JVM 与 GC

尚硅谷 JAVA 研究院

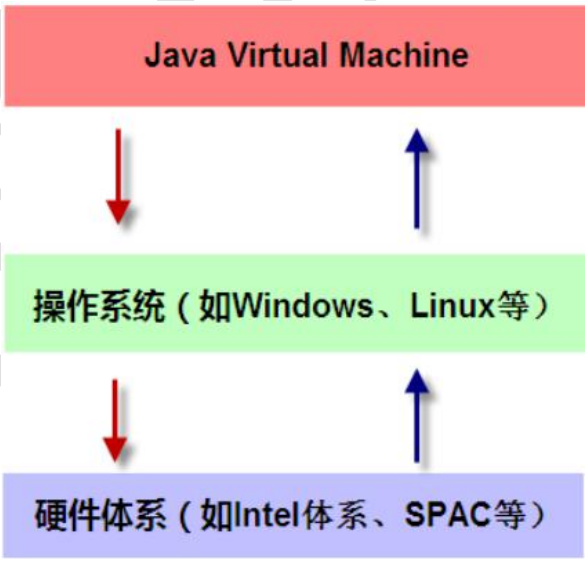
版本：V1.1

第 1 章JVM 简介

1. 是什么

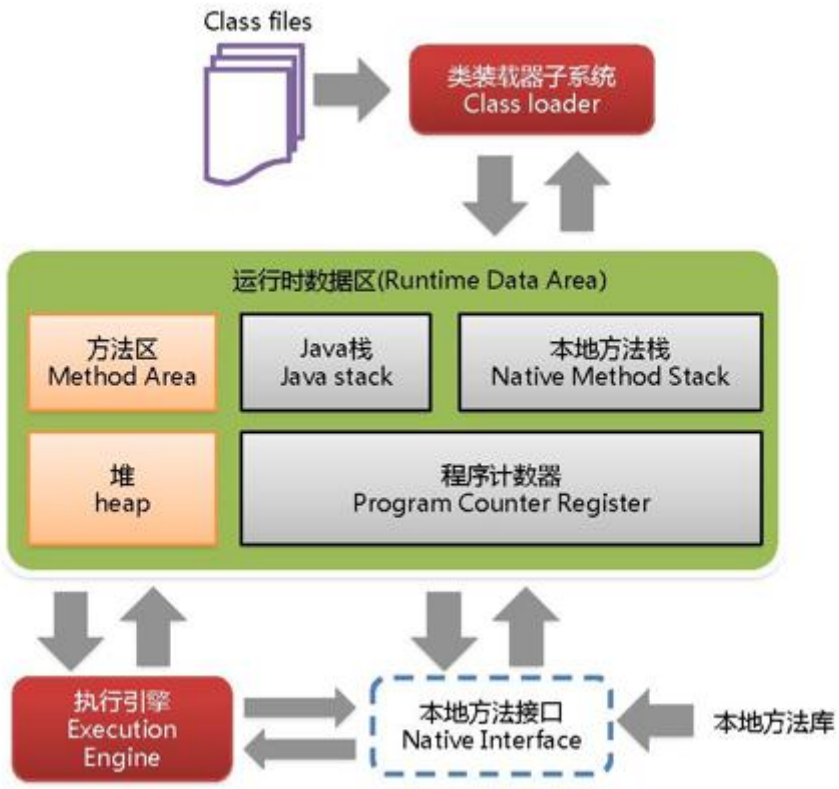
JVM 是 Java Virtual Machine（Java 虚拟机）的缩写，JVM 是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

1.1 JVM 的位置



JVM 是运行在操作系统之上的，它与硬件没有直接的交互

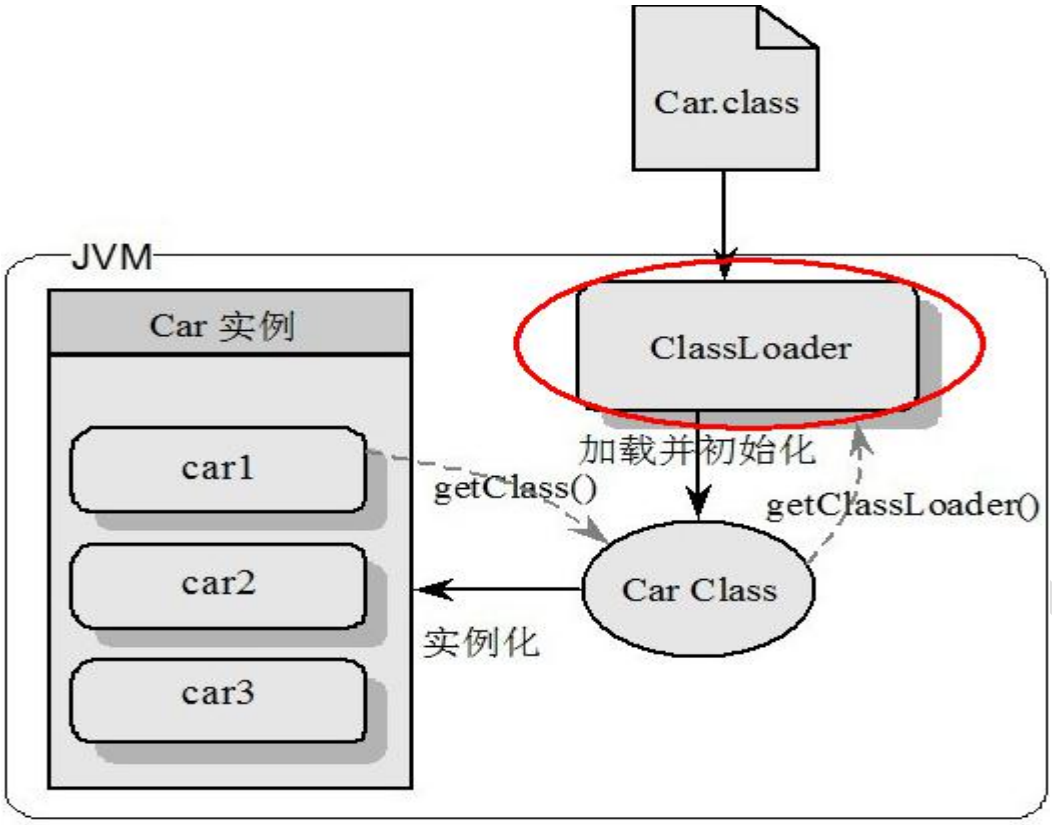
1.2 JVM 体系结构概览



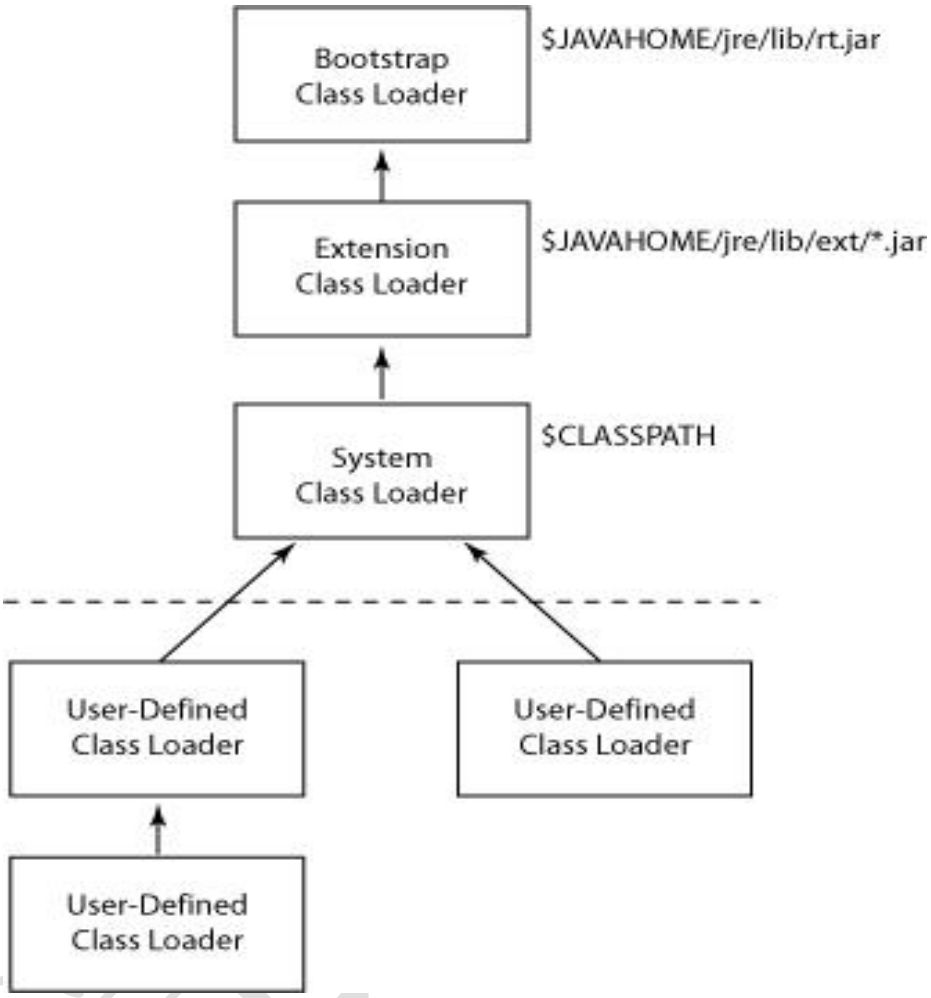
第 2 章类装载器 ClassLoader 、执行引擎 ExecutionEngine

1. 定义

负责加载 class 文件，class 文件在文件开头有特定的文件标示，并且 ClassLoader 只负责 class 文件的加载，至于它是否可以运行，则由 Execution Engine 决定。



2. 类加载器分类



2.1 虚拟机自带的加载器

- 启动类加载器（Bootstrap）C++
- 扩展类加载器（Extension）Java
- 应用程序类加载器（AppClassLoader）Java （也叫系统类加载器，加载当前应用的 classpath 的所有类）

2.2 用户自定义加载器

- Java.lang.ClassLoader 的子类，用户可以定制类的加载方式

2.3 双亲委派与沙箱安全机制

（1） 双亲委派机制：某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。

（2） 沙箱安全机制：网络上有个名叫 java.lang.Integer 的类，它是某个黑客为了想混进 java.lang 包所起的名字，实际上里面含有恶意代码，但是这种 伎俩在双亲模式加载体系结构下是行不通的，因为网络类加载器在加载它的时候，它首先调用双亲类加载器，这样一直向上委托，直到启动类加载器，而启动类加载 器在核心 Java API 里发现了这个名字的类，所以它就直接加载 Java 核心 API 的 java.lang.Integer 类，然后将这个类返回，所以自始自终网络上的 java.lang.Integer 的类是不会被加载的。

2.4 示例代码

```
package com.atguigu.jvm;


public class Demo01 {
private static final String NUMBER="7";
public static void test01() {
}
public static void test02() {
        System.out.println("trst02");
}
        public static int add(int x,int y) {
        int result = -1;
        result = x + y;
        return result;
}
```

```
    public static void main(String[] args) {
        Object obj = new Object();
        Demo01 d01 = new Demo01();
        String str = new String("abc");

        System.out.println(obj.getClass().getClassLoader());
        System.out.println(d01.getClass().getClassLoader().getParent().getParent());
        System.out.println(d01.getClass().getClassLoader().getParent());
        System.out.println(d01.getClass().getClassLoader());
    }

}
```

3. 执行引擎 ExecutionEngine

执行引擎负责解释命令，提交操作系统执行。

第 3 章Native、PC 寄存器

1. 本地方法栈 Native Method Stack

它的具体做法是 Native Method Stack 中登记 native 方法，在 Execution Engine 执行时加载本地方法库。

2. 本地接口 Native Interface

本地接口的作用是融合不同的编程语言为 Java 所用，它的初衷是融合 C/C++程序，Java 诞生的时候是 C/C++横行的时候，要想立足，必须有调用 C/C++程序，于是就在内存中专门开辟了一块区域处理标记为 native 的代码，它的具体做法是 Native Method Stack 中登记 native 方法，在 Execution Engine 执行时加载 native libraies。

目前该方法使用的越来越少了，除非是与硬件有关的应用，比如通过 Java 程序驱动打印机或者 Java 系统管理生产设备，在企业级应用中已经比较少见。因为现在的异构领域间的通信很发达，比如可以使用 Socket 通信，也可以使用 Web Service 等等，不多做介绍。

3. PC 寄存器

每个线程都有一个程序计数器，是线程私有的,就是一个指针，指向方法区中的方法字节码（用来存储指向下一条指令的地址,也即将要执行的指令代码），由执行引擎读取下一条指令，是一个非常小的内存空间，几乎可以忽略不记。

第 4 章方法区 Method Area

方法区是被所有线程共享，所有字段和方法字节码，以及一些特殊方法如构造函数，接口代码也在此定义。简单说，所有定义的方法的信息都保存在该区域，此区属于共享区间。

静态变量+常量+类信息(构造方法/接口定义)+运行时常量池存在方法区中。

实例变量存在堆内存中,和方法区无关。

第 5 章Java 栈 Java Stack

1. Stack 栈是什么

栈也叫栈内存，主管 Java 程序的运行，是在线程创建时创建，它的生命期是跟随线程的生命期，线程结束栈内存也就释放，**对于栈来说不存在垃圾回收问题**，只要线程一结束该栈就 Over，生命周期和线程一致，是线程私有的。8 种基本类型的变量+对象的引用变量+实例方法都是在函数的栈内存中分配。

2. 栈存储什么

栈帧中主要保存 3 类数据：

本地变量（Local Variables）:输入参数和输出参数以及方法内的变量；

栈操作（Operand Stack）:记录出栈、入栈的操作；

栈帧数据（Frame Data）:包括类文件、方法等等。

3. 栈运行原理

栈中的数据都是以栈帧（Stack Frame）的格式存在，栈帧是一个内存区块，是一个数据集，是一个有关方法(Method)和运行期数据的数据集，当一个方法 A 被调用时就产生了一个栈帧 F1，并被压入到栈中，

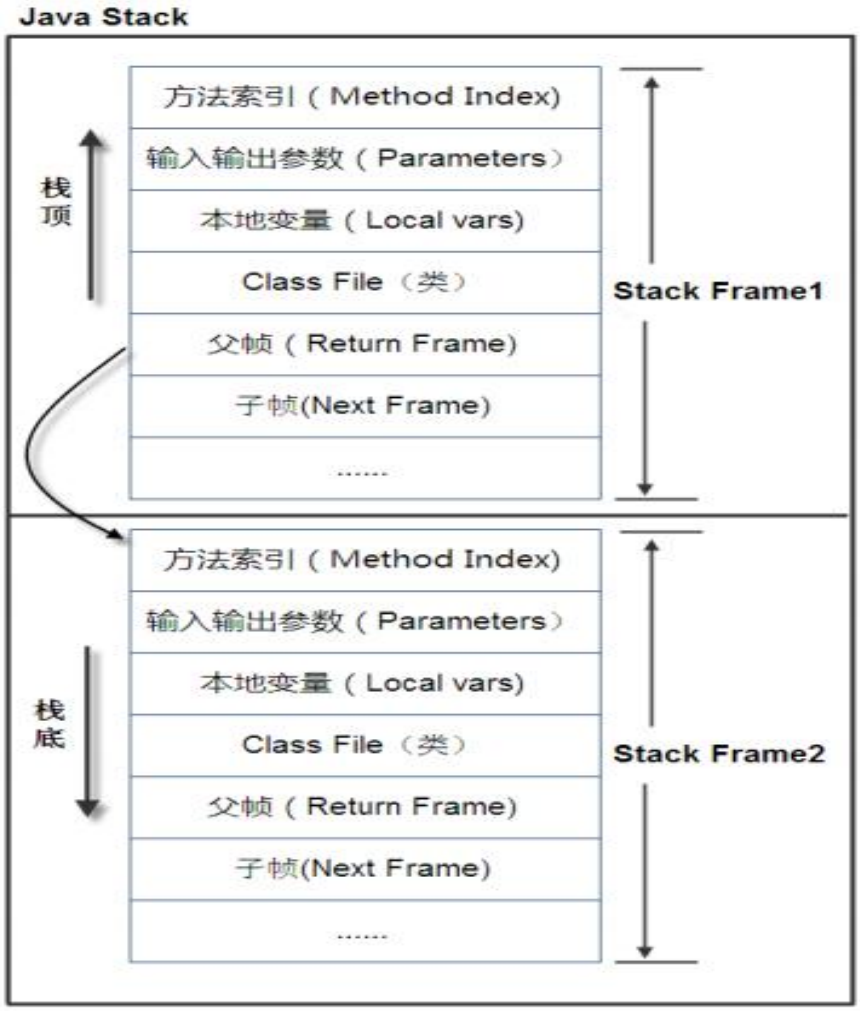
A 方法又调用了 B 方法，于是产生栈帧 F2 也被压入栈，

B 方法又调用了 C 方法，于是产生栈帧 F3 也被压入栈，

……

执行完毕后，先弹出 F3 栈帧，再弹出 F2 栈帧，再弹出 F1 栈帧……

遵循“先进后出”/“后进先出”原则。



图示在一个栈中有两个栈帧：

栈帧 2 是最先被调用的方法，先入栈，

然后方法 2 又调用了方法 1，栈帧 1 处于栈顶的位置，

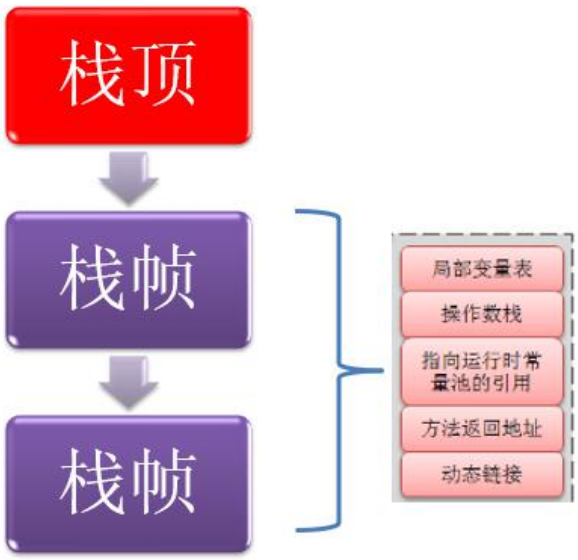
栈帧 2 处于栈底，执行完毕后，依次弹出栈帧 1 和栈帧 2，

线程结束，栈释放。

每执行一个方法都会产生一个栈帧，保存到栈(后进先出)的**顶部，顶部栈就是当前的方法，该方法执行完毕 后会自动将此栈帧出栈**。

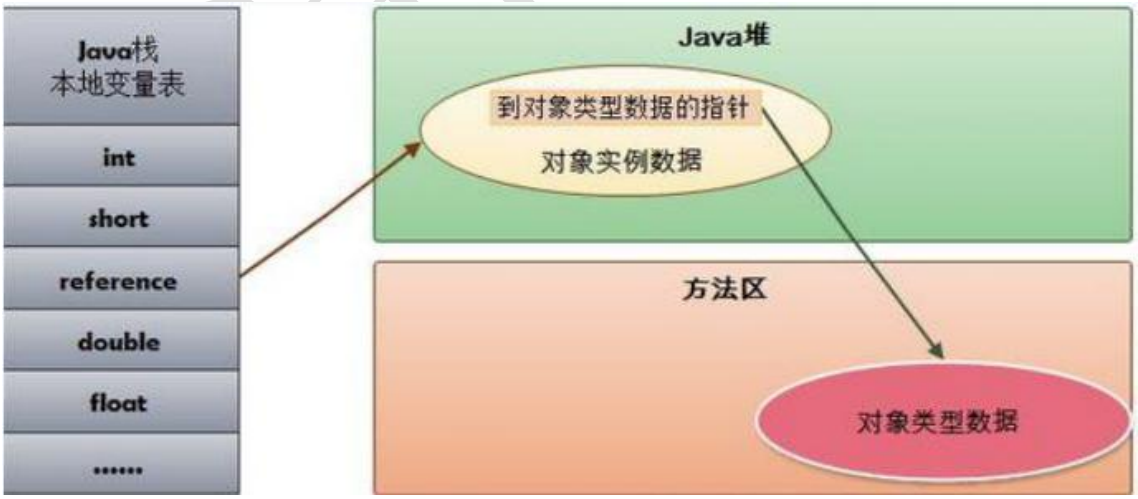
4. 栈内存溢出

栈内存溢出错误：Exception in thread "main" java.lang.StackOverflowError



如果一个线程在计算时所需要用到栈大小 > 配置允许最大的栈大小，那么 Java 虚拟机将抛出 StackOverflowError。

5. 栈、堆、方法区的交互关系



HotSpot 是使用指针的方式来访问对象：Java 堆中会存放访问类元数据的地址，reference 存储的就直接是对象的地址

6. 三种 JVM

- Sun 公司的 HotSpot
- BEA 公司的 JRockit
- IBM 公司的 J9 VM

第 6 章堆 Heap

1. Heap 堆是什么

一个 JVM 实例只存在一个堆内存，堆内存的大小是可以调节的。类加载器读取了类文件后，需要把类、方法、常变量放到堆内存中，保存所有引用类型的真实信息，以方便执行器执行，堆内存分为三部分：

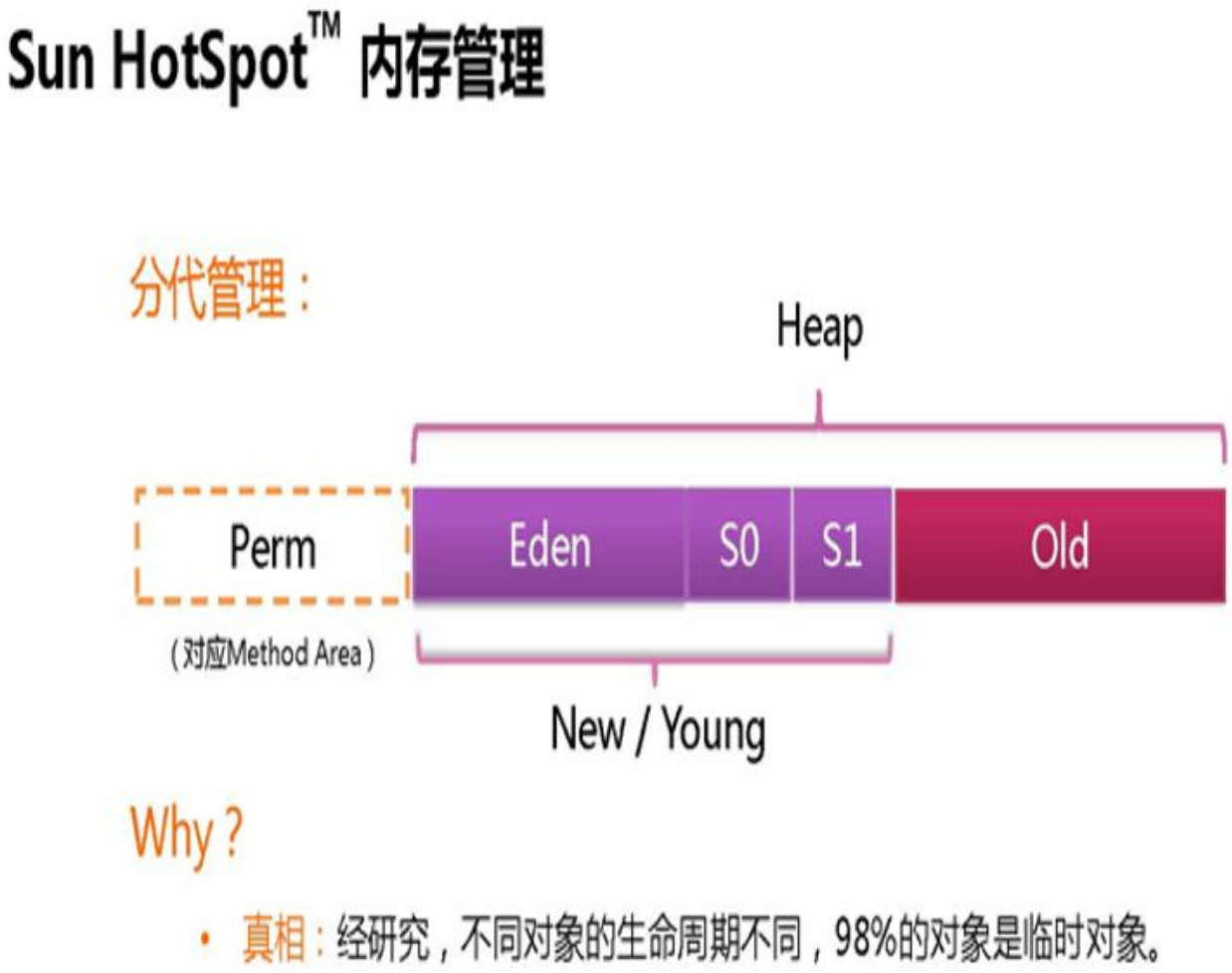
- Young Generation Space 新生区 Young/New
- Tenure generation space 养老区 Old/ Tenure
- Permanent Space 永久区 Perm

2. 堆的划分（Java7 之前）

一个 JVM 实例只存在一个堆内存，堆内存的大小是可以调节的。类加载器读取了类文件后，需要把类、方法、常变量放到堆内存中，保存所有引用类型的真实信息，以方便执行器执行。

堆内存逻辑上分为三部分：新生+养老+永久

堆内存物理上分为二部分：新生+养老



3. 新生区

新生区是类的诞生、成长、消亡的区域，一个类在这里产生，应用，最后被垃圾回收器收集，结束生命。新生区又分为两部分：

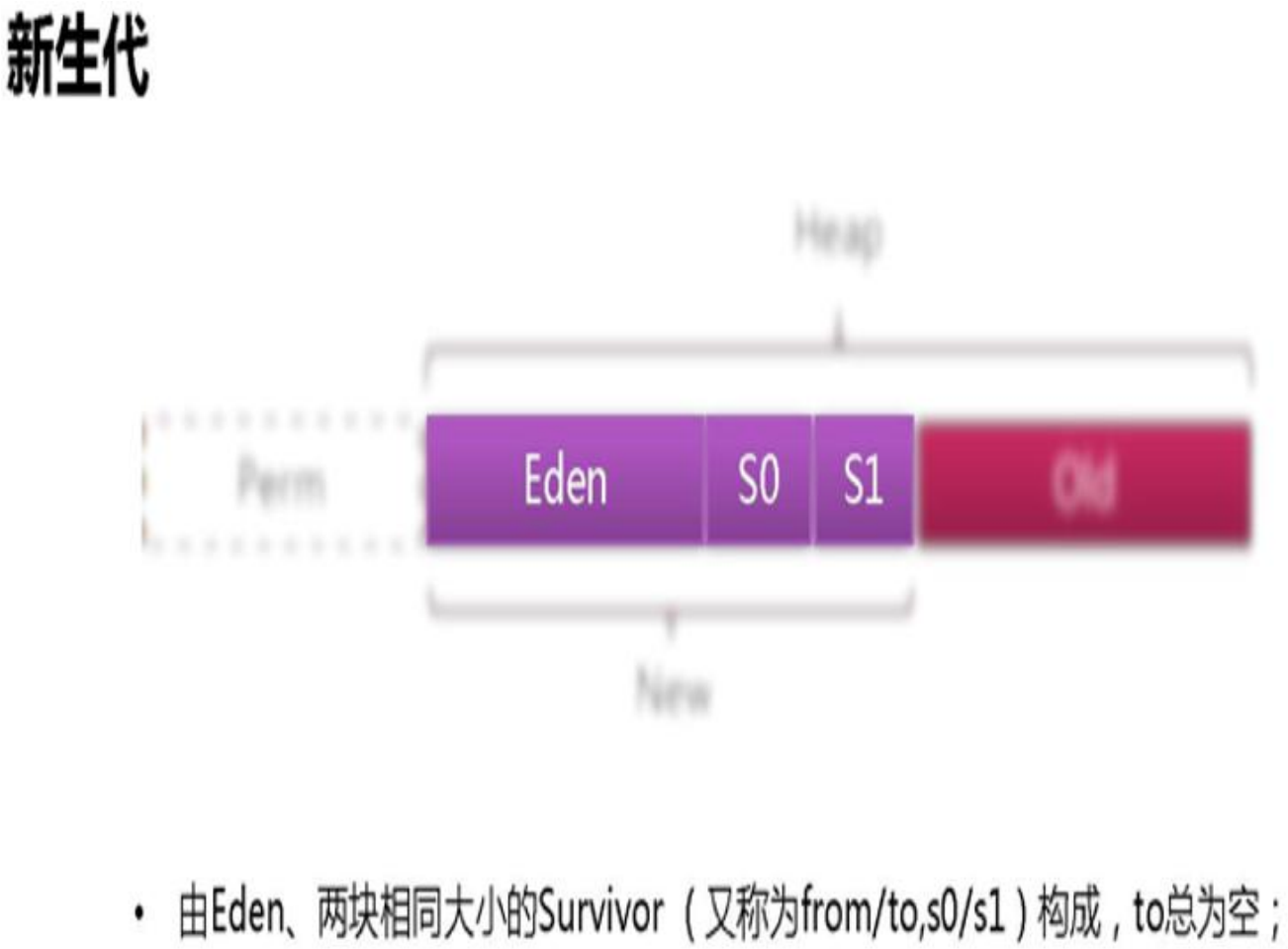
伊甸区（Eden space）

幸存者区（Survivor pace）

所有的类都是在伊甸区被 new 出来的。

幸存区有两个： 0 区（Survivor 0 space）和 1 区（Survivor 1 space）。当伊甸园的空间用完时，程序又需要创建

对象，JVM 的垃圾回收器将对伊甸园区进行垃圾回收(Minor GC)，将伊甸园区中的不再被其他对象所引用的对象进行销毁。然后将伊甸园中的剩余对象移动到幸存 0 区。若幸存 0 区也满了，再对该区进行垃圾回收，然后移动到 1 区。那如果 1 区也满了呢？再移动到养老区。



4. 养老区

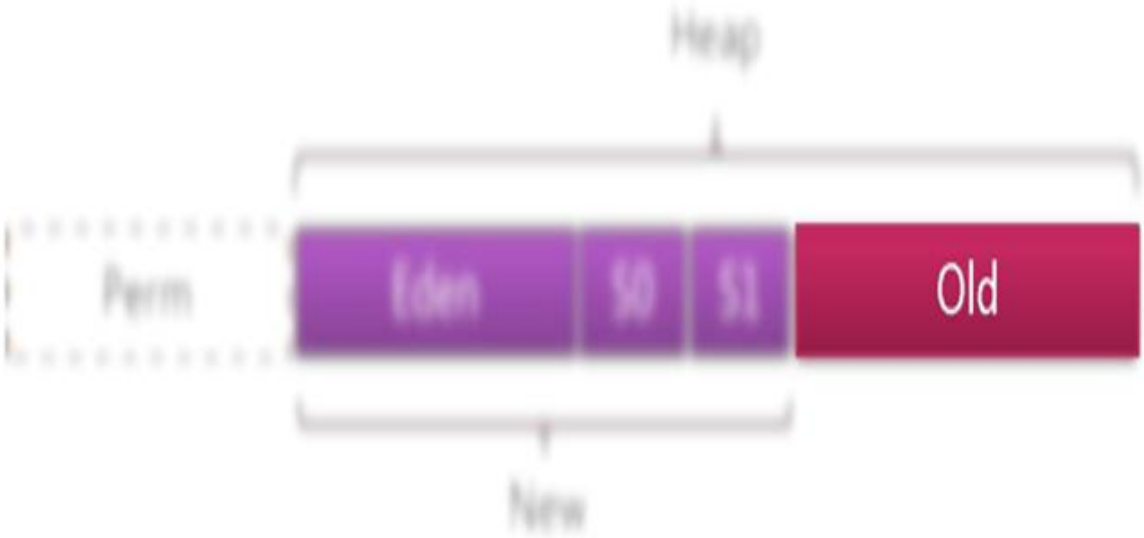
当对象在新生区经历过多次（默认 15 次）GC 依然幸存则进入养老区。若养老区也满了，那么这个时候将产生 MajorGC（FullGC），进行养老区的内存清理。若养老区执行了 Full GC 之后发现依然无法进行对象的保存，就会产生 OOM 异常“OutOfMemoryError”。

如果出现 java.lang.OutOfMemoryError: Java heap space 异常，说明 Java 虚拟机的堆内存不够。原因有二：

（1）Java 虚拟机的堆内存设置不够，可以通过参数-Xms、-Xmx 来调整。

（2）代码中创建了大量大对象，并且长时间不能被垃圾收集器收集（存在被引用）。

旧生代



- 存放新生代中经历多次GC仍然存活的对象；

5. 永久区

永久存储区是一个常驻内存区域，用于存放 JDK 自身所携带的 Class,Interface 的元数据，也就是说它存储的是运行环境必须的类信息，被装载进此区域的数据是不会被垃圾回收器回收掉的，关闭 JVM 才会释放此区域所占用的内存。

如果出现 java.lang.OutOfMemoryError: PermGen space，说明是 Java 虚拟机对永久代 Perm 内存设置不够。一般出现这种情况，都是程序启动需要加载大量的第三方 jar 包。例如：在一个 Tomcat 下部署了太多的应用。或者大量动态反射生成的类不断被加载，最终导致 Perm 区被占满。

Jdk1.6 及之前： 有永久代，常量池 1.6 在方法区

Jdk1.7： 有永久代，但已经逐步“去永久代”，常量池 1.7 在堆

Jdk1.8 及之后： 无永久代，常量池 1.8 在元空间

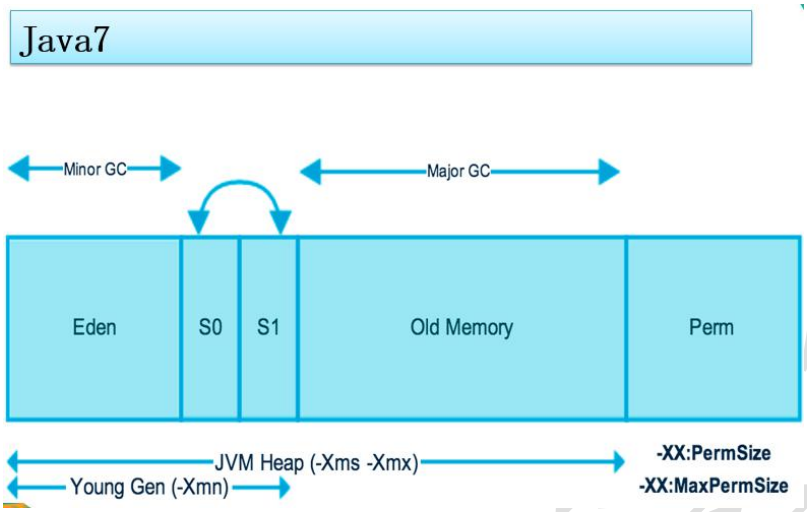
实际而言，方法区（Method Area）和堆一样，是各个线程共享的内存区域，它用于存储虚拟机加载的：类信息+普通常量+静态常量+编译器编译后的代码等等，虽然 JVM 规范将方法区描述为堆的一个逻辑部分，但它却还有一个别名叫做 Non-Heap(非堆)，目的就是要和堆分开。

对于 HotSpot 虚拟机，很多开发者习惯将方法区称之为“永久代(Parmanent Gen)”，但严格本质上说两者不同，或者说使用永久代来实现方法区而已，永久代是方法区(相当于是一个接口 interface)的一个实现，jdk1.7 的版本中，已经将原本放在永久代的字符串常量池移走。

常量池（Constant Pool）是方法区的一部分，Class 文件除了有类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池，这部分内容将在类加载后进入方法区的运行时常量池中存放。

第 7 章堆参数调优

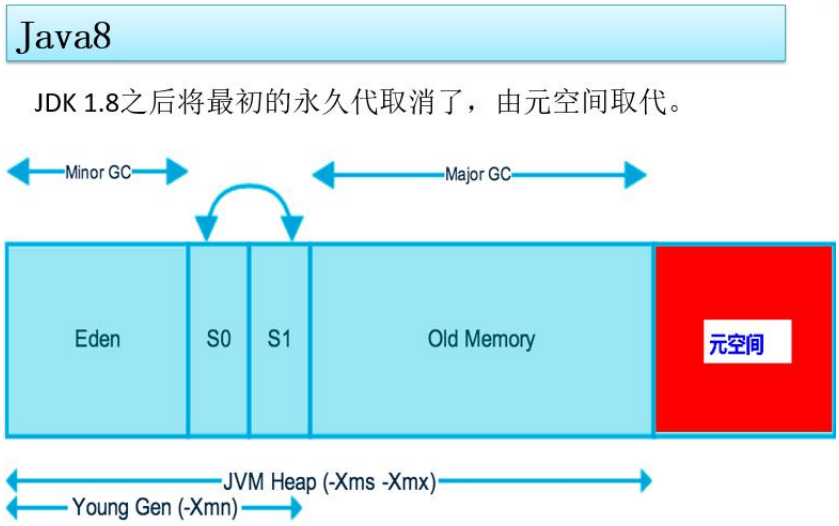
1. Java7 堆内存分布



参数-Xms：初始内存大小

参数-Xmx：最大内存大小

2. Java8 堆内存分布



3. 堆内存调优

3.1 调优参数

-Xms	设置初始分配大小，默认为物理内存的“1 / 64”
-Xmx	最大分配内存，默认为物理内存的 “1 / 4”
-XX:+PrintGCDetails	输出详细的GC处理日志

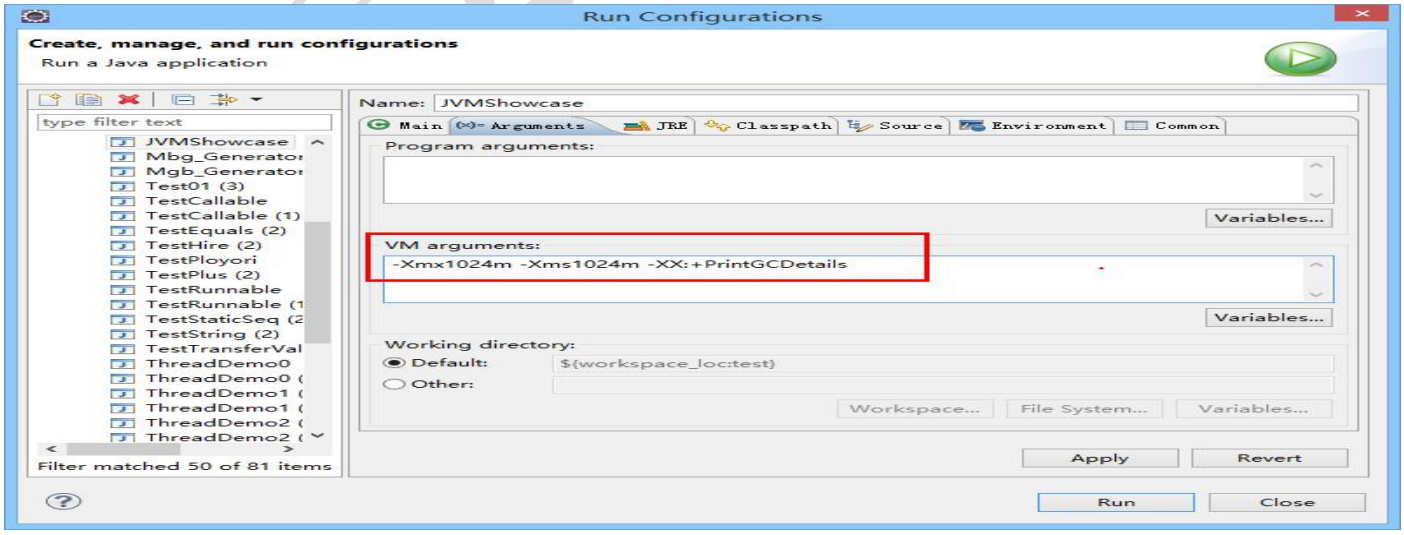
代码：

```
public static void main(String[] args){
    long maxMemory = Runtime.getRuntime().maxMemory() ;//返回 Java 虚拟机试图使用的最大内存量。
    long totalMemory = Runtime.getRuntime().totalMemory() ;//返回 Java 虚拟机中的内存总量。
    System.out.println("MAX_MEMORY = " + maxMemory + "（字节）、" + (maxMemory / (double)1024 / 1024) + "MB");
    System.out.println("TOTAL_MEMORY = " + totalMemory + "（字节）、" + (totalMemory / (double)1024 / 1024) + "MB");
}
```

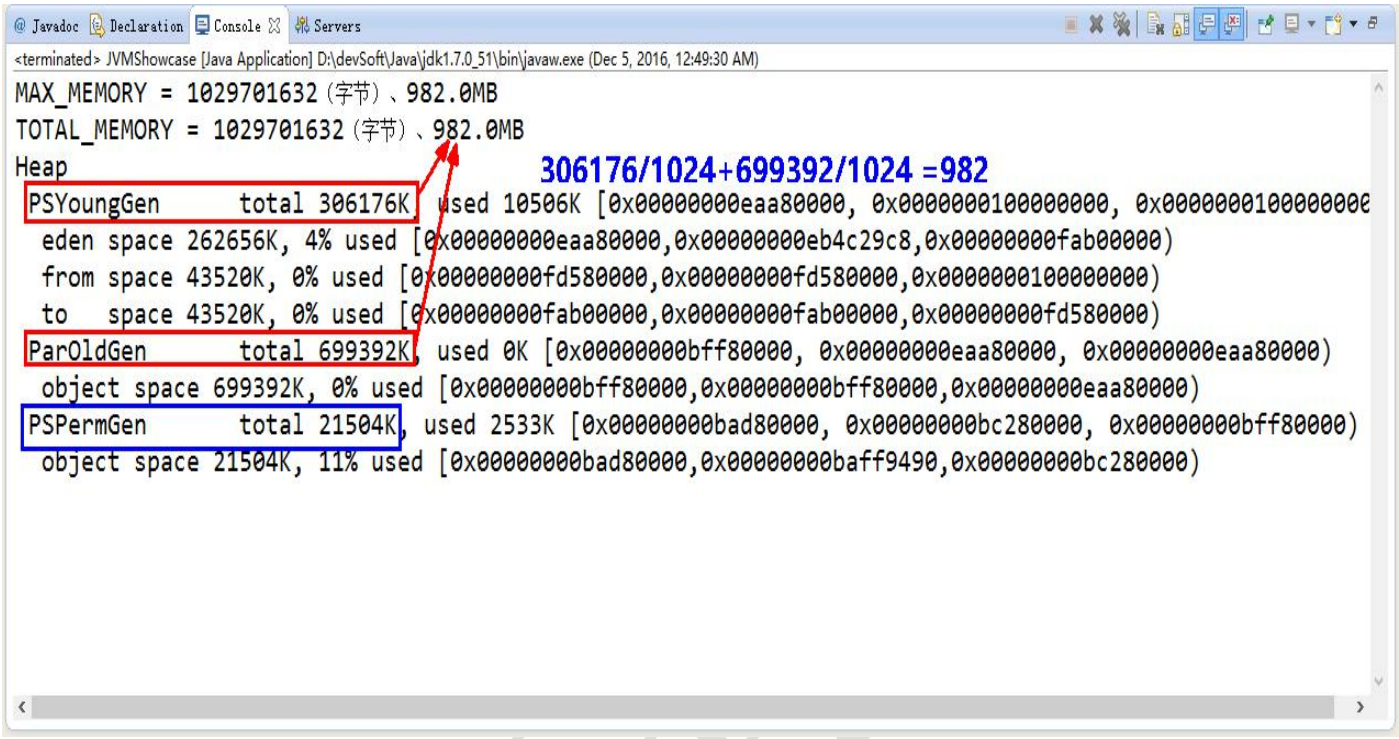
3.2 调整内存大小

发现默认的情况下分配的内存是总内存的“1 / 4”、而初始化的内存为“1 / 64”

参数调整：-Xms1024m -Xmx1024m -XX:+PrintGCDetails



3.3 运行后结果（Java7）



3.4 内存调小，出现 OOM

代码：

```
String str = "www.atguigu.com" ;

while(true)

{

str += str + new Random().nextInt(88888888) + new Random().nextInt(999999999) ;

}
```

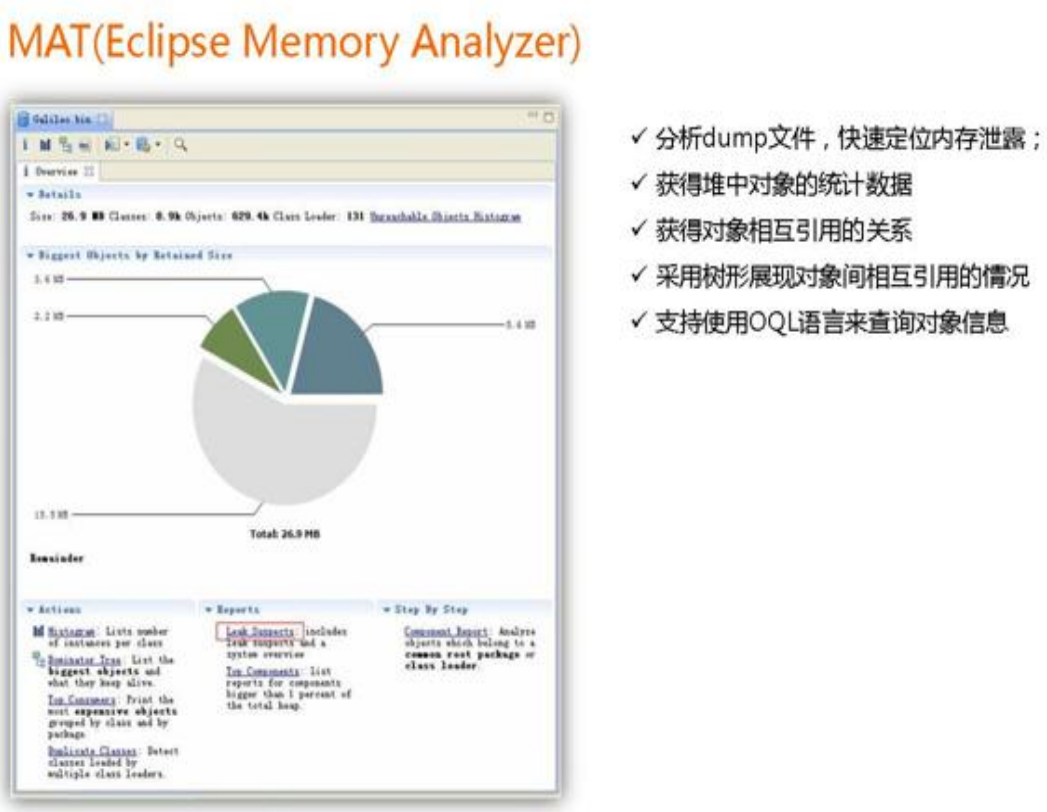
调整内存大小：-Xms8m -Xmx8m -XX:+PrintGCDetails

运行结果：

```
[GC (Allocation Failure) [PSYoungGen: 30K->0K(2048K)] 4643K->4613K(7680K), 0.0003931 secs]
[GC (Allocation Failure) [PSYoungGen: 0K->0K(2048K)] 4613K->4613K(7680K), 0.0003334 secs]
[Full GC (Allocation Failure) [PSYoungGen: 0K->0K(2048K)] [ParOldGen: 4613K->4613K(5632K)]
[GC (Allocation Failure) [PSYoungGen: 0K->0K(2048K)] 4613K->4613K(7680K), 0.0002412 secs]
[Full GC (Allocation Failure) [PSYoungGen: 0K->0K(2048K)] [ParOldGen: 4613K->4600K(5632K)]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.util.Arrays.copyOf(Unknown Source)
```

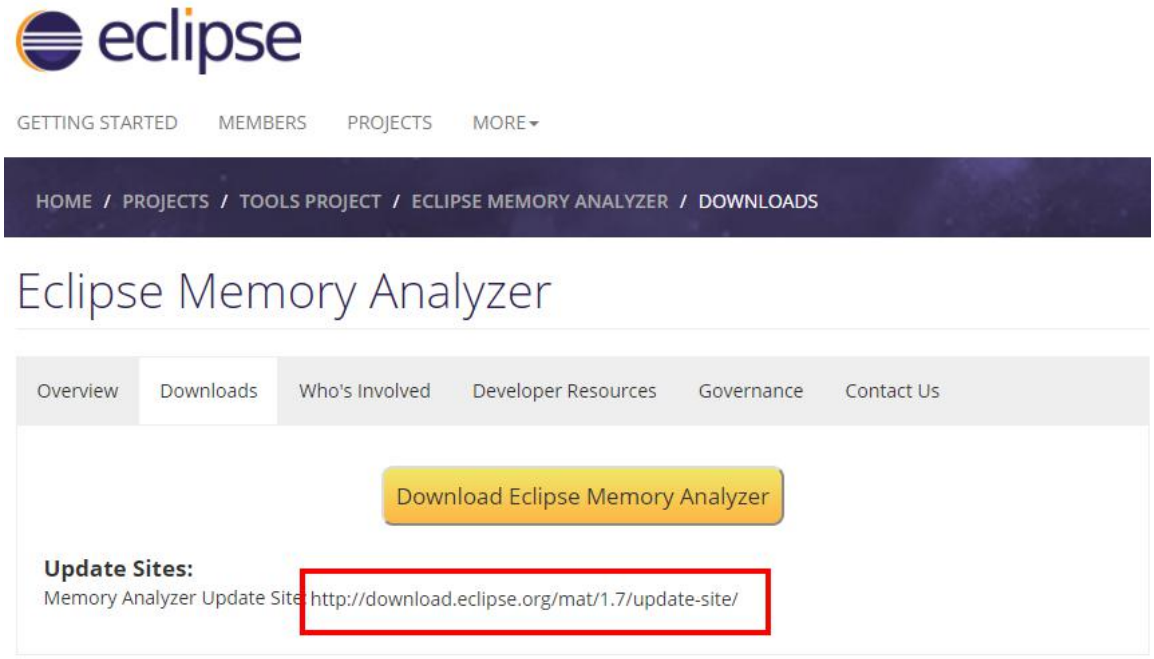
4. 内存溢出定位

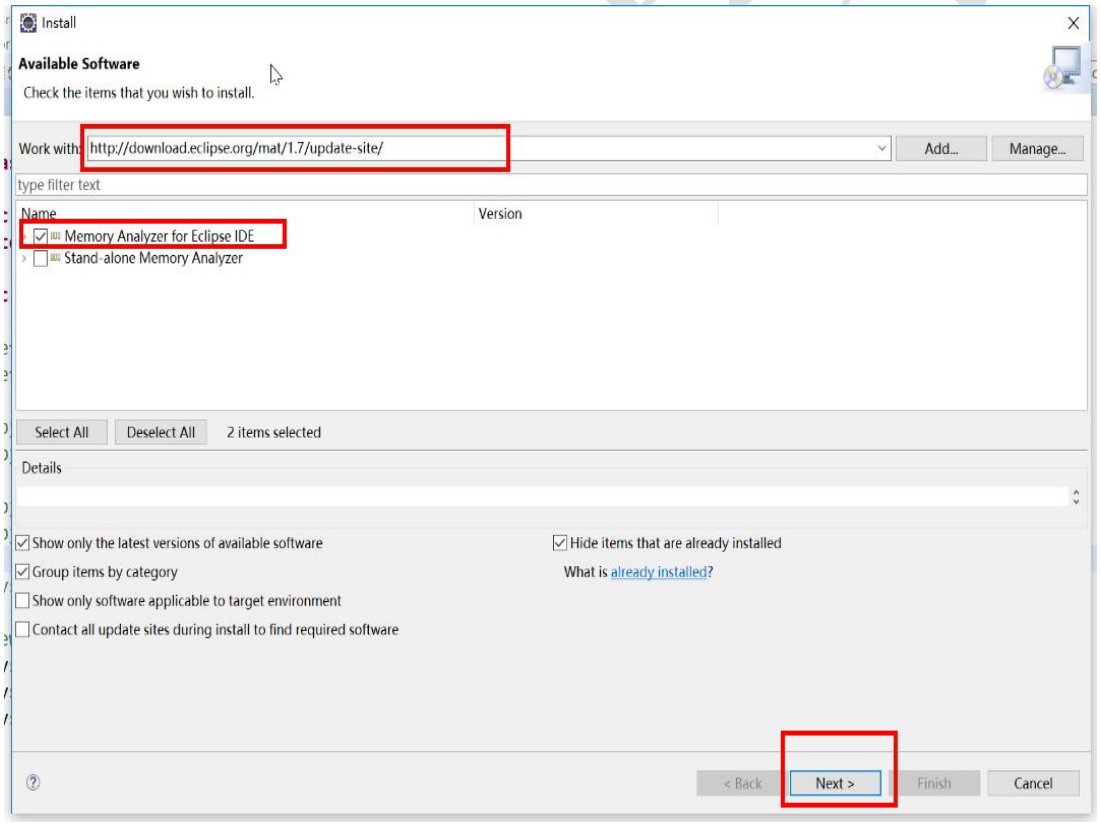
4.1 内存溢出定位工具——MAT



4.2 MAT 下载

官网访问地址：https://projects.eclipse.org/projects/tools.mat/downloads





安装插件

4.2 MAT 使用

运行参数-XX:+HeapDumpOnOutOfMemoryError 如果出现 OOM 时导出堆到文件

（1） 安装好插件后，调整参数-Xms1m -Xmx8m -XX:+HeapDumpOnOutOfMemoryError

（2） 刷新 eclipse 目录列表，打开 dump 文件，查看内存溢出分析报告

第 8 章GC

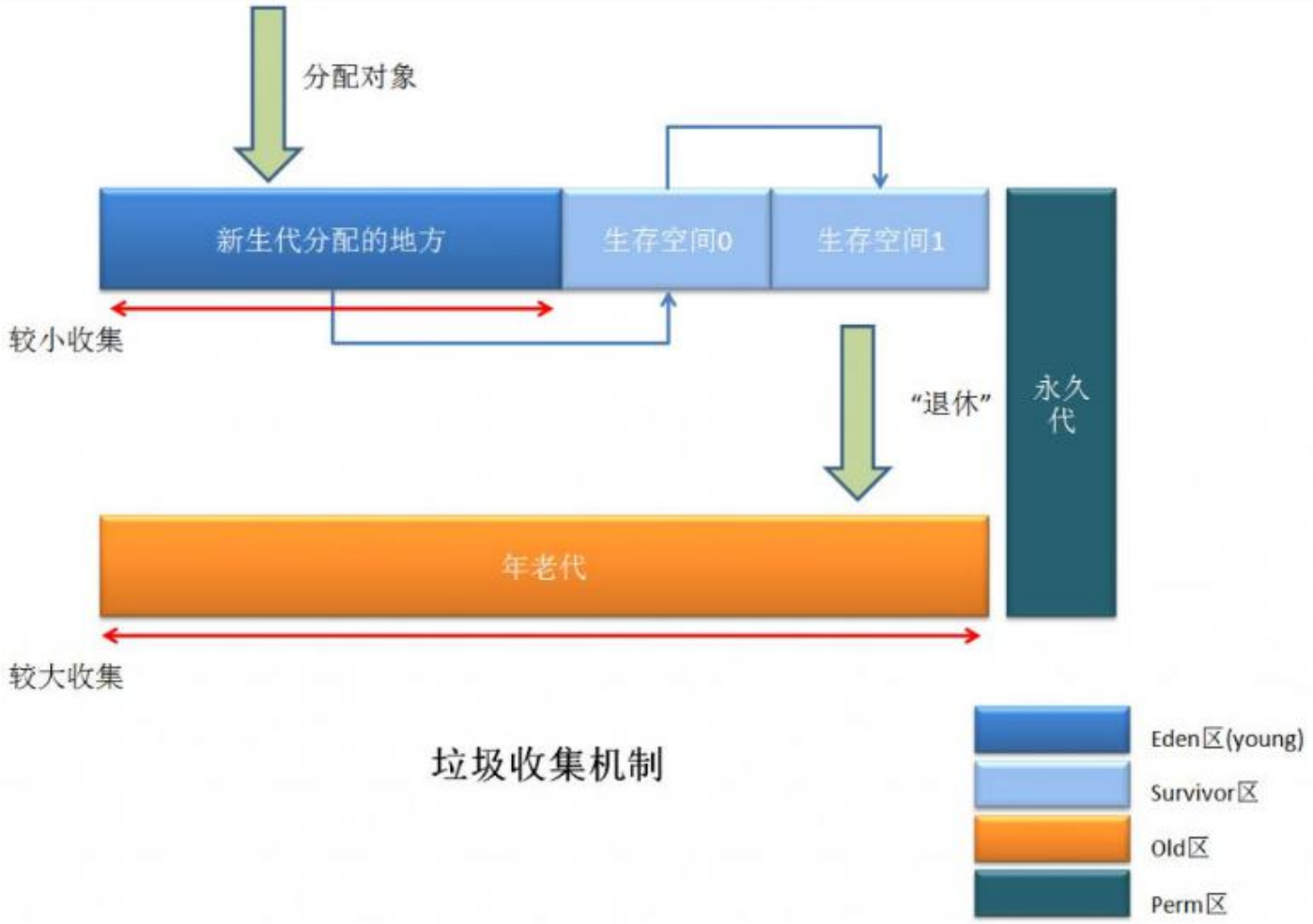
1. GC 是什么

JVM 垃圾收集(Java Garbage Collection)

GC 采用分代收集算法：

- 次数上频繁收集 Young 区
- 次数上较少收集 Old 区
- 基本不动 Perm 区

2. GC 算法总体概述



JVM 在进行 GC 时，并非每次都对上面三个内存区域一起回收的，大部分时候回收的都是指新生代。

因此 GC 按照回收的区域又分了两种类型，一种是普通 GC（minor GC），一种是全局 GC（major GC or Full GC），

普通 GC（minor GC）：只针对新生代区域的 GC。

全局 GC（major GC or Full GC）：针对年老代的 GC，偶尔伴随对新生代的 GC 以及对永久代的 GC。

3. 引用计数法

4. 复制算法(Copying)

年轻代中使用的是 Minor GC，这种 GC 算法采用的是复制算法(Copying)。

（1） 原理

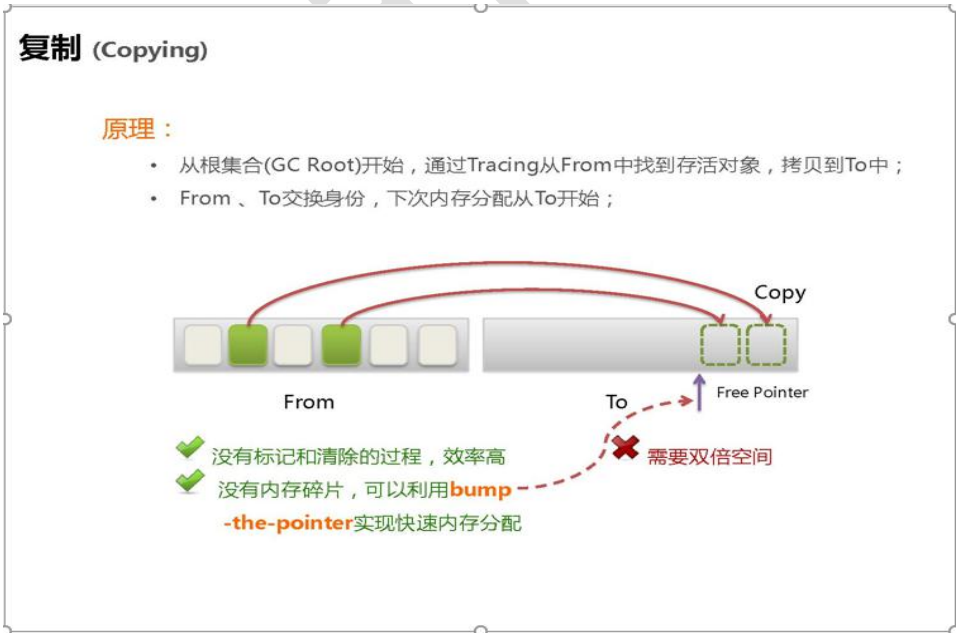
Minor GC 会把 Eden 中的所有活的对象都移到 Survivor 区域中，如果 Survivor 区中放不下，那么剩下的活的对象就被移到 Old generation 中，也即一旦收集后，Eden 是就变成空的了。

当对象在 Eden (包括一个 Survivor 区域，这里假设是 from 区域) 出生后，在经过一次 Minor GC 后，如果对象还存活，并且能够被另外一块 Survivor 区域所容纳(上面已经假设为 from 区域，这里应为 to 区域，即 to 区域有足够的内存空间来存储 Eden 和 from 区域中存活的对象)，则使用复制算法将这些仍然还存活的对象复制到另外一块 Survivor 区域 (即 to 区域) 中，然后清理所使用过的 Eden 以及 Survivor 区域 (即 from 区域)，并且将这些对象的年龄设置为 1，以后对象在 Survivor 区每熬过一次 Minor GC，就将对象的年龄 + 1，当对象的年龄达到某个值时 (默认是 15 岁，通过-XX:MaxTenuringThreshold 来设定参数)，这些对象就会成为老年代。

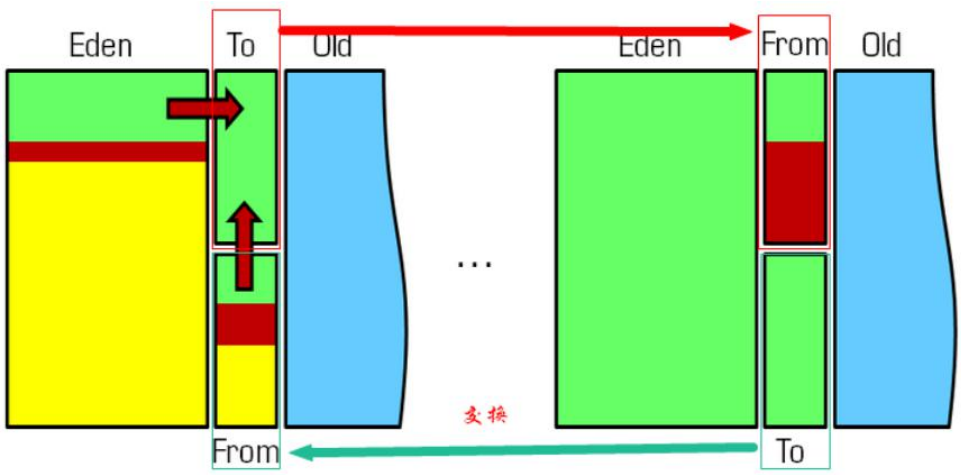
（2） 算法分析

年轻代中的 GC,主要是复制算法（Copying）

HotSpot JVM 把年轻代分为了三部分：1 个 Eden 区和 2 个 Survivor 区（分别叫 from 和 to）。默认比例为 8:1:1,一般情况下，新创建的对象都会被分配到 Eden 区(一些大对象特殊处理),这些对象经过第一次 Minor GC 后，如果仍然存活，将会被移到 Survivor 区。对象在 Survivor 区中每熬过一次 Minor GC，年龄就会增加 1 岁，当它的年龄增加到一定程度时，就会被移动到年老代中。因为年轻代中的对象基本都是朝生夕死的(90%以上)，所以在年轻代的垃圾回收算法使用的是复制算法，复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片。



在 GC 开始的时候，对象只会存在于 Eden 区和名为“From”的 Survivor 区，Survivor 区“To”是空的。紧接着进行 GC，Eden 区中所有存活的对象都会被复制到“To”，而在“From”区中，仍存活的对象会根据他们的年龄值来决定去向。年龄达到一定值(年龄阈值，可以通过-XX:MaxTenuringThreshold 来设置)的对象会被移动到年老代中，没有达到阈值的对象会被复制到“To”区域。经过这次 GC 后，Eden 区和 From 区已经被清空。这个时候，“From”和“To”会交换他们的角色，也就是新的“To”就是上次 GC 前的“From”，新的“From”就是上次 GC 前的“To”。不管怎样，都会保证名为 To 的 Survivor 区域是空的。Minor GC 会一直重复这样的过程，直到“To”区被填满，“To”区被填满之后，会将所有对象移动到年老代中。



因为 Eden 区对象一般存活率较低，一般的，使用两块 10%的内存作为空闲和活动区间，而另外 80%的内存，则是用来给新建对象分配内存的。一旦发生 GC，将 10%的 from 活动区间与另外 80%中存活的 eden 对象转移到 10%的 to 空闲区间，接下来，将之前 90%的内存全部释放，以此类推。

（3） 劣势

复制算法它的缺点也是相当明显的。

1、它浪费了一半的内存，这太要命了。

2、如果对象的存活率很高，我们可以极端一点，假设是 100%存活，那么我们需要将所有对象都复制一遍，并将所有引用地址重置一遍。复制这一工作所花费的时间，在对象存活率达到一定程度时，将会变的不可忽视。所以从以上描述不难看出，复制算法要想使用，最起码对象的存活率要非常低才行，而且最重要的是，我们必须要克服 50%内存的浪费。

5. 标记清除(Mark-Sweep)

老年代一般是由标记清除或者是标记清除与标记整理的混合实现

（1） 原理

当堆中的有效内存空间（available memory）被耗尽的时候，就会停止整个程序（也被称为 stop the world），然后进行两项工作，第一项则是标记，第二项则是清除。

标记：从引用根节点开始标记所有被引用的对象。标记的过程其实就是遍历所有的 GC Roots，然后将所有 GC Roots 可达的对象　　标记为存活的对象。

清除：遍历整个堆，把未标记的对象清除。

缺点：此算法需要暂停整个应用，会产生内存碎片

用通俗的话解释一下标记/清除算法，就是当程序运行期间，若可以使用的内存被耗尽的时候，GC 线程就会被

触发并将程序暂停，随后将依旧存活的对象标记一遍，最终再将堆中所有没被标记的对象全部清除掉，接下来便让程序恢复运行。

（2） 劣势

1、首先，它的缺点就是效率比较低（递归与全堆对象遍历），而且在进行 GC 的时候，需要停止应用程序，这会导致用户体验非常差劲

2、其次，主要的缺点则是这种方式清理出来的空闲内存是不连续的，这点不难理解，我们的死亡对象都是随即的出现在内存的各个角落的，现在把它们清除之后，内存的布局自然会乱七八糟。而为了应付这一点，JVM 就不得不维持一个内存的空闲列表，这又是一种开销。而且在分配数组对象的时候，寻找连续的内存空间会不太好找。

6. 标记清除(Mark-Sweep)

老年代一般是由标记清除或者是标记清除与标记整理的混合实现

（1） 原理

在整理压缩阶段，不再对标记的对像做回收，而是通过所有存活对像都向一端移动，然后直接清除边界以外的内存。

可以看到，标记的存活对象将会被整理，按照内存地址依次排列，而未被标记的内存会被清理掉。如此一来，当我们需要给新对象分配内存时，JVM 只需要持有一个内存的起始地址即可，这比维护一个空闲列表显然少了许多开销。

标记/整理算法不仅可以弥补标记/清除算法当中，内存区域分散的缺点，也消除了复制算法当中，内存减半的高额代价

（2） 劣势

标记/整理算法唯一的缺点就是效率也不高，不仅要标记所有存活对象，还要整理所有存活对象的引用地址。从效率上来说，标记/整理算法要低于复制算法。

7. 标记清除压缩(Mark-Sweep-Compact)

标记清除、标记压缩的结合使用

（1） 原理

标记-清除-压缩 (Mark-Sweep-Compact)

原理：

1. Mark-Sweep 和 Mark-Compact的结合。
2. 和Mark-Sweep一致，当进行多次GC后才Compact。

减少移动对象的成本

8. 算法总结

内存效率：复制算法>标记清除算法>标记整理算法（此处的效率只是简单的对比时间复杂度，实际情况不一定如此）。

内存整齐度：复制算法=标记整理算法>标记清除算法。

内存利用率：标记整理算法=标记清除算法>复制算法。

可以看出，效率上来说，复制算法是当之无愧的老大，但是却浪费了太多内存，而为了尽量兼顾上面所提到的三个指标，标记/整理算法相对来说更平滑一些，但效率上依然不尽如人意，它比复制算法多了一个标记的阶段，又比标记/清除多了一个整理内存的过程

难道就没有一种最优算法吗？

回答：无，没有最好的算法，只有最合适的算法。

分代收集算法。

（1） 年轻代(Young Gen)

年轻代特点是区域相对老年代较小，对像存活率低。

这种情况复制算法的回收整理，速度是最快的。复制算法的效率只和当前存活对像大小有关，因而很适用于年轻代的回收。而复制算法内存利用率不高的问题，通过 hotspot 中的两个 survivor 的设计得到缓解。

（2） 老年代(Tenure Gen)

老年代的特点是区域较大，对像存活率高。

这种情况，存在大量存活率高的对像，复制算法明显变得不合适。一般是由标记清除或者是标记清除与标记整理的混合实现。

Mark 阶段的开销与存活对像的数量成正比，这点上说来，对于老年代，标记清除或者标记整理有一些不符，但可以通过多核/线程利用，对并发、并行的形式提标记效率。

Sweep 阶段的开销与所管理区域的大小形正相关，但 Sweep“就地处决”的特点，回收的过程没有对像的移动。使其相对其它有对像移动步骤的回收算法，仍然是效率最好的。但是需要解决内存碎片问题。

Compact 阶段的开销与存活对像的数据成开比，如上一条所描述，对于大量对像的移动是很大开销的，做为老年代的第一选择并不合适。

基于上面的考虑，老年代一般是由标记清除或者是标记清除与标记整理的混合实现。以 hotspot 中的 CMS 回收器为例，CMS 是基于 Mark-Sweep 实现的，对于对像的回收效率很高，而对于碎片问题，CMS 采用基于 Mark-Compact 算法的 Serial Old 回收器做为补偿措施：当内存回收不佳（碎片导致的 Concurrent Mode Failure 时），将采用 Serial Old 执行 Full GC 以达到对老年代内存的整理。

9. 练习

（1）JVM 内存模型以及分区，需要详细到每个区放什么？

（2）堆里面的分区：Eden，survival from to，老年代，各自的特点？

（3）GC 的三种收集方法：标记清除、标记整理、复制算法的原理与特点，分别用在什么地方？

（4）Minor GC 与 Full GC 分别在什么时候发生？