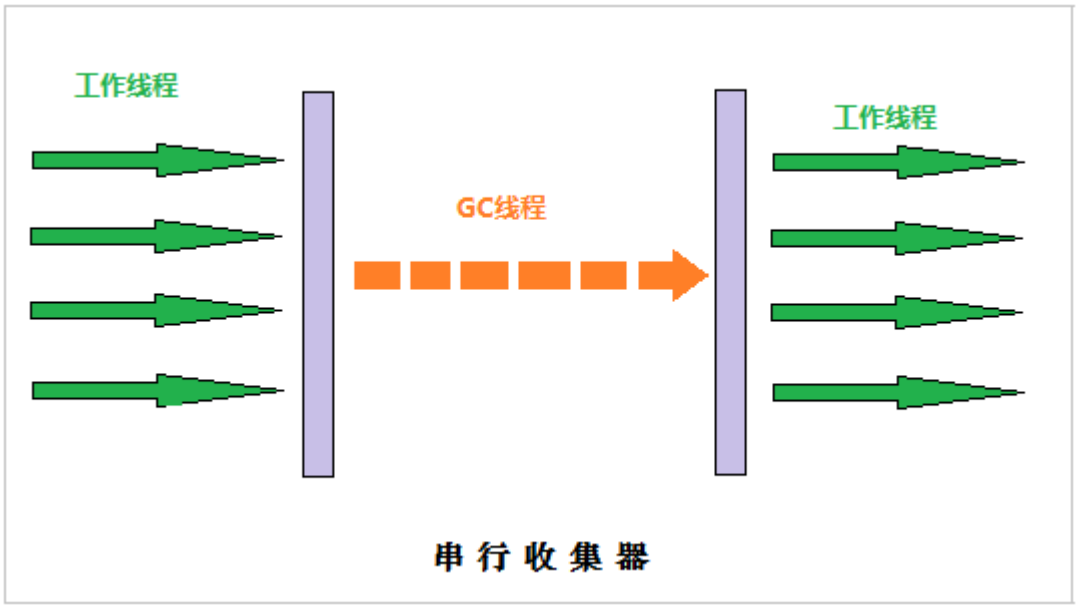
Serial + SerialOld

1). Serial 工作在新生代的单线程收集器，采用『复制算法』，垃圾回收发生时，会暂停所有用户线程

2). SerialOld 工作在老年代的单线程收集器，采用『标记-整理算法』，垃圾回收发生时，会暂停所有用户线程（stop-the-world）

```
配置 ：

    -XX:+UseSerialGC

```

采用单线程执行所有的垃圾回收工作， 适用于单核CPU服务器，无法利用多核硬件的优势

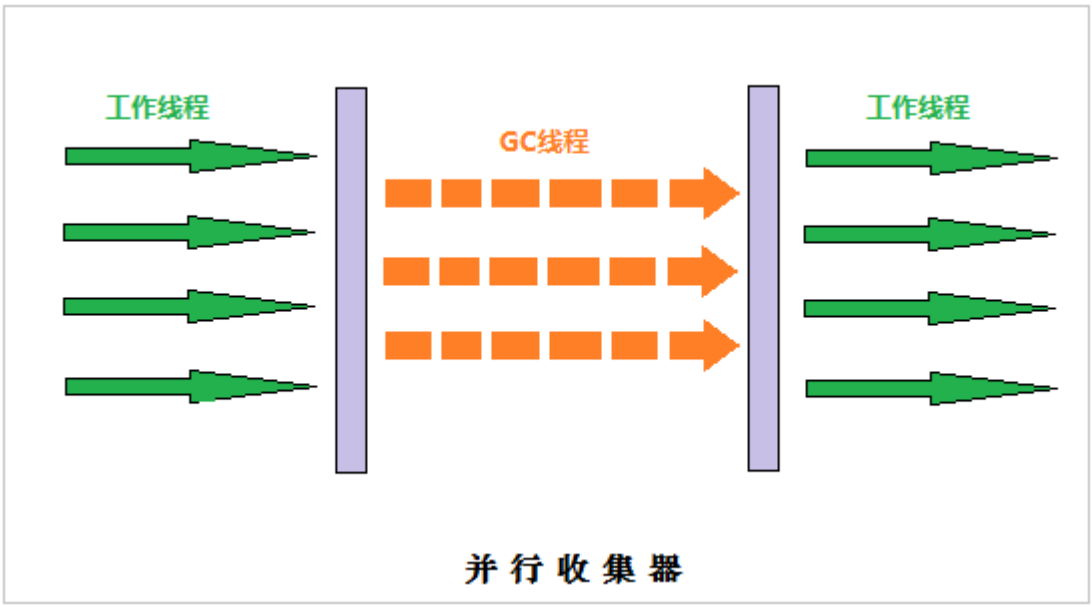


3.4.2 多线程回收器-吞吐量优先

1). Parallel Scavenge 工作在新生代的多线程收集器，采用『复制算法』，垃圾回收发生时，会暂停所有用户线程 单核cpu并不能工作地比Serial好，它的特点是一个以吐量优先 的回收器，下面选项打开 Parallel Scavenge + SerialOld。

2). Parallel Old 工作在老年代的多线程收集器，采用『标记-整理算法』，垃圾回收发生时，会暂停所有用户线程，也是以 吞吐量优先 的回收器，下面选项打开 Parallel Scavenge + Parallel Old。

```
配置

    -XX:+UseParallelGC

    -XX:+UseParallelOldGC
```



3.4.3 多线程回收器-响应时间优先

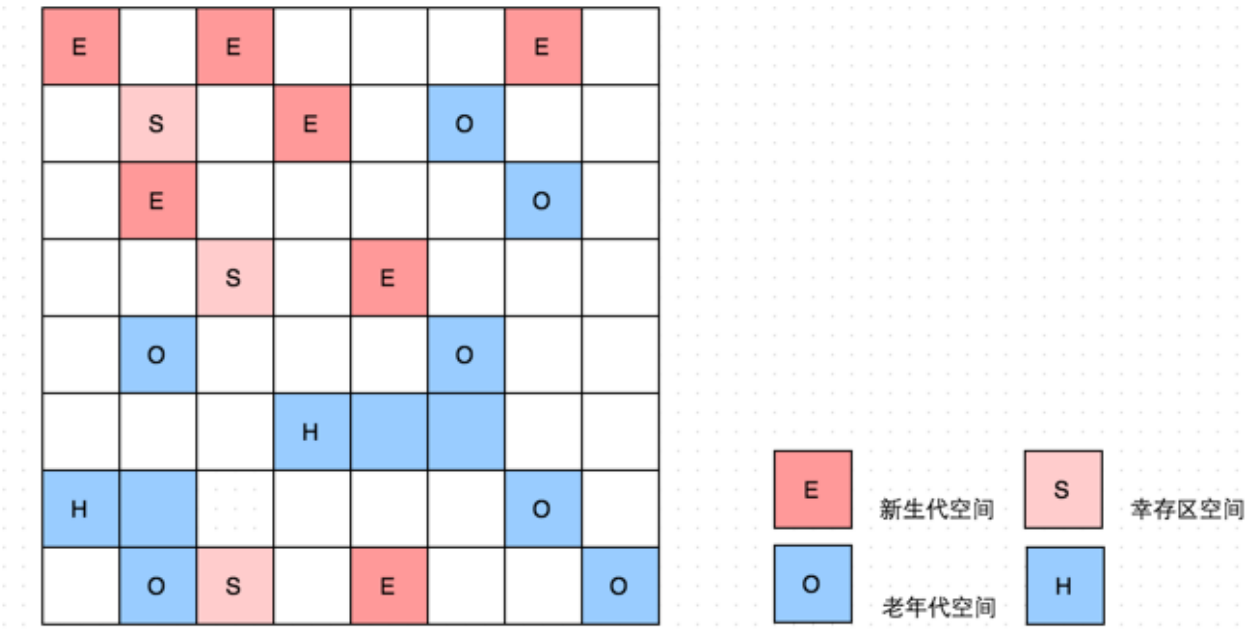
ParNew + SerialOld + CMS

1). ParNew 工作在新生代的多线程收集器，采用『复制算法』，垃圾回收发生时，会暂停所有用户线程，单核 cpu 并不能工作地比 Serial 好。

2). CMS（Concurrent Mark Sweep）用在重视响应速度，停顿时间最短的场合。工作在老年代，基于多线程和『标记-清除算法』，特点是在标记和清理的某些阶段不必暂停用户线程。

3.4.4 G1收集器

G1（Garbage-First）把整个内存区域划分为大小相等的若干区域（region），分为Eden ，Survivor ，Old ，Humongous 四种类型，G1优先回收其中垃圾最多的区域。它采用的算法是 Mark-Copy 不会产生大量内存碎片，它的优势在于可预测的停顿时间 。




```
配置 ：

    -XX:+UseG1GC

```

3.5 GC参数

参数	描述
-XX:+UseSerialGC	启用串行收集器
-XX:+UseParallelGC	启用并行垃圾收集器，配置了该选项，那么 -XX:+UseParallelOldGC 默认启用
-XX:+UseParallelOldGC	FullGC 采用并行收集，默认禁用。如果设置了 -XX:+UseParallelGC 则自动启用
-XX:+UseParNewGC	年轻代采用并行收集器，如果设置了 -XX:+UseConcMarkSweepGC选项，自动启用
-XX:ParallelGCThreads	年轻代及老年代垃圾回收使用的线程数。默认值依赖于JVM使用的CPU个数
-XX:+UseConcMarkSweepGC	对于老年代，启用CMS垃圾收集器。 当并行收集器无法满足应用的延迟需求是，推荐使用CMS或G1收集器。 启用该选项后， -XX:+UseParNewGC 自动启用。
-XX:+UseG1GC	启用G1收集器。 G1是服务器类型的收集器， 用于多核、大内存的机器。它在保持高吞吐量的情况下，高概率满足GC暂停时间的目标。

我们也可以在测试的时候，将JVM参数调整之后，将GC的信息打印出来，便于为我们进行参数调整提供依据，具体参数如下：

选项	描述
-XX:+PrintGC	打印每次GC的信息
-XX:+PrintGCApplicationConcurrentTime	打印最后一次暂停之后所经过的时间， 即响应并发执行的时间
-XX:+PrintGCApplicationStoppedTime	打印GC时应用暂停时间
-XX:+PrintGCDateStamps	打印每次GC的日期戳
-XX:+PrintGCDetails	打印每次GC的详细信息
-XX:+PrintGCTaskTimeStamps	打印每个GC工作线程任务的时间戳
-XX:+PrintGCTimeStamps	打印每次GC的时间戳

如果是在Tomcat中运行 ， 需要在bin/catalina.sh的脚本中 ， 追加如下配置 ：

```
JAVA_OPTS="-XX:+UseConcMarkSweepGC  -XX:+PrintGCDetails"
```

Windows

```
set JAVA_OPTS=-server  -Xms2048m  -Xmx2048m  -XX:MetaspaceSize=256m -
XX:MaxMetaspaceSize=256m -XX:SurvivorRatio=8
```