No Exploration， No Soul !

JVM Further Introduction

本文详细讲解了 JVM(Java Virtual Machine)的方方面面，首先由 java 的特性来描绘 JVM 的大致应用，再细细阐述了 JVM 的原理及内存管理机制和调优，讲述了与 JVM 密切相关的 Java GC 机制，最后对 JVM 调优进行了总结。

本文内容是本人对网络内容的总结，是学习 JVM 的好资料。

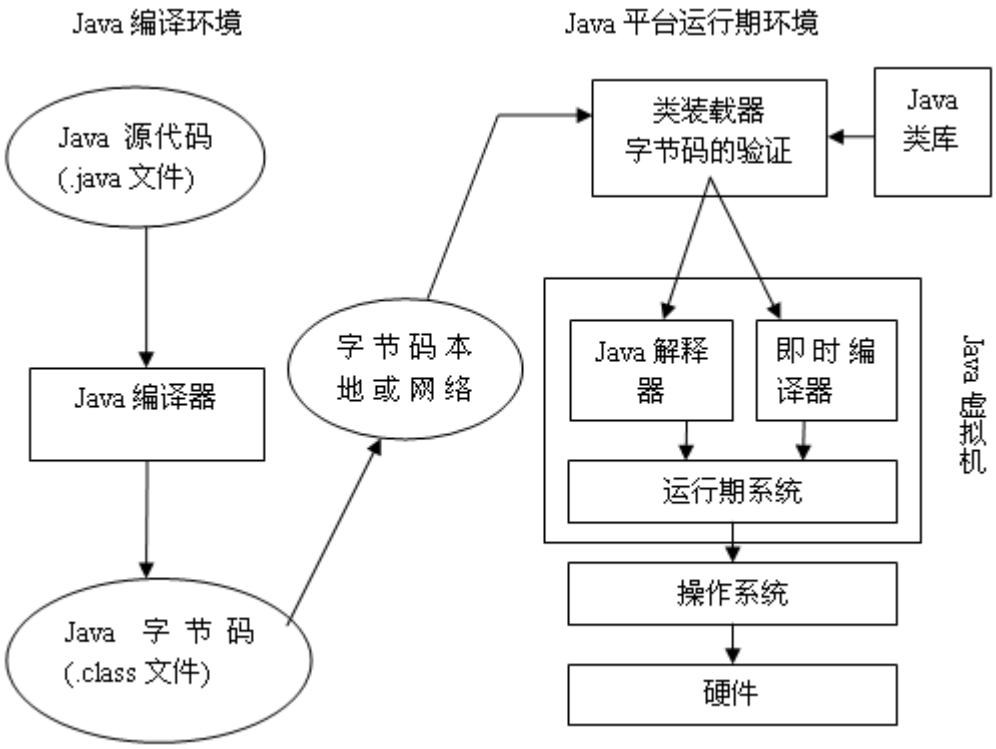
1. 目录

2. Java 技术

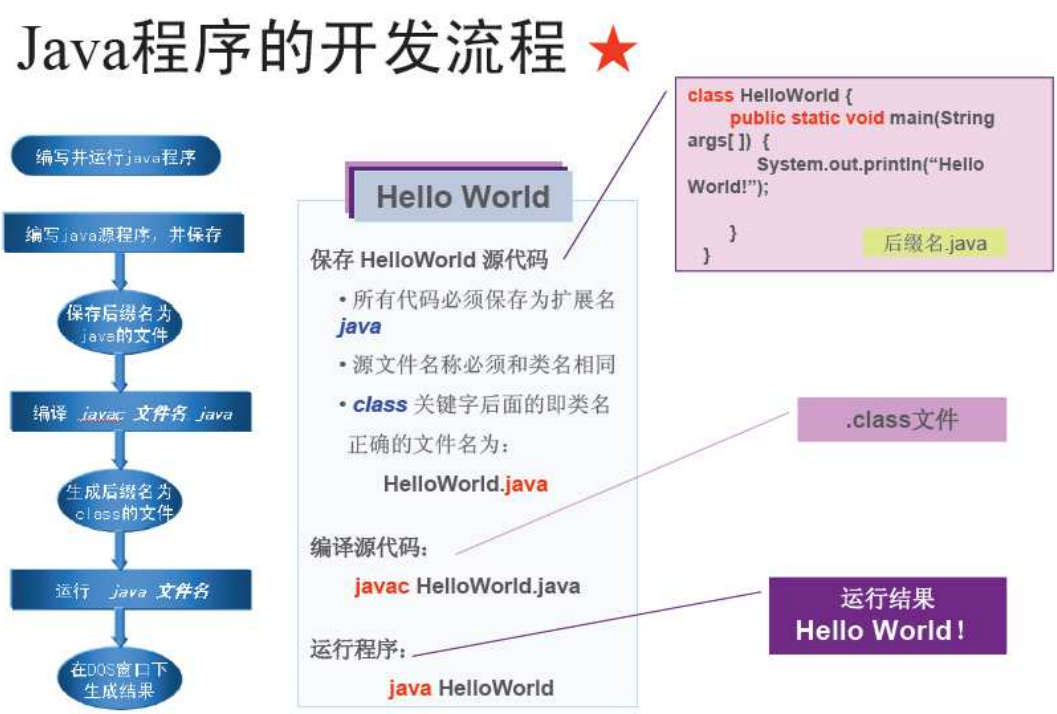
2.1. Java 定义

一种简单、面向对象、分布式、跨平台、半编译半解释、健壮、安全、高性能、多线程的动态的语言 —— Sun定义

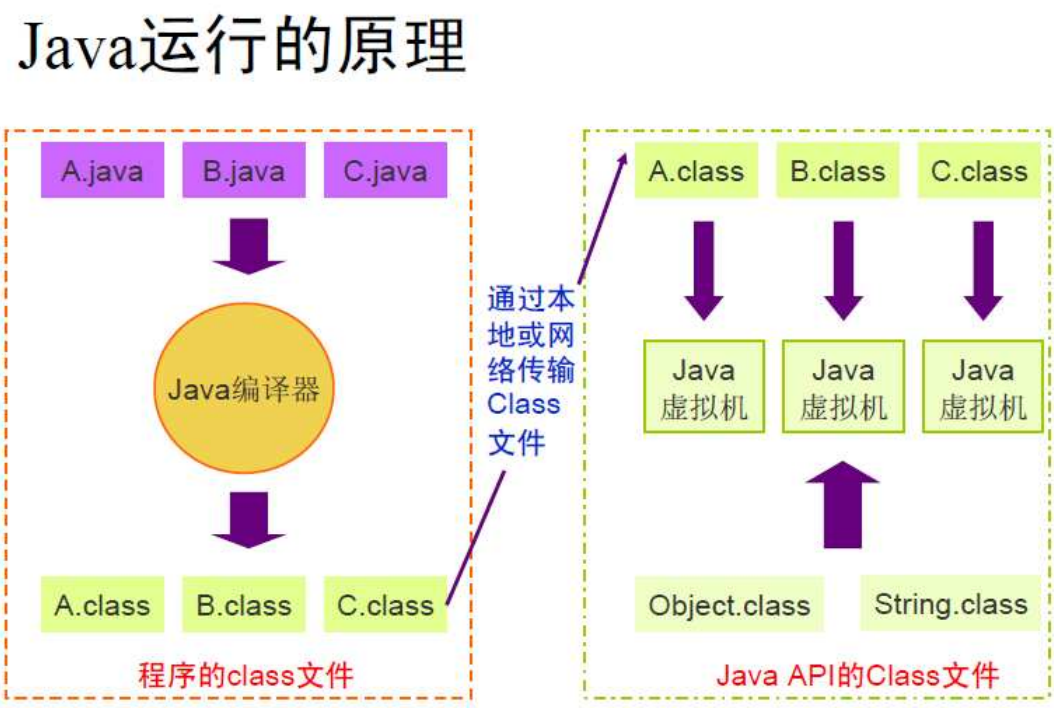
Java 是一门技术，它由四方面组成：Java 编程语言、Java 类文件格式、Java 虚拟机和 Java 应用程序接口(Java API)。它们的关系如下图所示：



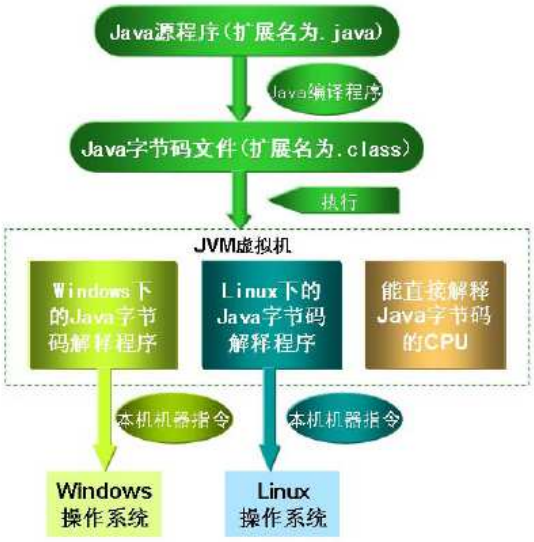
2.2. Java 的开发流程



2.3. Java 的运行原理



Java运行的原理

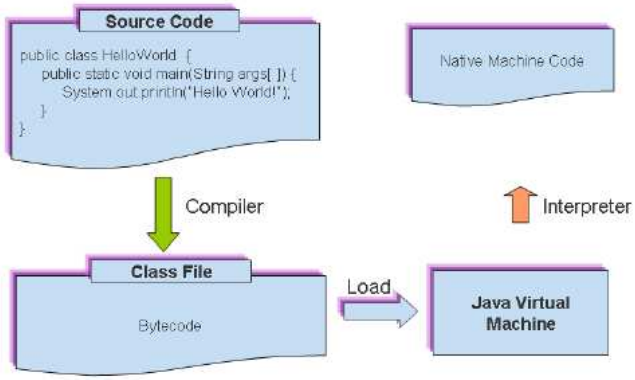


- **JVM (Java Virtual Machine) — Java虚拟机**
  - ✓ 一个虚构出来的计算机
  - ✓ 通过在实际的计算机上仿真模拟各种计算机功能来实现的。
  - ✓ Java虚拟机有自己完善的硬件架构,如处理器、堆栈、寄存器等,还具有相应的指令系统。

- 开发人员编写 Java 代码（.java 文件）。
- 然后将其编译成字节码（.class 文件）。
- 最后字节码被装入内存，一旦字节码进入虚拟机中，它就会被解释器解释执行，或者是被即时编译器有选择的转换成机器码执行。

2.4. 半编译半解释

- **半编译半解释？**
  - ✓ 系统先将用户输入的指令翻译成一种通用的，比较规则的中间形式的代码，保密性强，运行时则由所在机器的解释器进行解释
  - ✓ java 语言的开发效率高，但执行效率低。（相当于c++的55%）

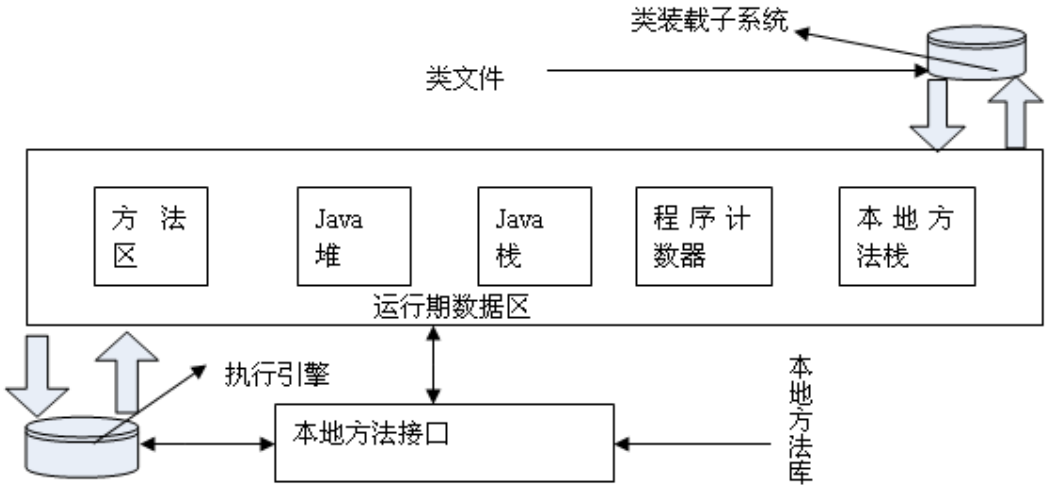


2.5. 平台无关性

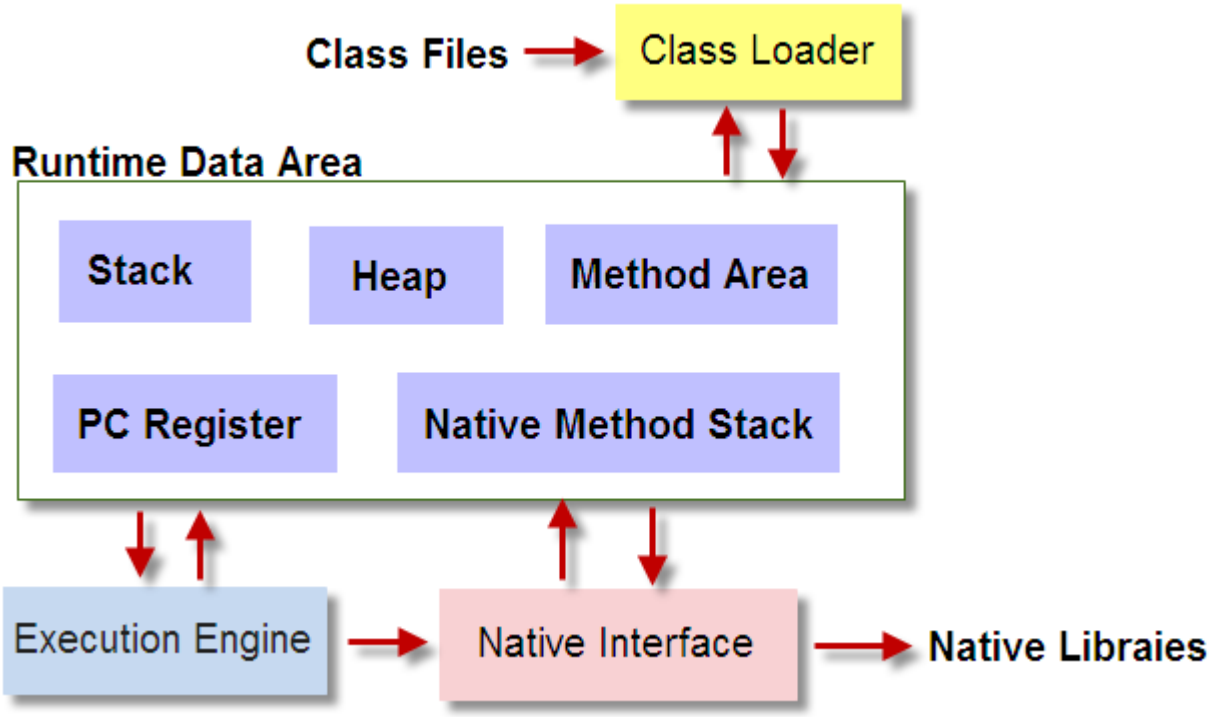
- 平台无关性？
  - ✓ 何谓平台：即一套特定的硬件再加上运行其上的操作系统，即硬件＋软件。编程语言对不同平台的支持有所不同。（VB、C/C++ 、Java）
  - ✓ Java完全不用修改任何源代码，也不用重新编译就可以直接移植到其他平台。
  - ✓ Java的平台无关性给程序的部署带来了很大的灵活性，节约开发和升级成本。
  - ✓ 怎样理解平台无关性呢？JVM（java Virtual Machine)起到了主要作用。JVM是运行在平台之上的程序，它能够虚拟出一台目标机，所有字节码就是在虚拟出的目标机上运行。
  - ✓ 程序不可能在所有的平台上都可以运行：（1）因为不同平台的内存管理模式和CPU的指令集等都有很大的差别。（2）为了让java实现平台无关性，Sun公司在不同平台上用软件模拟出虚拟目标机，虚拟出CPU指令集和内存。（3）因此虽然平台间的差异比较大，但是虚拟出来的JVM是完全一样的。（4）Java的字节码仅仅运行在JVM上，不会和平台的底层直接打交道。（5）JVM根据平台的不同，把字节码解释成不同的本地代码（6）JVM就像翻译，把通用的普通话翻译成不同地方特色的方言。
  - ✓ 但是有一个缺点：java代码必须要经过JVM解释才能运行，使得java运行的效率降低。
  - ✓ WORA：Write Once, Run Anywhere(一次编写,到处运行).

3. JVM 的体系结构

从上可以看出，Java 虚拟机（JVM）处在核心的位置，是程序与底层操作系统和硬件无关的关键。



对应的英文结构图



JVM 体系结构主要包括两个子系统和两个组件：

- 类装载（Class Loader）子系统。
- 执行引擎（Execution Engine）子系统。
- 运行数据域（Runtime Data Area）组件。
- 本地接口（Native Interface）组件。

3.1. 类装载（Class Loader）子系统

根据给定的全限定名类名(如 java.lang.Object)来装载 .class 文件的内容到 Runtime data area 中的 method area(方法区域)。程序员可以 extends java.lang.ClassLoader 类来写自己的 Class loader。

3.2. 执行引擎（Execution Engine）子系统

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。

3.3. 本地接口（Native Interface）组件

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 libraries。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。

3.4. 运行数据域（Runtime Data Area）组件

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。接下来我们深入探讨。

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：

3.4.1. Java 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。

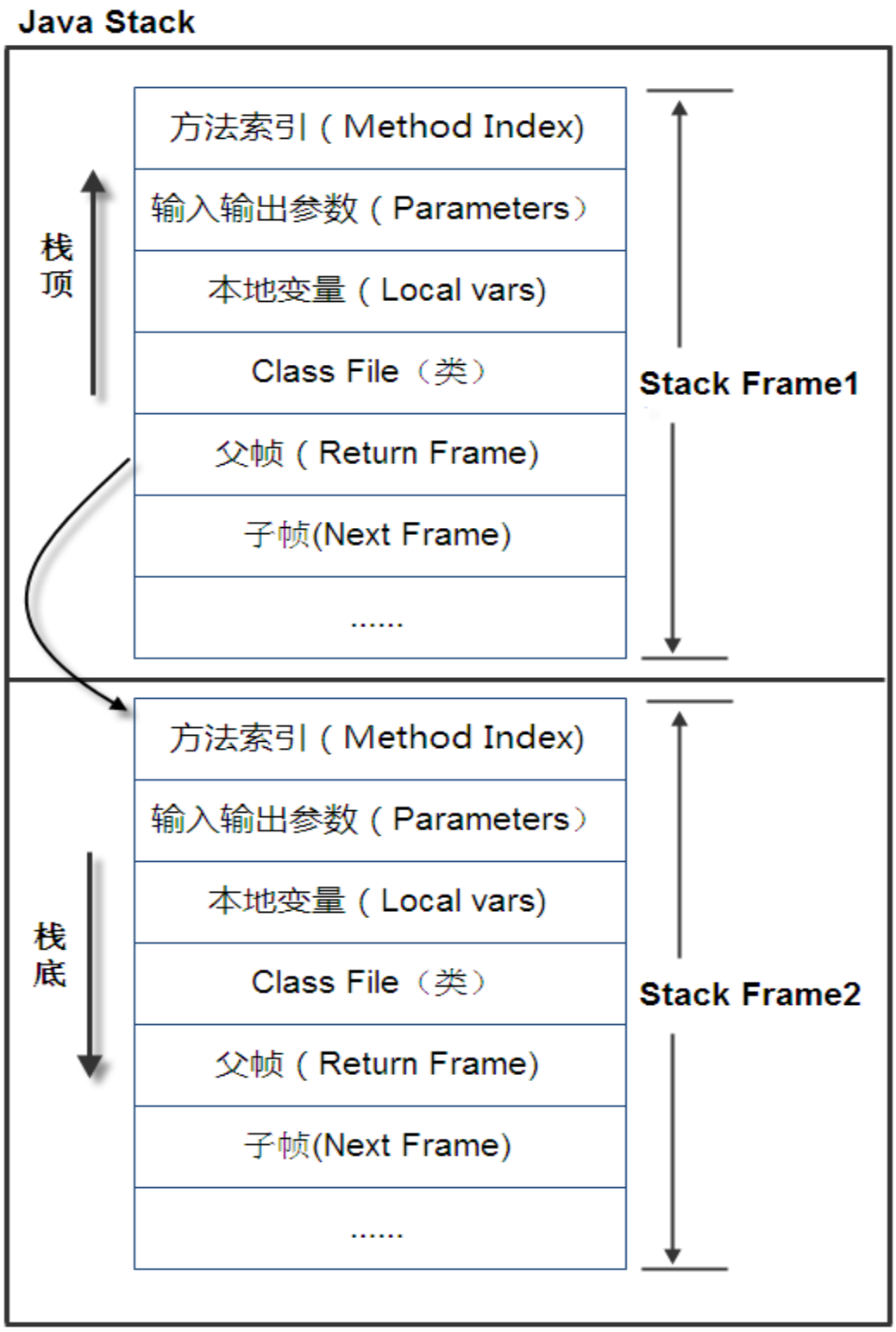
问题出来了：栈中存的是哪些数据呢？又什么是格式呢？

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。

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。

3.4.2. Java Heap（堆）

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：

3.4.3. 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。

3.4.4. 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。

3.4.5. Native Method Stack（本地方法栈）

本地方法栈与 VM 栈所发挥作用是类似的，只不过 VM 栈为虚拟机运行 VM 原语服务，而本地方法栈是为虚拟机使用到的 Native 方法服务。它的实现的语言、方式与结构并没有强制规定，甚至有的虚拟机（譬如 Sun Hotspot 虚拟机）直接就把本地方法栈和 VM 栈合二为一。和 VM 栈一样，这个区域也会抛出 StackOverflowError 和 OutOfMemoryError 异常。

4. JVM 相关问题

4.1. 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。
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。
总体来说：栈，主要存放引用和基本数据类型；堆，用来存放 new 出来的对象实例。

4.2. 问：堆内存中到底存放什么东西？

答：对象，包括对象变量以及对象方法。

4.3. 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。
静态变量存在方法区（Method Area）中，实例变量存在堆内存（Heap）中。

4.4. 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。

4.5. 问：为什么会产生 OutOfMemory？

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。如果此时 GC 没有回收就会产生挂起情况，系统不响应了。

4.6. 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。

4.7. 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。

4.8. 问：一个机器上可以有多个 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。

4.9. 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。

4.10. 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。

4.11. 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。

4.12. 问：什么是 JIT？

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。

4.13. 问：为什么不建议在程序中显式的声明 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？

4.14. 问：JVM 有哪些调整参数？

答：非常多。

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。

5. 深入 Java 内存区域与 OOM

5.1. 概述

对于从事 C、C++程序开发的开发人员来说，在内存管理领域，他们即是拥有最高权力的皇帝又是执行最基础工作的劳动人民——拥有每一个对象的“所有权”，又担负着每一个对象生命开始到终结的维护责任。

对于 Java 程序员来说，不需要再为每一个 new 操作去写配对的 delete/free，不容易出现内容泄漏和内存溢出错误，看起来由 JVM 管理内存一切都很美好。不过，也正是因为 Java 程序员把内存控制的权力交给了 JVM，一旦出现泄漏和溢出，如果不了解 JVM 是怎样使用内存的，那排查错误将会是一件非常困难的事情。

5.2. JVM 运行时的数据区域（参照 3.4）

该章请参照 3.4 章节，补充下面两个点。

5.2.1. 运行时常量池（Runtime Constant Pool）

Class 文件中除了有类的版本、字段、方法、接口等描述等信息外，还有一项信息是常量表(constant_pool table)，用于存放编译期已可知的常量，这部分内容将在类加载后进入方

法区（永久代）存放。但是 Java 语言并不要求常量一定只有编译期预置入 Class 的常量表的内容才能进入方法区常量池，运行期间也可将新内容放入常量池（最典型的 String.intern()方法）。

运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。

5.2.2. 本机直接内存（Direct Memory）

直接内存并不是虚拟机运行时数据区的一部分，它根本就是本机内存而不是 VM 直接管理的区域。但是这部分内存也会导致 OutOfMemoryError 异常出现，因此我们放到这里一起描述。

在 JDK1.4 中新加入了 NIO 类，引入一种基于渠道与缓冲区的 I/O 方式，它可以通过本机 Native 函数库直接分配本机内存，然后通过一个存储在 Java 堆里面的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在 Java 对和本机堆中来回复制数据。

显然本机直接内存的分配不会受到 Java 堆大小的限制，但是即然是内存那肯定还是要受到本机物理内存（包括 SWAP 区或者 Windows 虚拟内存）的限制的，一般服务器管理员配置 JVM 参数时，会根据实际内存设置-Xmx 等参数信息，但经常忽略掉直接内存，使得各个内存区域总和大于物理内存限制（包括物理的和操作系统级的限制），而导致动态扩展时出现 OutOfMemoryError 异常。

5.3. 实战 OutOfMemoryError（OOM）

上述数据区域中，除了程序计数器，其他在 VM Spec 中都描述了产生 OutOfMemoryError 的情形，那我们就实战模拟一下，通过几段简单的代码，令对应的区域产生 OOM 异常以便加深认识，同时初步介绍一些与内存相关的虚拟机参数。下文的代码都是基于 Sun Hotspot 虚拟机 1.6 版的实现，对于不同公司的不同版本的虚拟机，参数与程序运行结果可能结果会有所差别。

5.3.1. Java 堆中产生 OOM

Java 堆存放的是对象实例，因此只要不断建立对象，并且保证 GC Roots 到对象之间有可达路径即可产生 OOM 异常。测试中限制 Java 堆大小为 20M，不可扩展，通过参数-XX:+HeapDumpOnOutOfMemoryError 让虚拟机在出现 OOM 异常的时候 Dump 出内存映像以便分析。（关于 Dump 映像文件分析方面的内容，可参见本文第三章《JVM 内存管理：深入 JVM 内存异常分析与调优》。）

清单 1：Java 堆 OOM 测试

```
/**
 * VM Args：-Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError
 */
public class HeapOOM {

    static class OOMObject {
    }

    public static void main(String[] args) {
        List<OOMObject> list = new ArrayList<OOMObject>();

        while (true) {
            list.add(new OOMObject());
        }
    }
}
```

运行结果

```
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid3404.hprof ...
Heap dump file created [22045981 bytes in 0.663 secs]
```

5.3.2. 栈和本地方法栈产生 OOM

Hotspot 虚拟机并不区分 VM 栈和本地方法栈，因此-Xoss 参数实际上是无效的，栈容量只由-Xss 参数设定。关于 VM 栈和本地方法栈在 VM Spec 描述了两种异常：StackOverflowError 与 OutOfMemoryError，当栈空间无法继续分配分配时，到底是内存太小还是栈太大其实某种意义上是对同一件事情的两种描述而已，在笔者的实验中，对于单线程应用尝试下面 3 种方法均无法让虚拟机产生 OOM，全部尝试结果都是获得 SOF 异常。

- 使用-Xss 参数削减栈内存容量。结果：抛出 SOF 异常时的堆栈深度相应缩小。
- 定义大量的本地变量，增大此方法对应帧的长度。结果：抛出 SOF 异常时的堆栈深度相应缩小。
- 创建几个定义很多本地变量的复杂对象，打开逃逸分析和标量替换选项，使得 JIT 编译器允许对象拆分后在栈中分配。结果：实际效果同第二点。

清单 2：VM 栈和本地方法栈 OOM 测试（仅作为第 1 点测试程序）

```
/**
 * VM Args：-Xss128k
 */
public class JavaVMStackSOF {

    private int stackLength = 1;

    public void stackLeak() {
        stackLength++;
        stackLeak();
```

```
    }

    public static void main(String[] args) throws Throwable {
        JavaVMStackSOF oom = new JavaVMStackSOF();
        try {
            oom.stackLeak();
        } catch (Throwable e) {
            System.out.println("stack length:" + oom.stackLength);
            throw e;
        }
    }
}
```

运行结果

```
stack length:2402
Exception in thread "main" java.lang.StackOverflowError
    at org.fenixsoft.oom.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:20)
    at org.fenixsoft.oom.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:21)
    at org.fenixsoft.oom.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:21)
```

5.3.3. 运行时常量池产生 OOM

要在常量池里添加内容，最简单的就是使用 String.intern()这个 Native 方法。由于常量池分配在方法区内，我们只需要通过-XX:PermSize 和-XX:MaxPermSize 限制方法区大小即可限制常量池容量。

清单 3：运行时常量池导致的 OOM 异常

```
/**
 * VM Args：-XX:PermSize=10M -XX:MaxPermSize=10M
 */
public class RuntimeConstantPoolOOM {

    public static void main(String[] args) {
        // 使用 List 保持着常量池引用，压制 Full GC 回收常量池行为
        List<String> list = new ArrayList<String>();
        // 10M 的 PermSize 在 integer 范围内足够产生 OOM 了
        int i = 0;
        while (true) {
            list.add(String.valueOf(i++).intern());
        }
    }
}
```

```
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
    at java.lang.String.intern(Native Method)
    at org.fenixsoft.oom.RuntimeConstantPoolOOM.main(RuntimeConstantPoolOOM.java:18)
```

5.3.4. 方法区产生 OOM

方法区用于存放 Class 相关信息，所以这个区域的测试我们借助 CGLib 直接操作字节码动态生成大量的 Class，值得注意的是，这里我们这个例子中模拟的场景其实经常会在实际应用中出现：当前很多主流框架，如 Spring、Hibernate 对类进行增强时，都会使用到 CGLib 这类字节码技术，当增强的类越多，就需要越大的方法区用于保证动态生成的 Class 可以加载入内存。

清单 4：借助 CGLib 使得方法区出现 OOM 异常

```
/**
 * VM Args： -XX:PermSize=10M -XX:MaxPermSize=10M
 */
public class JavaMethodAreaOOM {

    public static void main(String[] args) {
        while (true) {
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(OOMObject.class);
            enhancer.setUseCache(false);
            enhancer.setCallback(new MethodInterceptor() {
                public Object intercept(Object obj, Method method, Object[] args, MethodProxy
proxy) throws Throwable {
                    return proxy.invokeSuper(obj, args);
                }
            });
            enhancer.create();
        }
    }

    static class OOMObject {

    }
}
```

```
Caused by: java.lang.OutOfMemoryError: PermGen space
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClassCond(ClassLoader.java:632)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:616)
    ... 8 more
```

5.3.5. 总结

到此为止，我们弄清楚虚拟机里面的内存是如何划分的，哪部分区域，什么样的代码、操作可能导致 OOM 异常。虽然 Java 有垃圾收集机制，但 OOM 仍然离我们并不遥远，本章内容我们只是知道各个区域 OOM 异常出现的原因，后续我们将看看 Java 垃圾收集机制为了避免 OOM 异常出现，做出了什么样的努力。

6. 深入垃圾收集器和内存分配策略

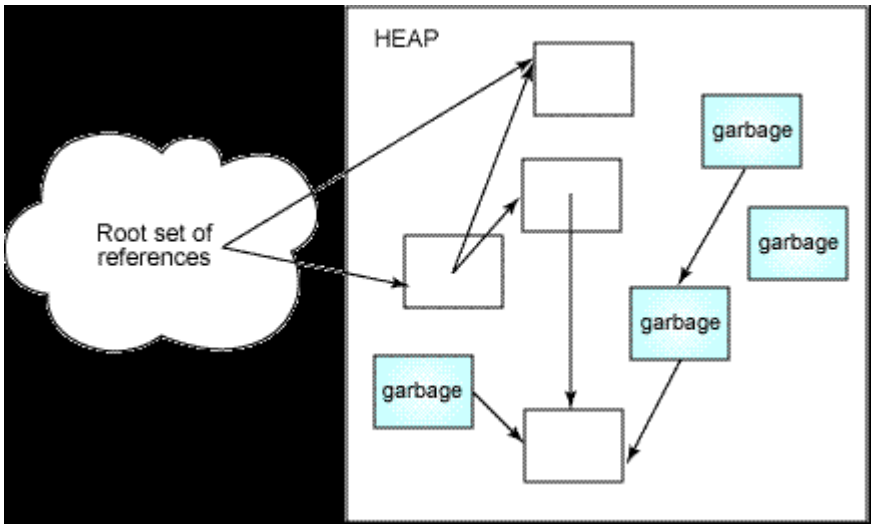
6.1. 概述

前面介绍了 Java 内存运行时区域的各个部分，其中程序计数器、VM 栈、本地方法栈三个区域随线程而生，随线程而灭；栈中的帧随着方法进入、退出而有条不紊的进行着出栈入栈操作；每一个帧中分配多少内存基本上是在 Class 文件生成时就已知的（可能会由 JIT 动态晚期编译进行一些优化，但大体上可以认为是编译期可知的），因此这几个区域的内存分配和回收具备很高的确定性，因此在这几个区域不需要过多考虑回收的问题。

而 Java 堆和方法区（包括运行时常量池）则不一样，我们必须等到程序实际运行期间才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，我们本文后续讨论中的“内存”分配与回收仅仅指这一部分内存。

6.2. 垃圾收集算法

在堆里面存放着 Java 世界中几乎所有的对象，在回收前首先要确定这些对象之中哪些还在存活，哪些已经“死去”了，即不可能再被任何途径使用的对象。



6.2.1. 引用计数算法（Reference Counting）

最初的想法，也是很多教科书判断对象是否存活的算法：给对象中添加一个引用计数器，当有一个地方引用它，计数器加 1，当引用失效，计数器减 1，任何时刻计数器为 0 的对象就是不可能再被使用的。

客观的说，引用计数算法实现简单，判定效率很高，在大部分情况下它都是一个不错的算法，但引用计数算法无法解决对象循环引用的问题。

举个简单的例子：对象 A 和 B 分别有字段 b、a，令 A.b=B 和 B.a=A，除此之外这 2 个对象再无任何引用，那实际上这 2 个对象已经不可能再被访问，但是引用计数算法却无法回收他们。

6.2.2. 根追踪算法（GC Roots Tracing）

在实际生产的语言中（Java、C#、Lisp），都是使用根追踪算法判定对象是否存活。算法基本思路就是通过一系列的称为“GC Roots”的点作为起始进行向下搜索，当一个对象到 GC Roots 没有任何引用链（Reference Chain）相连，则证明此对象是不可用的。在 Java 语言中，

GC Roots 包括：

- 在 VM 栈（帧中的本地变量）中的引用。
- 方法区中的静态引用。
- JNI（即一般说的 Native 方法）中的引用。

判断生存还是死亡？

判定一个对象死亡，至少经历两次标记过程：如果对象在进行根搜索后，发现没有与 GC Roots 相连接的引用链，那它将会被第一次标记，并在稍后执行他的 finalize()方法（如果它有的话）。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束。这点是必须的，否则一个对象在 finalize()方法执行缓慢，甚至有死循环什么的将会很容易导致整个系统崩溃。finalize()方法是对象最后一次逃脱死亡命运的机会，稍后 GC 将进行第二次规模稍小的标记，如果在 finalize()中对象成功拯救自己（只要重新建立到 GC Roots 的连接即可，譬如把自己赋值到某个引用上），那在第二次标记时它将被移除出“即将回收”的集合，如果对象这时候还没有逃脱，那基本上它就真的离死不远了。

需要特别说明的是，这里对 finalize()方法的描述可能带点悲情的艺术加工，并不代表笔者鼓励大家去使用这个方法来拯救对象。相反，笔者建议大家尽量避免使用它，这个不是 C/C++里面的析构函数，它运行代价高昂，不确定性大，无法保证各个对象的调用顺序。需要关闭外部资源之类的事情，基本上它能做的使用 try-finally 可以做的更好。

6.2.3. 标记-清除算法（Mark-Sweep）

如它的名字一样，算法分成“标记”和“清除”两个阶段。首先标记出所有需要回收的对象，然后回收所有需要回收的对象。

它的主要两个缺点：

- 效率问题，标记和清理两个过程效率都不高。
- 空间问题，标记清理之后会产生大量不连续的内存碎片，空间碎片太多可能会导致后续使用中无法找到足够的连续内存而提前触发另一次的垃圾搜集动作。

为了解决效率问题，“复制”（Copying）的收集算法出现。

6.2.4. “复制”（Copying）收集算法

它将可用内存划分为两块，每次只使用其中的一块，当半区内存用完了，仅将还存活的对象复制到另外一块上面，然后就把原来整块内存空间一次过清理掉。这样使得每次内存回收都是对整个半区的回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存就可以了，实现简单，运行高效。

只是这种算法的代价是将内存缩小为原来的一半，未免太高了一点。

现在的商业虚拟机中都是用了这一种收集算法来回收新生代，IBM 有专门研究表明新生代中的对象 98%是朝生夕死的，所以并不需要按照 1：1 的比例来划分内存空间，而是将内存分为一块较大的 eden 空间和 2 块较少的 survivor 空间，每次使用 eden 和其中一块 survivor，当回收时将 eden 和 survivor 还存活的对象一次过拷贝到另外一块 survivor 空间上，然后清理掉 eden 和用过的 survivor。Sun Hotspot 虚拟机默认 eden 和 survivor 的大小比例是 8:1，也就是每次只有 10%的内存是“浪费”的。当然，98%的对象可回收只是一般场景下的数据，

我们没有办法保证每次回收都只有 10%以内的对象存活，当 survivor 空间不够用时，需要依赖其他内存（譬如老年代）进行分配担保（Handle Promotion）。

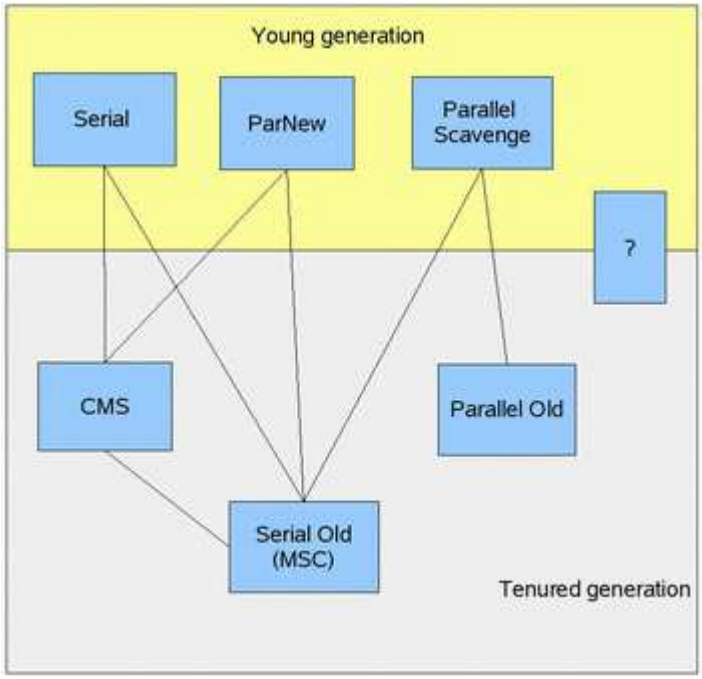
复制收集算法在对象存活率高的时候，效率有所下降。更关键的是，如果不想浪费 50%的空间，就需要有额外的空间进行分配担保用于应付半区内存中所有对象都 100%存活的极端情况，所以在老年代一般不能直接选用这种算法。因此人们提出另外一种“标记－整理”（Mark-Compact）算法，标记过程仍然一样，但后续步骤不是进行直接清理，而是令所有存活的对象一端移动，然后直接清理掉这端边界以外的内存。

6.2.5. “分代收集”（Generational Collecting）算法

当前商业虚拟机的垃圾收集都是采用“分代收集”（Generational Collecting）算法，这种算法并没有什么新的思想出现，只是根据对象不同的存活周期将内存划分为几块。一般是把 Java 堆分成新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法，譬如新生代每次 GC 都有大批对象死去，只有少量存活，那就选用复制算法只需要付出少量存活对象的复制成本就可以完成收集。

6.3. 垃圾收集器

垃圾收集器就是收集算法的具体实现，不同的虚拟机会提供不同的垃圾收集器。



上图提供了 6 种作用于不同年代的收集器，两个收集器之间存在连线的话就说明它们可以搭配使用。

我们明确一个观点：没有最好的收集器，也没有万能的收集器，只有最合适的收集器。

7. JVM 调优总结

7.1. 概念

7.1.1. 数据类型

Java 虚拟机中，数据类型可以分为两类：

- 基本类型。
- 引用类型。

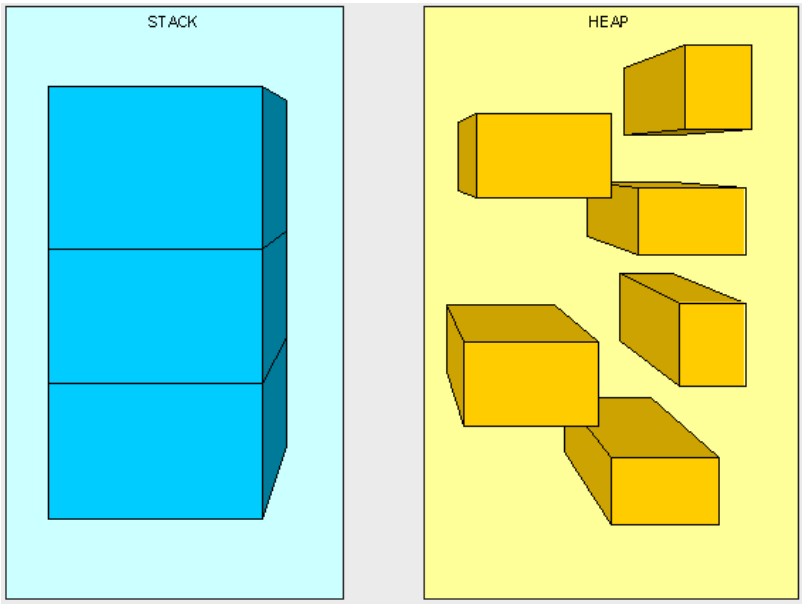
基本类型的变量保存原始值，即：它代表的值就是数值本身；而引用类型的变量保存引用值。“引用值”代表了某个对象的引用，而不是对象本身，对象本身存放在这个“引用值”所表示的地址的位置。

基本类型包括：**byte，short，int，long，char，float，double，Boolean，returnAddress。**

引用类型包括：**类类型，接口类型**和**数组。**

7.1.2. 堆与栈

堆和栈是程序运行的关键，很有必要把他们的关系说清楚。



- **栈是运行时的单位，而堆是存储的单位。**

栈解决程序的运行问题，即程序如何执行，或者说如何处理数据；堆解决的是数据存储的问题，即数据怎么放、放在哪儿。

在 Java 中一个线程就会相应有一个线程栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈。而堆则是所有线程共享的。栈因为是运行单位，因此里面存储的信息都是跟当前线程（或程序）相关信息的。包括局部变量、程序运行状态、方法返回值等等；而堆只负责存储对象信息。

- **为什么要把堆和栈区分出来？栈中不是也可以存储数据吗？**

第一，　从软件设计的角度看，栈代表了处理逻辑，而堆代表了数据。这样分开，使得处理逻辑更为清晰。分而治之的思想。这种隔离、模块化的思想在软件设计的方方面面都有体现。

第二，　堆与栈的分离，使得堆中的内容可以被多个栈共享（也可以理解为多个线程访问同一个对象）。这种共享的收益是很多的。一方面这种共享提供了一种有效的数据交互方式(如：共享内存)，另一方面，堆中的共享常量和缓存可以被所有栈访问，节省了空间。

第三，　栈因为运行时的需要，比如保存系统运行的上下文，需要进行地址段的划分。由于栈只能向上增长，因此就会限制住栈存储内容的能力。而堆不同，堆中的对象是可以根据需要动态增长的，因此栈和堆的拆分，使得动态增长成为

可能，相应栈中只需记录堆中的一个地址即可。

第四，面向对象就是堆和栈的完美结合。其实，面向对象方式的程序与以前结构化的程序在执行上没有任何区别。但是，面向对象的引入，使得对待问题的思考方式发生了改变，而更接近于自然方式的思考。当我们把对象拆开，你会发现，对象的属性其实就是数据，存放在堆中；而对象的行为（方法），就是运行逻辑，放在栈中。我们在编写对象的时候，其实即编写了数据结构，也编写的处理数据的逻辑。不得不承认，面向对象的设计，确实很美。

- **在 Java 中，main 函数就是栈的起始点，也是程序的起始点。**

程序要运行总是有一个起点的。同 C 语言一样，Java 中的 Main 就是那个起点。无论什么 Java 程序，找到 main 就找到了程序执行的入口。

- **堆中存什么？栈中存什么？**

堆中存的是对象，栈中存的是基本数据类型和堆中对象的引用。一个对象的大小是不可估计的，或者说是可以动态变化的，但是在栈中，一个对象只对应了一个 4btye 的引用（堆栈分离的好处）。

为什么不把基本类型放堆中呢？因为其占用的空间一般是 1~8 个字节——需要空间比较少，而且因为是基本类型，所以不会出现动态增长的情况——长度固定，因此栈中存储就够了，如果把他存在堆中是没有什么意义的（还会浪费空间，后面说明）。可以这么说，基本类型和对象的引用都是存放在栈中，而且都是几个字节的一个数，因此在程序运行时，他们的处理方式是统一的。但是基本类型、对象引用和对象本身就有所区别了，因为一个是栈中的数据一个是堆中的数据。最常见的一个问题就是，Java 中参数传递时的问题。

- **Java 中参数传递是传值？还是传引用？**

要说明这个问题，先要明确两点：

- 不要试图与 C 进行类比，Java 中没有指针的概念。
- 程序运行永远都是在栈中进行的，因而参数传递时，只存在传递基本类型和对象引用的问题。不会直接传对象本身。

明确以上两点后。Java 在方法调用传递参数时，因为没有指针，所以**它都是进行传值调用**（这点可以参考 C 的传值调用）。因此，很多书里面都说 Java 是进行传值调用，这点没有问题，而且也简化的 C 中复杂性。

但是传引用的错觉是如何造成的呢？在运行栈中，基本类型和引用的处理是一样的，都是传值，所以，如果是传引用的方法调用，也同时可以理解为“传引用值”的传值调用，即引用的处理跟基本类型是完全一样的。但是当进入被调用方法时，被传递的这个引用的值，被程序解释（或者查找）到堆中的对象，这个时候才对应到真正的对象。如果此时进行修改，修改的是引用对应的对象，而不是引用本身，即：修改的是堆中的数据。所以这个修改是可以保持的了。

对象，从某种意义上说，是由基本类型组成的。可以把一个对象看作为一棵树，对象的属性如果还是对象，则还是一颗树（即非叶子节点），基本类型则为树的叶子节点。程序参数传递时，被传递的值本身都是不能进行修改的，但是，如果这个值是一个非叶子节点（即一个对象引用），则可以修改这个节点下面的所有内容。

堆和栈中，栈是程序运行最根本的东西。程序运行可以没有堆，但是不能没有栈。而堆是为栈进行数据存储服务，说白了堆就是一块共享的内存。不过，正是因为堆和栈的分离的

思想，才使得 Java 的垃圾回收成为可能。

Java 中，栈的大小通过-Xss 来设置，当栈中存储数据比较多时，需要适当调大这个值，否则会出现 java.lang.StackOverflowError 异常。常见的出现这个异常的是无法返回的递归，因为此时栈中保存的信息都是方法返回的记录点。

7.1.3. Java 对象的大小

基本数据的类型的大小是固定的，这里就不多说了。对于非基本类型的 Java 对象，其大小就值得商榷。

在 Java 中，**一个空 Object 对象的大小是 8byte**，这个大小只是保存堆中一个没有任何属性的对象的大小。看下面语句：

```
Object ob = new Object();
```

这样在程序中完成了一个 Java 对象的生命，但是它所占的空间为：**4byte+8byte**。4byte 是上面部分所说的 Java 栈中保存引用的所需要的空间。而那 8byte 则是 Java 堆中对象的信息。因为所有的 Java 非基本类型的对象都需要默认继承 Object 对象，因此不论什么样的 Java 对象，其大小都必须是大于 8byte。

有了 Object 对象的大小，我们就可以计算其他对象的大小了。

```
Class NewObject {
    int count;
    boolean flag;
    Object ob;
}
```

其大小为：空对象大小(8byte)+int 大小(4byte)+Boolean 大小(1byte)+空 Object 引用的大小(4byte)=17byte。但是因为 Java 在对对象内存分配时都是以 8 的整数倍来分，因此大于 17byte 的最接近 8 的整数倍的是 24，因此此对象的大小为 24byte。

这里需要注意一下**基本类型的包装类型的大小**。因为这种包装类型已经成为对象了，因此需要把他们作为对象来看待。包装类型的大小至少是 12byte（声明一个空 Object 至少需要的空间），而且 12byte 没有包含任何有效信息，同时，因为 Java 对象大小是 8 的整数倍，因此**一个基本类型包装类的大小至少是 16byte**。这个内存占用是很恐怖的，它是使用基本类型的 N 倍（N>2），有些类型的内存占用更是夸张（随便想下就知道了）。因此，可能的话应尽量少使用包装类。在 JDK5.0 以后，因为加入了自动类型装换，因此，Java 虚拟机会在存储方面进行相应的优化。

7.1.4. 引用类型

对象引用类型分为：

- **强引用**：就是我们一般声明对象是时虚拟机生成的引用，强引用环境下，垃圾回收时需要严格判断当前对象是否被强引用，如果被强引用，则不会被垃圾回收。
- **软引用**：软引用一般被做为缓存来使用。与强引用的区别是，软引用在垃圾回收时，虚拟机会根据当前系统的剩余内存来决定是否对软引用进行回收。如果剩余内存比较紧张，则虚拟机会回收软引用所引用的空间；如果剩余内存相对富裕，则不会进行回收。

换句话说，虚拟机在发生 OutOfMemory 时，肯定是没有软引用存在的。

- **弱引用**：弱引用与软引用类似，都是作为缓存来使用。但与软引用不同，弱引用在进行垃圾回收时，是一定会被回收掉的，因此其生命周期只存在于一个垃圾回收周期内。

强引用不用说，我们系统一般在使用时都是用的强引用。而“软引用”和“弱引用”比较少见。他们一般被作为缓存使用，而且一般是在内存大小比较受限的情况下做为缓存。因为如果内存足够大的话，可以直接使用强引用作为缓存即可，同时可控性更高。因而，他们常见的是被使用在桌面应用系统的缓存。

7.2. 基本垃圾回收算法

可以从不同的的角度去划分垃圾回收算法。

7.2.1. 按照基本回收策略分

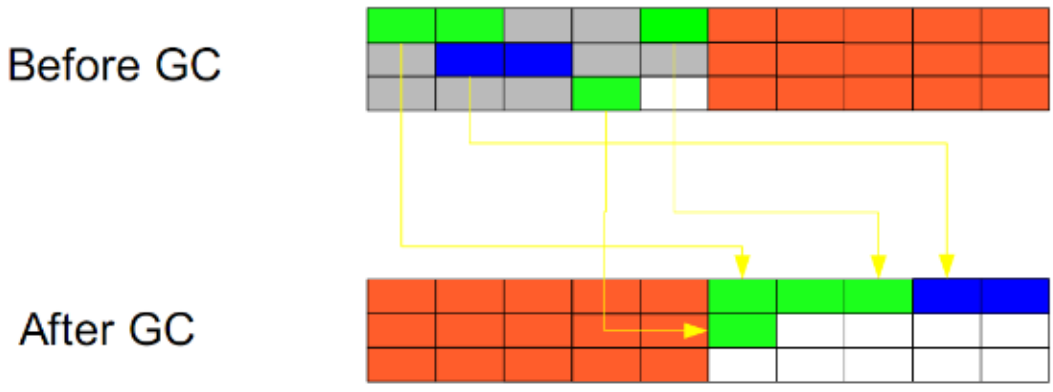
引用计数（Reference Counting）：

比较古老的回收算法。原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。垃圾回收时，只用收集计数为 0 的对象。此算法最致命的是无法处理循环引用的问题。

标记-清除（Mark-Sweep）：

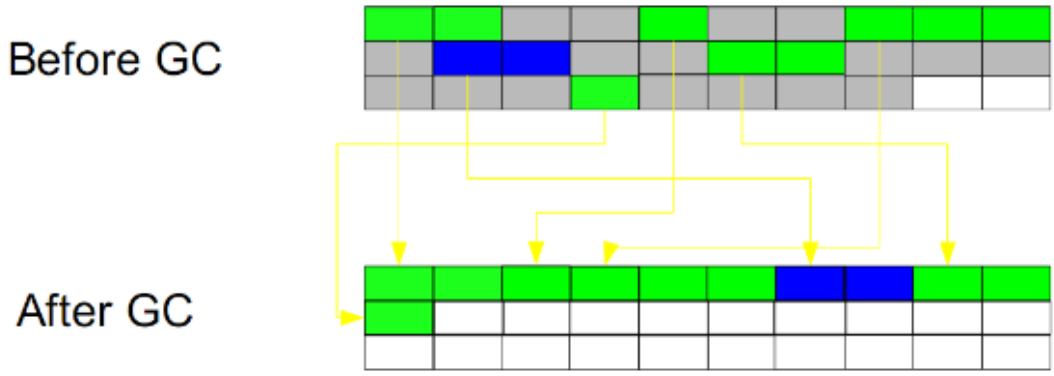
此算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，同时，会产生内存碎片。

复制（Copying）：



此算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。次算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现"碎片"问题。当然，此算法的缺点也是很明显的，就是需要两倍内存空间。

标记-整理（Mark-Compact）：



此算法结合了"标记-清除"和"复制"两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象"压缩"到堆的其中一块，按顺序排放。此算法避免了"标记-清除"的碎片问题，同时也避免了"复制"算法的空间问题。

7.2.2. 按分区对待的方式分

增量收集（Incremental Collecting）：实时垃圾回收算法，即：在应用进行的同时进行垃圾回收。不知道什么原因 JDK5.0 中的收集器没有使用这种算法的。

分代收集（Generational Collecting）：基于对对象生命周期分析后得出的垃圾回收算法。把对象分为年青代、年老代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进行回收。现在的垃圾回收器（从 J2SE1.2 开始）都是使用此算法的。

7.2.3. 按系统线程分

串行收集：串行收集使用单线程处理所有垃圾回收工作，因为无需多线程交互，实现容易，而且效率比较高。但是，其局限性也比较明显，即无法使用多处理器的优势，所以此收集适合单处理器机器。当然，此收集器也可以用在小数据量（100M 左右）情况下的多处理器机器上。

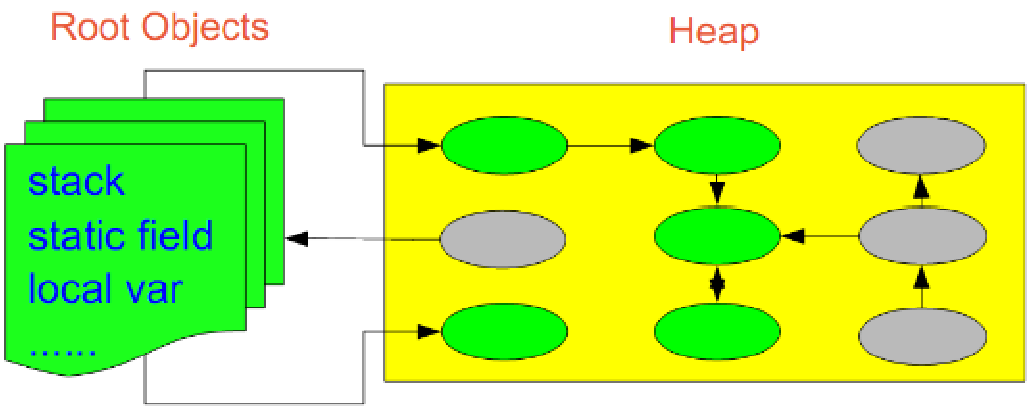
并行收集：并行收集使用多线程处理垃圾回收工作，因而速度快，效率高。而且理论上 CPU 数目越多，越能体现出并行收集器的优势。

并发收集：相对于串行收集和并行收集而言，前面两个在进行垃圾回收工作时，需要暂停整个运行环境，而只有垃圾回收程序在运行，因此，系统在垃圾回收时会有明显的暂停，而且暂停时间会因为堆越大而越长。

7.3. 垃圾回收面临的问题

7.3.1. 如何区分垃圾

上面说到的“引用计数”法，通过统计控制生成对象和删除对象时的引用数来判断。垃圾回收程序收集计数为 0 的对象即可。但是这种方法无法解决循环引用。所以，后来实现的垃圾判断算法中，都是从程序运行的根节点出发，遍历整个对象引用，查找存活的对象。那么在这种方式的实现中，**垃圾回收从哪儿开始的呢**？即，从哪儿开始查找哪些对象是正在被当前系统使用的。上面分析的堆和栈的区别，其中栈是真正进行程序执行地方，所以要获取哪些对象正在被使用，则需要从 Java 栈开始。同时，一个栈是与一个线程对应的，因此，如果有多个线程的话，则必须对这些线程对应的所有的栈进行检查。



同时，除了栈外，还有系统运行时的寄存器等，也是存储程序运行数据的。这样，以栈或寄存器中的引用为起点，我们可以找到堆中的对象，又从这些对象找到对堆中其他对象的引用，这种引用逐步扩展，最终以 null 引用或者基本类型结束，这样就形成了一颗以 Java 栈中引用所对应的对象为根节点的一颗对象树，如果栈中有多个引用，则最终会形成多颗对象树。在这些对象树上的对象，都是当前系统运行所需要的对象，不能被垃圾回收。而其他剩余对象，则可以视为无法被引用到的对象，可以被当做垃圾进行回收。

因此，**垃圾回收的起点是一些根对象（java 栈，静态变量，寄存器...）**。而最简单的 Java 栈就是 Java 程序执行的 main 函数。这种回收方式，也是上面提到的“标记-清除”的回收方式。

7.3.2. 如何处理碎片

由于不同 Java 对象存活时间是不一定的，因此，在程序运行一段时间以后，如果不进行内存整理，就会出现零散的内存碎片。碎片最直接的问题就是会导致无法分配大块的内存空间，以及程序运行效率降低。所以，在上面提到的基本垃圾回收算法中，“复制”方式和“标记-整理”方式，都可以解决碎片的问题。

7.3.3. 如何解决同时存在的对象创建和对象回收问题

垃圾回收线程是回收内存的，而程序运行线程则是消耗（或分配）内存的，**一个回收内存，一个分配内存**，从这点看，两者是矛盾的。因此，在现有的垃圾回收方式中，要进行垃圾回收前，一般都需要暂停整个应用（即：暂停内存的分配），然后进行垃圾回收，回收完成后再继续应用。这种实现方式是最直接，而且最有效的解决二者矛盾的方式。

但是**这种方式有一个很明显的弊端，就是当堆空间持续增大时，垃圾回收的时间也将会相应的持续增大，对应应用暂停的时间也会相应的增大**。一些对相应时间要求很高的应用，比如最大暂停时间要求是几百毫秒，那么当堆空间大于几个 G 时，就很有可能超过这个限制，在这种情况下，垃圾回收将会成为系统运行的一个瓶颈。为解决这种矛盾，有了**并发垃圾回收算法**，使用这种算法，垃圾回收线程与程序运行线程同时运行。在这种方式下，

解决了暂停的问题，但是因为需要在新生成对象的同时又要回收对象，算法复杂性会大大增加，系统的处理能力也会相应降低，同时，“碎片”问题将会比较难解决。

7.4. 分代垃圾回收详述

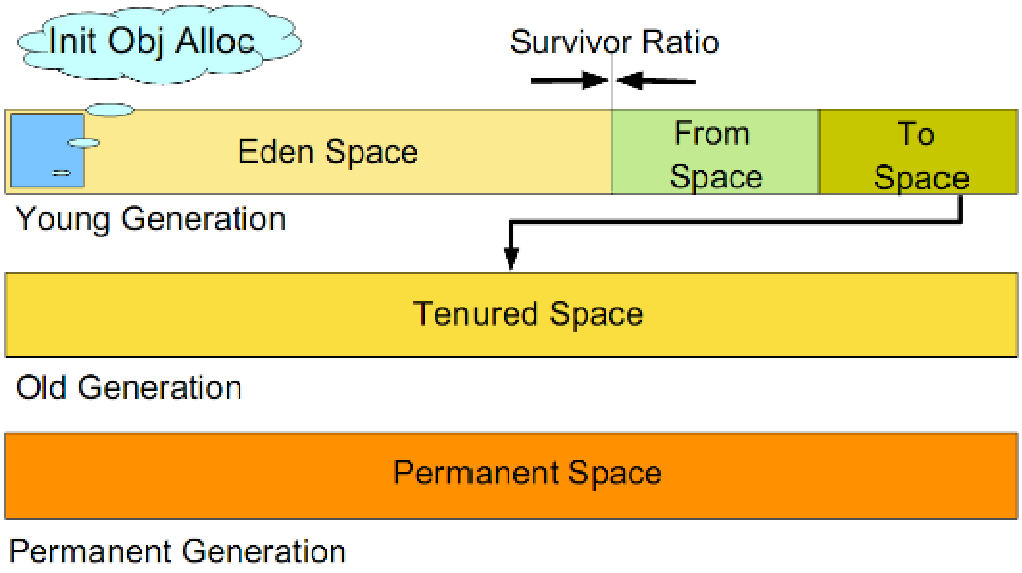
7.4.1. 为什么分代

分代的垃圾回收策略，是基于这样一个事实：**不同的对象的生命周期是不一样的**。因此，不同生命周期的对象可以采取不同的收集方式，以便提高回收效率。

在 Java 程序运行的过程中，会产生大量的对象，其中有些对象是与业务信息相关，比如 Http 请求中的 Session 对象、线程、Socket 连接，这类对象跟业务直接挂钩，因此生命周期比较长。但是还有一些对象，主要是程序运行过程中生成的临时变量，这些对象生命周期会比较短，比如：String 对象，由于其不变类的特性，系统会产生大量的这些对象，有些对象甚至只用一次即可回收。

试想，在不进行对象存活时间区分的情况下，每次垃圾回收都是对整个堆空间进行回收，花费时间相对会长，同时，因为每次回收都需要遍历所有存活对象，但实际上，对于生命周期长的对象而言，这种遍历是没有效果的，因为可能进行了很多次遍历，但是他们依旧存在。因此，分代垃圾回收采用分治的思想，进行代的划分，把不同生命周期的对象放在不同代上，不同代上采用最适合它的垃圾回收方式进行回收。

7.4.2. 如何分代



虚拟机中的共划分为三个代：**年轻代（Young Generation）、年老点（Old Generation）和持久代（Permanent Generation）**。其中持久代主要存放的是 Java 类的类信息，与垃圾收集要收集的 Java 对象关系不大。年轻代和年老代的划分是对垃圾收集影响比较大的。

- **年轻代**

所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。年轻代分三个区。一个 Eden 区，两个 Survivor 区(一般而言)。大部分对象在 Eden 区中生成。当 Eden 区满时，还存活的对象将被复制到 Survivor 区（两个中的一个），

当这个 Survivor 区满时，此区的存活对象将被复制到另外一个 Survivor 区，当这个 Survivor 去也满了的时候，从第一个 Survivor 区复制过来的并且此时还存活的对象，将被复制“年老区(Tenured)”。需要注意，Survivor 的两个区是对称的，没先后关系，所以同一个区中可能同时存在从 Eden 复制过来 对象，和从前一个 Survivor 复制过来的对象，而复制到年老区的只有从第一个 Survivor 去过来的对象。而且，Survivor 区总有一个是空的。同时，根据程序需要，Survivor 区是可以配置为多个的（多于两个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

- **年老代**

在年轻代中经历了 N 次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

- **持久代**

用于存放静态文件，如今 Java 类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些 class，例如 Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。持久代大小通过-XX:MaxPermSize=<N>进行设置。

7.5. 典型配置和调优举例

7.5.1. 堆设置

-Xms：初始堆大小。
-Xmx：最大堆大小。
-XX:NewSize=n：设置年轻代大小。
-XX:NewRatio=n：设置年轻代和年老代的比值。如:为 3，表示年轻代与年老代比值为 1：3，年轻代占整个年轻代年老代和的 1/4。
-XX:SurvivorRatio=n：年轻代中 Eden 区与两个 Survivor 区的比值。注意 Survivor 区有两个。如：3，表示 Eden：Survivor=3：2，一个 Survivor 区占整个年轻代的 1/5。
-XX:MaxPermSize=n：设置持久代大小。

7.5.2. 收集器设置

-XX:+UseSerialGC：设置串行收集器。
-XX:+UseParallelGC：设置并行收集器。
-XX:+UseParalledlOldGC：设置并行年老代收集器。
-XX:+UseConcMarkSweepGC：设置并发收集器。

7.5.3. 垃圾回收统计信息设置

-XX:+PrintGC
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
-Xloggc:filename

7.5.4. 并行收集器设置

-XX:ParallelGCThreads=n：设置并行收集器收集时使用的 CPU 数。并行收集线程数。
-XX:MaxGCPauseMillis=n：设置并行收集最大暂停时间。

-XX:GCTimeRatio=n：设置垃圾回收时间占程序运行时间的百分比。公式为 1/(1+n)。

7.5.5. 并发收集器设置

-XX:+CMSIncrementalMode：设置为增量模式。适用于单 CPU 情况。

-XX:ParallelGCThreads=n：设置并发收集器年轻代收集方式为并行收集时，使用的 CPU 数。并行收集线程数。

7.5.6. 年轻代大小选择

响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制（根据实际情况选择）。在此种情况下，年轻代收集发生的频率也是最小的。同时，减少到达年老代的对象。

吞吐量优先的应用：尽可能的设置大，可能到达 Gbit 的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合 8CPU 以上的应用。

7.5.7. 年老代大小选择

响应时间优先的应用：年老代使用并发收集器，所以其大小需要小心设置，一般要考虑**并发会话率**和**会话持续时间**等一些参数。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：

1. 并发垃圾收集信息
2. 持久代并发收集次数
3. 传统 GC 信息
4. 花在年轻代和年老代回收上的时间比例

减少年轻代和年老代花费的时间，一般会提高应用的效率。