慢慢琢磨 JVM

1 JVM 简介

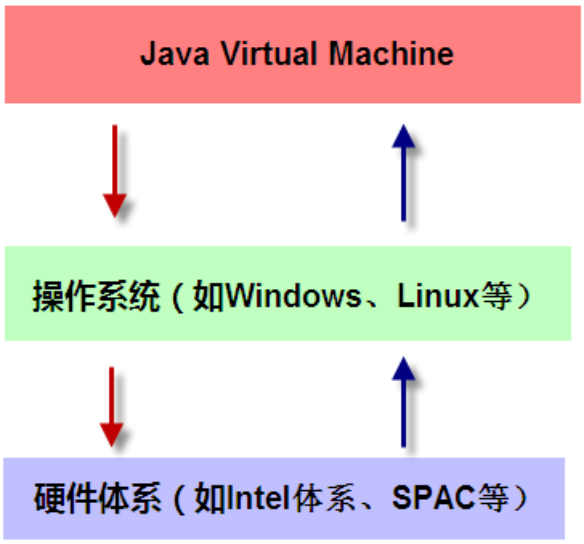
JVM 是我们 Javaer 的最基本功底了，刚开始学 Java 的时候，一般都是从“Hello World”开始的，然后会写个复杂点 class，然后再找一些开源框架，比如 Spring，Hibernate 等等，再然后就开发企业级的应用，比如网站、企业内部应用、实时交易系统等等，直到某一天突然发现做的系统咋就这么慢呢，而且时不时还来个内存溢出什么的，今天是交易系统报了 StackOverflowError，明天是网站系统报了个 OutOfMemoryError，这种错误又很难重现，只有分析 Javacore 和 dump 文件，运气好点还能分析出个结果，运行遭的点，就直接去庙里烧香吧！每天接客户的电话都是战战兢兢的，生怕再出什么幺蛾子了。我想 Java 做的久一点的都有这样的经历，那这些问题的最终根结是在哪呢？—— JVM。

JVM 全称是 Java Virtual Machine，Java 虚拟机，也就是在计算机上再虚拟一个计算机，这和我们使用 VMWare 不一样，那个虚拟的东西你是可以看到的，这个 JVM 你是看不到的，它存在内存中。我们知道计算机的基本构成是：运算器、控制器、存储器、输入和输出设备，那这个 JVM 也是有这成套的元素，运算器是当然是交给硬件 CPU 还处理了，只是为了适应“一次编译，随处运行”的情况，需要做一个翻译动作，于是就用了 JVM 自己的命令集，这与汇编的命令集有点类似，每一种汇编命令集针对一个系列的 CPU，比如 8086 系列的汇编也是可以用在 8088 上的，但是就不能跑在 8051 上，而 JVM 的命令集则是可以到处运行的，因为 JVM 做了翻译，根据不同的 CPU，翻译成不同的机器语言。

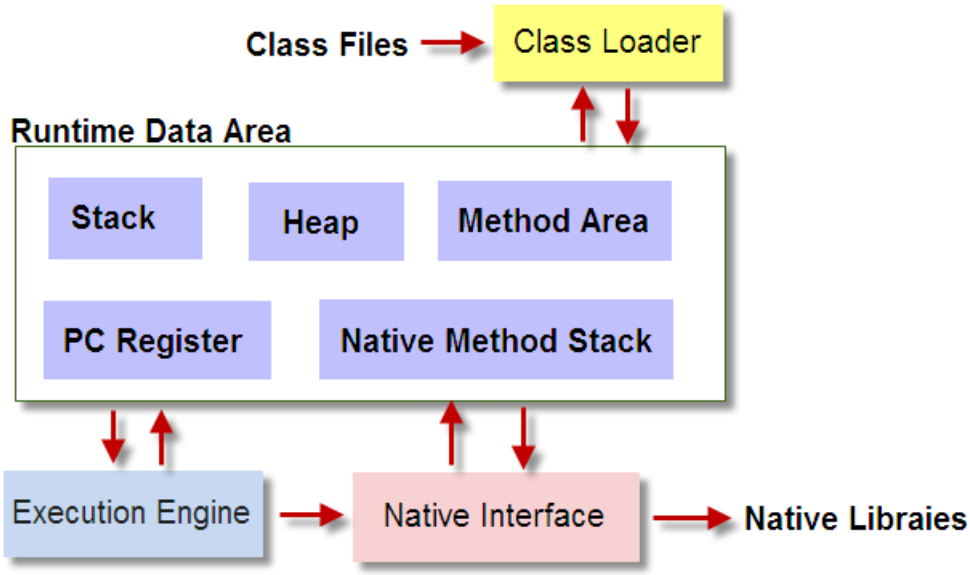
JVM 中我们最需要深入理解的就是它的存储部分，存储？硬盘？NO，NO， JVM 是一个内存中的虚拟机，那它的存储就是内存了，我们写的所有类、常量、变量、方法都在内存中，这决定着我们程序运行的是否健壮、是否高效，接下来的部分就是重点介绍之。

2 JVM 的组成部分

我们先把 JVM 这个虚拟机画出来，如下图所示：



从这个图中可以看到，JVM 是运行在操作系统之上的，它与硬件没有直接的交互。我们再来看下 JVM 有哪些组成部分，如下图所示：



该图参考了网上广为流传的 JVM 构成图，大家看这个图，整个 JVM 分为四部分：

- Class Loader 类加载器

类加载器的作用是加载类文件到内存，比如编写一个 HelloWord.java 程序，然后通过 javac 编译成 class 文件，那怎么才能加载到内存中被执行呢？Class Loader 承担的就是这个责任，那不可能随便建立一个.class 文件就能被加载的，Class Loader 加载的 class 文件是有格式要求，在《JVM Specification》中式这样定义 Class 文件的结构：

```
ClassFile {
  u4 magic;
  u2 minor_version;
```

```
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
  }
```

需要详细了解的话，可以仔细阅读《JVM Specification》的第四章“The class File Format”，这里不再详细说明。

友情提示：Class Loader 只管加载，只要符合文件结构就加载，至于说能不能运行，则不是它负责的，那是由 Execution Engine 负责的。

- Execution Engine 执行引擎

执行引擎也叫做解释器(Interpreter)，负责解释命令，提交操作系统执行。

- Native Interface 本地接口

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，必须有一个聪明的、睿智的调用 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。目前该方法使用的是越来越少了，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。

- Runtime data area 运行数据区

运行数据区是整个 JVM 的重点。我们所有写的程序都被加载到这里，之后才开始运行，Java 生态系统如此的繁荣，得益于该区域的优良自治，下一章节详细介绍之。

整个 JVM 框架由加载器加载文件，然后执行器在内存中处理数据，需要与异构系统交互是可以

通过本地接口进行，瞧，一个完整的系统诞生了！

2 JVM 的内存管理

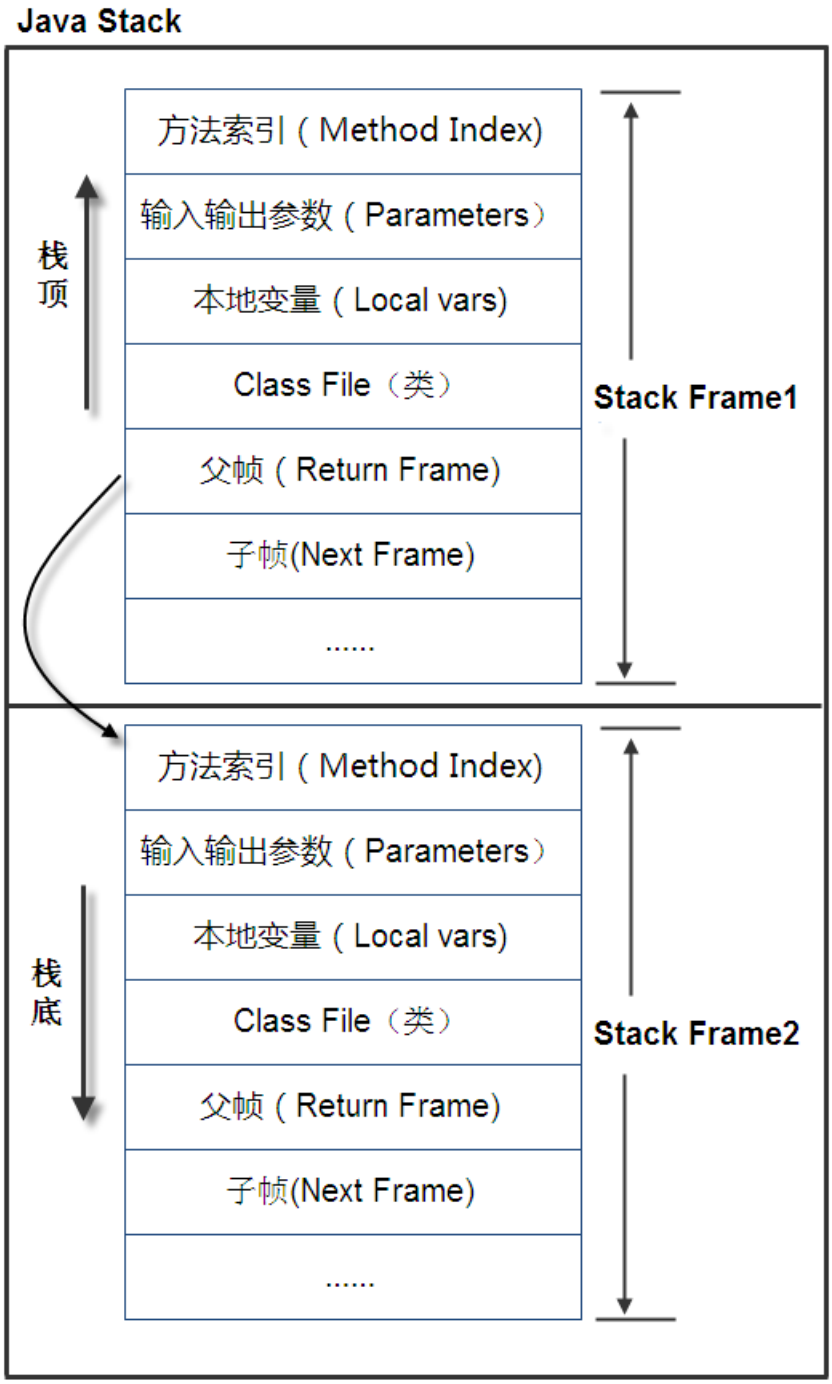
所有的数据和程序都是在运行数据区存放，它包括以下几部分：

- Stack 栈

栈也叫栈内存，是 Java 程序的运行区，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。问题出来了：栈中存的是那些数据呢？又什么是格式呢？

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，执行完毕后，先弹出 F2 栈帧，再弹出 F1 栈帧，遵循“先进后出”原则。

那栈帧中到底存在着什么数据呢？栈帧中主要保存 3 类数据：本地变量（Local Variables），包括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。光说比较枯燥，我们画个图来理解一下 Java 栈，如下图所示：



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。

- Heap 堆内存

一个 JVM 实例只存在一个堆类存，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，以方便执行器执行，堆内存分为三部分：

Permanent Space 永久存储区

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。

Young Generation Space 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）和幸存者区（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，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。

Tenure generation space 养老区

养老区用于保存从新生区筛选出来的 JAVA 对象，一般池对象都在这个区域活跃。 三个区的示意图如下：

- Method Area 方法区

方法区是被所有线程共享,该区域保存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。

- PC Register 程序计数器

每个线程都有一个程序计数器，就是一个指针，指向方法区中的方法字节码，由执行引擎读取下一条指令。

❑ Native Method Stack 本地方法栈

以上都是纯理论，我们举个例子来说明 JVM 的运行原理，我们来写一个简单的类，代码如下：

```
public class JVMShowcase {
    //静态类常量,
    public final static String ClASS_CONST = "I'm a Const";
    //私有实例变量
    private int instanceVar=15;
    public static void main(String[] args) {
        //调用静态方法
        runStaticMethod();
        //调用非静态方法
        JVMShowcase showcase=new JVMShowcase();
        showcase.runNonStaticMethod(100);
    }
    //常规静态方法
    public static String runStaticMethod(){
        return ClASS_CONST;
    }
    //非静态方法
    public int runNonStaticMethod(int parameter){
        int methodVar=this.instanceVar * parameter;
        return methodVar;
    }
}
```

这个类没有任何意义，不用猜测这个类是做什么用，只是写一个比较典型的类，然后我们来看看 JVM 是如何运行的，也就是输入 java JVMShow 后，我们来看 JVM 是如何处理的：

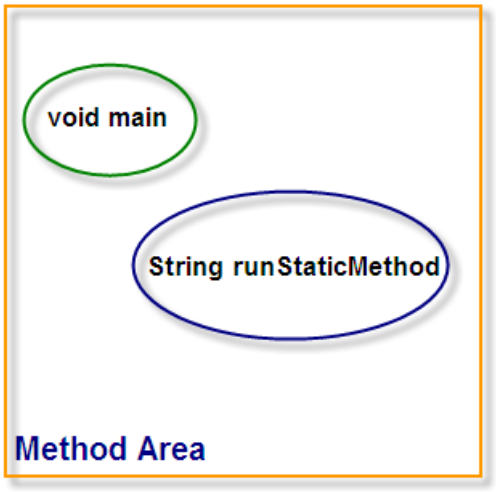
第 1 步，JVM 向操作系统申请空闲内存。JVM 对操作系统说“给我 64M 空闲内存”，于是操作系统就查找自己的内存分配表，找了段 64M 的内存写上“Java 占用”标签，然后把内存段的起

始地址和终止地址给 JVM，JVM 准备加载类文件。

第 2 步，JVM 分配内存。JVM 获得到 64M 内存，就开始得瑟了，首先给 heap 分个内存，并且是按照 heap 的三种不同类型分好的，然后给栈内存也分配好。

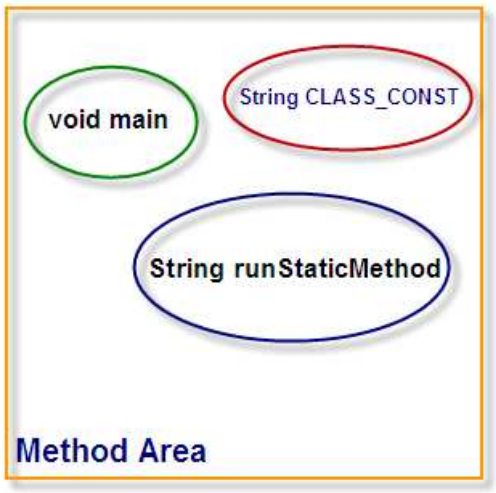
第 3 步，检查和分析 class 文件。若发现有错误即返回错误。

第 4 步，加载类。由于没有指定加载器，JVM 默认使用 bootstrap 加载器，就把 rt.jar 下的所有类都加载到了堆类存的永久存储区，JVMShow 也被加载到内存中。我们来看看栈内存，如下图：



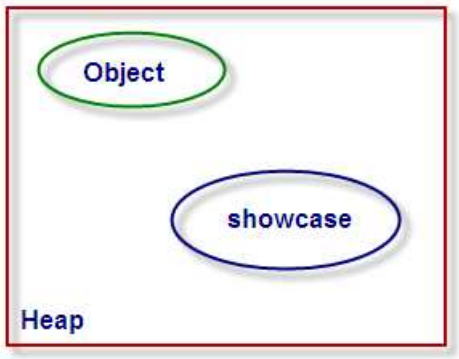
Heap 是空，Stack 是空，因为还没有线程被执行。Class Loader 通知 Execution Enginer 已经加载完毕。

第 5 步，执行引擎执行 main 方法。执行引擎启动一个线程，开始执行 main 方法，在 main 执行完毕前，方法区如下图所示：

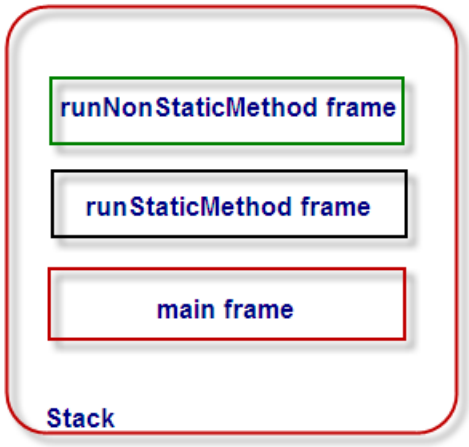


在 Method Area 加入了 CLASS_CONST 常量，它是在第一次被访问时产生的。堆内存中有两

个对象 object 和 showcase 对象，如下图所示：



为什么会有 Object 对象呢？是因为它是 JVMShowcase 的父类，JVM 是先初始化父类，然后再初始化子类，甭管有多少个父类都初始化。在栈内存中有三个栈帧，如下图所示：



于此同时，还创建了一个程序计数器指向下一条要执行的语句。

第 6 步，释放内存。运行结束，JVM 向操作系统发送消息，说“内存用完了，我还给你”，运行结束。

3 JVM 相关问题

问：堆和栈有什么区别

答：堆是存放对象的，但是对象内的临时变量是存在栈内存中，如例子中的 methodVar 是在运行期存放到栈中的。

栈是跟随线程的，有线程就有栈，堆是跟随 JVM 的，有 JVM 就有堆内存。

问：堆内存中到底存在着什么东西？

答：对象，包括对象变量以及对象方法。

问：类变量和实例变量有什么区别？

答：静态变量是类变量，非静态变量是实例变量，直白的说，有 static 修饰的变量是静态变量，没有 static 修饰的变量是实例变量。静态变量存在方法区中，实例变量存在堆内存中。

问：我听说类变量是在 JVM 启动时就初始化好的，和你这说的不同呀！

答：那你是道听途说，信我的，没错。

问：Java 的方法（函数）到底是传值还是传址？

答：都不是，是以传值的方式传递地址，具体的说原生数据类型传递的值，引用类型传递的地址。对于原始数据类型，JVM 的处理方法是从 Method Area 或 Heap 中拷贝到 Stack，然后运行 frame 中的方法，运行完毕后再把变量指拷贝回去。

问：为什么会产生 OutOfMemory 产生？

答：一句话：Heap 内存中没有足够的可用内存了。这句话要好好理解，不是说 Heap 没有内存了，是说新申请内存的对象大于 Heap 空闲内存，比如现在 Heap 还空闲 1M，但是新申请的内存需要 1.1M，于是就会报 OutOfMemory 了，可能以后的对象申请的内存都只要 0.9M，于是就只出现一次 OutOfMemory，GC 也正常了，看起来像偶发事件，就是这么回事。但如果此时 GC 没有回收就会产生挂起情况，系统不响应了。

问：我产生的对象不多呀，为什么还会产生 OutOfMemory？

答：你继承层次忒多了，Heap 中 产生的对象是先产生 父类，然后才产生子类，明白不？

问：OutOfMemory 错误分几种？

答：分两种，分别是“OutOfMemoryError:java heap size”和”OutOfMemoryError: PermGen space”，两种都是内存溢出，heap size 是说申请不到新的内存了，这个很常见，检查应用或调整堆内存大小。

“PermGen space”是因为永久存储区满了，这个也很常见，一般在热发布的环境中出现，是因为每次发布应用系统都不重启，久而久之永久存储区中的死对象太多导致新对象无法申请内存，

一般重新启动一下即可。

问：为什么会产生 StackOverflowError？

答：因为一个线程把 Stack 内存全部耗尽了，一般是递归函数造成的。

问：一个机器上可以看多个 JVM 吗？JVM 之间可以互访吗？

答：可以多个 JVM，只要机器承受得了。JVM 之间是不可以互访，你不能在 A-JVM 中访问 B-JVM 的 Heap 内存，这是不可能的。在以前老版本的 JVM 中，会出现 A-JVM Crack 后影响到 B-JVM，现在版本非常少见。

问：为什么 Java 要采用垃圾回收机制，而不采用 C/C++的显式内存管理？

答：为了简单，内存管理不是每个程序员都能折腾好的。

问：为什么你没有详细介绍垃圾回收机制？

答：垃圾回收机制每个 JVM 都不同，JVM Specification 只是定义了要自动释放内存，也就是说它只定义了垃圾回收的抽象方法，具体怎么实现各个厂商都不同，算法各异，这东西实在没必要深入。

问：JVM 中到底哪些区域是共享的？哪些是私有的？

答：Heap 和 Method Area 是共享的，其他都是私有的，

问：什么是 JIT，你怎么没说？

答：JIT 是指 Just In Time，有的文档把 JIT 作为 JVM 的一个部件来介绍，有的是作为执行引擎的一部分来介绍，这都能理解。Java 刚诞生的时候是一个解释性语言，别嘘，即使编译成了字节码（byte code）也是针对 JVM 的，它需要再次翻译成原生代码(native code)才能被机器执行，于是效率的担忧就提出来了。Sun 为了解决该问题提出了一套新的机制，好，你想编译成原生代码，没问题，我在 JVM 上提供一个工具，把字节码编译成原生码，下次你来访问的时候直接访问原生码就成了，于是 JIT 就诞生了，就这么回事。

问：JVM 还有哪些部分是你没有提到的？

答：JVM 是一个异常复杂的东西，写一本砖头书都不为过，还有几个要说明的：

常量池（constant pool）：按照顺序存放程序中的常量，并且进行索引编号的区域。比如 int i =100，这个 100 就放在常量池中。

安全管理器（Security Manager）：提供 Java 运行期的安全控制，防止恶意攻击，比如指定读取文件，写入文件权限，网络访问，创建进程等等，Class Loader 在 Security Manager 认证通过后才能加载 class 文件的。

方法索引表（Methods table），记录的是每个 method 的地址信息，Stack 和 Heap 中的地址指针其实是指向 Methods table 地址。

问：为什么不建议在程序中显式的生命 System.gc()？

答：因为显式声明是做堆内存全扫描，也就是 Full GC，是需要停止所有的活动的（Stop The World Collection），你的应用能承受这个吗？

问：JVM 有哪些调整参数？

答：非常多，自己去找，堆内存、栈内存的大小都可以定义，甚至是堆内存的三个部分、新生代的各个比例都能调整。