JAVA虚拟机

一、什么是JVM

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。

引入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。

Java为什么是“平台无关的编程语言”？

- Java虚拟机是一个可以执行Java字节码的虚拟机进程。
- Java源文件被编译成能被Java虚拟机执行的字节码文件(.class文件)。
- Java被设计成允许应用程序可以运行在任意的平台，而不需要程序员为每一个平台单独重写或者是重新编译。
- Java虚拟机让这个变为可能，因为它知道底层硬件平台的指令长度和其他特性。

JRE/JDK/JVM是什么关系？

答：JRE是Java运行环境，即（Java Runtime Environment），也就是Java平台。所有的Java程序都要在JRE下才能运行。

JDK是开发工具包，即（Java Development Kit），它是程序开发者用来编译、调试Java程序，它也是Java程序，也需要JRE才能运行。

JVM是Java虚拟机，即（Java Virual Machine），它是JRE的一部分，一个虚构出来的计算机，它支持跨平台。

二、JVM结构原理

1.JVM原理？

JVM是Java核心和基础，在Java编译器和os平台之间的虚拟处理器。他可以在上面执行Java的字节码程序。Java编译器只要面向JVM，生成JVM能理解的代码或字节码文件。Java源文件经编译成字节码程序，通过JVM将每一条指令翻译成不同平台的机器码，通过特定平台运行。

2.JVM体系结构：

类加载器：加载class文件；

执行引擎：执行字节码或者执行本地方法

运行时数据区：包括方法区、堆、Java栈、PC寄存器、本地方法栈

3.JVM运行时数据区？

a.PC寄存器：用于存储每个线程下一步将执行的JVM指令；

b.栈：栈是线程私有的，每个线程创建的同时都会创建的同时都会创建JVM栈，JVM栈中存放当前线程中的局部基本类型的变量；

c.堆：它是JVM用来存储对象实例以及数组值的区域，可以认为Java中所有通过new创建的对象的内存都在此分配，Heap中的对象的内存需要等待GC进行回收。堆是JVM所有线程共享的。

d.方法区：存放了所加载的类信息（名称、修饰符等）、类中的静态变量、类中定义为final类型的常量、类中的Field信息、类中的方法信息；通过class对象中的getName等方法来获取信息时，实际这些数据是来源于方法区，方法区是全局共享的。

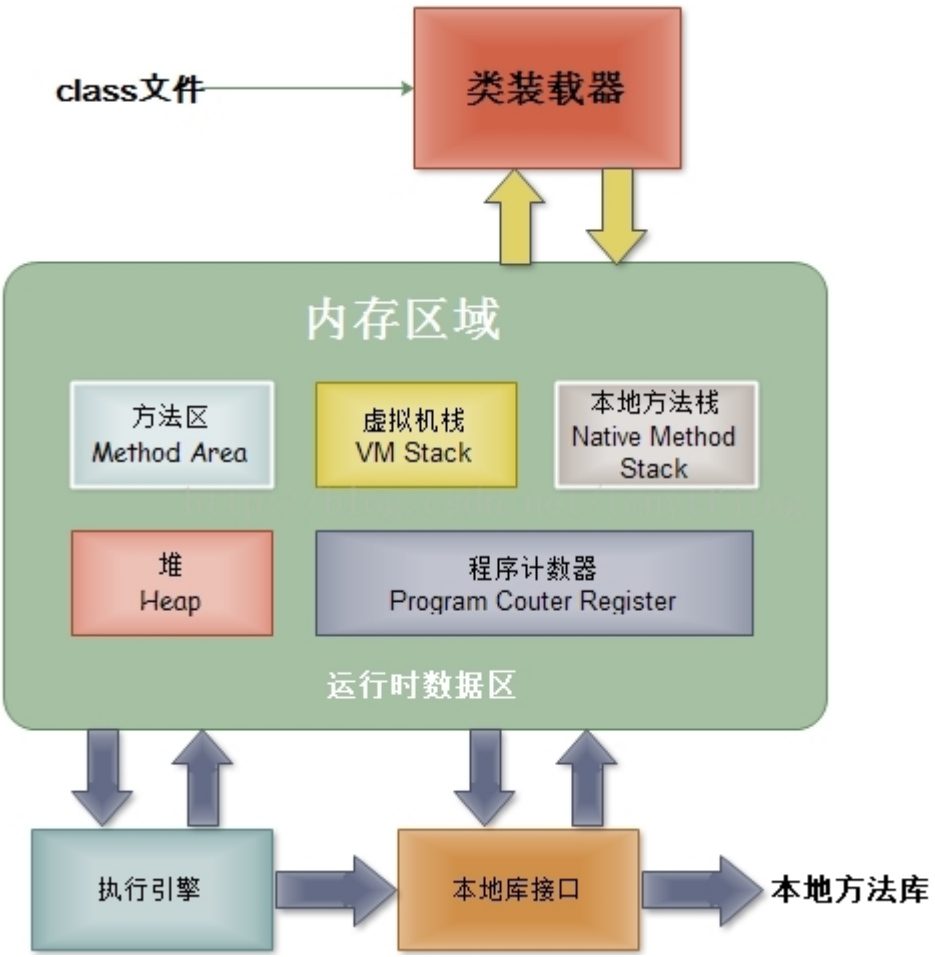
e.运行时常量池：存放类中固定的常量信息、方法和Field的引用信息等，其空间是从方法区中分配。

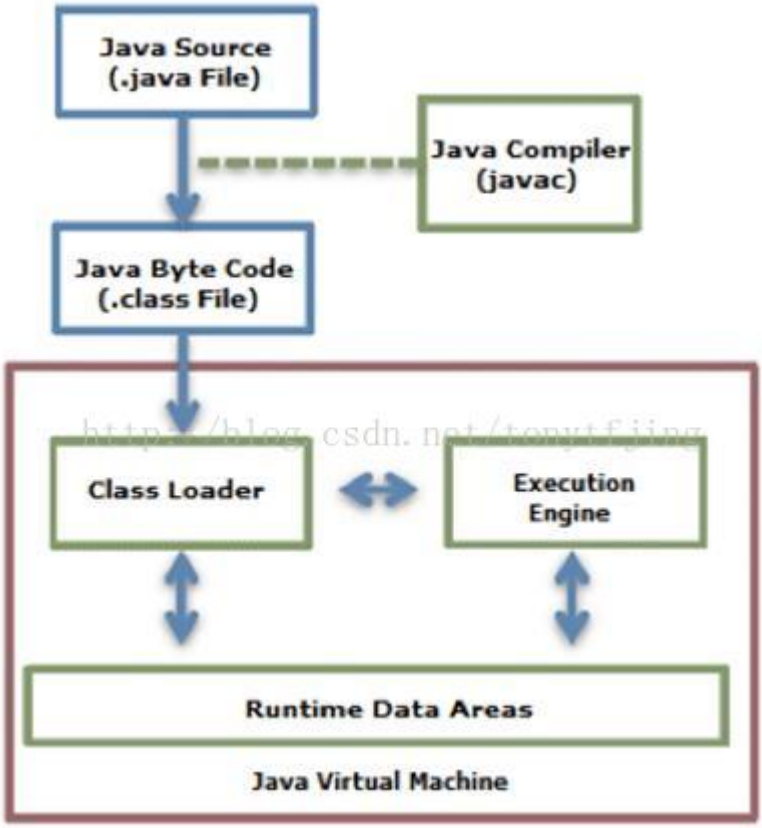
f.本地方法栈：JVM采用本地方法栈来支持native方法的执行，此区域用于存储每个native方法调用的状态。

6、如何判断对象是否存活：

a.引用计数法：给对象中添加一个引用计数器，当一个地方引用了对象，计数加1；当引用失效，计数器减1；当计数器为0表示该对象已死、可回收；但很难解决循环引用问题；

b.可达性分析：通过一系列称为“GC Root”的对象作为起点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象GC Roots没有任何引用链相连，则证明此对象已死、可回收。Java中可以作为GC Roots的对象包括：虚拟机栈中引用的对象、本地方法栈中native方法引用的对象、方法区静态属性引用的对象、方法区常量引用的对象。



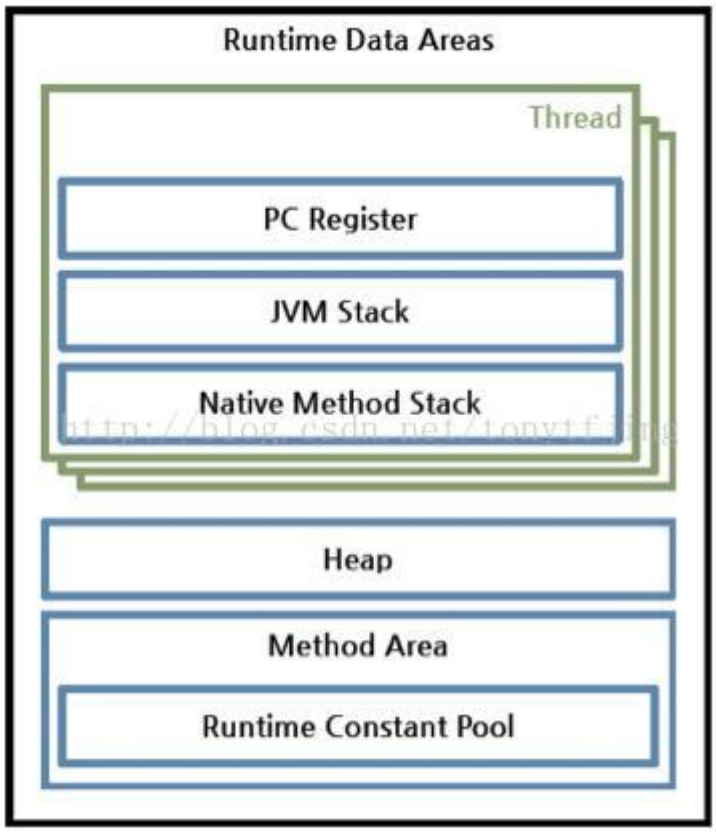


1.类加载器（ClassLoader）:在JVM启动时或者在类运行时将需要的class加载到JVM中。

2.执行引擎：负责执行class文件中包含的字节码指令

3.内存区（也叫运行时数据区）

是在JVM运行的时候操作所分配的内存区。运行时内存区主要可以划分为5个区域，如图：



1.方法区(Method Area)：用于存储类结构信息的地方，包括常量池、静态变量、构造函数等。虽然JVM规范把方法区描述为堆的一个逻辑部分， 但它却有个别名non-heap（非堆），所以大家不要搞混淆了。方法区还包含一个运行时常量池。

2.java堆(Heap)：存储java实例或者对象的地方。这块是GC的主要区域（后面解释）。从存储的内容我们可以很容易知道，方法区和堆是被所有java线程共享的。

3.java栈(Stack)：java栈总是和线程关联在一起，每当创建一个线程时，JVM就会为这个线程创建一个对应的java栈。在这个java栈中又会包含多个栈帧，每运行一个方法就创建一个栈帧，用于存储局部变量表、操作栈、方法返回值等。每一个方法从调用直至执行完成的过程，就对应一个栈帧在java栈中入栈到出栈的过程。所以java栈是现成私有的。

4.程序计数器(PC Register)：用于保存当前线程执行的内存地址。由于JVM程序是多线程执行的（线程轮流切换），所以为了保证线程切换回来后，还能恢复到原先状态，就需要一个独立的计数器，记录之前中断的地方，可见程序计数器也是线程私有的。

5.本地方法栈(Native Method Stack)：和java栈的作用差不多，只不过是为JVM使用到的native方法服务的。

4.本地方法接口：主要是调用C或C++实现的本地方法及返回结果。

三种JVM:

① Sun公司的HotSpot；

② BEA公司的JRockit；

③ IBM公司的J9 JVM；

在JDK1.7及其以前我们所使用的都是Sun公司的HotSpot，但由于Sun公司和BEA公司都被oracle收购，jdk1.8将采用Sun公司的HotSpot和BEA公司的JRockit两个JVM中精华形成jdk1.8的JVM。

三、类加载机制

1.当你去面试，遇到这样的问题怎么办？

第一题：可以不可以自己写个String类

```
答：
如果包名相同 是不可以，因为 根据类加载的双亲委派机制，会去加载父类，父类发现冲突了String就不再加载了，
但是如果包的名不同就可以重写。
例如：
com.soft863;

public class String {

    public static void main(java.lang.String[] args) {
        System.out.println(String.class.getClassLoader());
        java.lang.String str = "123";
        System.out.println(str.getClass().getClassLoader());
    }
}
```

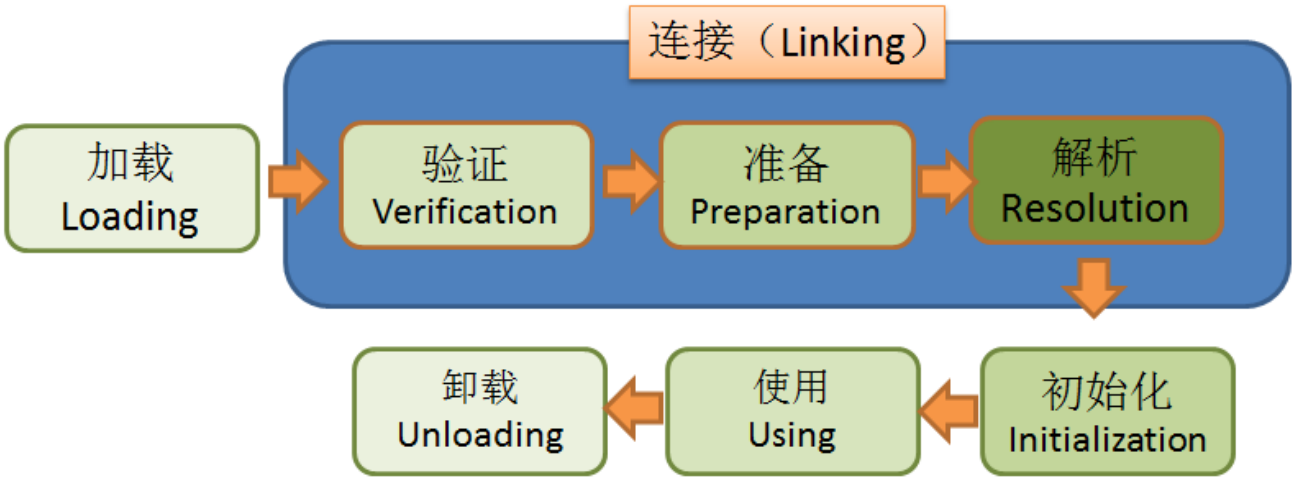
第二题：能否在加载类的时候，对类的字节码进行修改

```
答：
可以，使用Java探针技术

我们利用javaAgent和ASM字节码技术开发java探针工具，实现原理如下：

jdk1.5以后引入了javaAgent技术，javaAgent是运行方法之前的拦截器。我们利用javaAgent和ASM字节码技术，在JVM加载class二进制文件的时候，利用ASM动态的修改加载的class文件，在监控的方法前后添加计时器功能，用于计算监控方法耗时，同时将方法耗时及内部调用情况放入处理器，处理器利用栈先进后出的特点对方法调用先后顺序做处理，当一个请求处理结束后，将耗时方法轨迹和入参map输出到文件中，然后根据map中相应参数或耗时方法轨迹中的关键代码区分出我们要抓取的耗时业务。最后将相应耗时轨迹文件取下来，转化为xml格式并进行解析，通过浏览器将代码分层结构展示出来，方便耗时分析

Java探针工具功能点：

1、支持方法执行耗时范围抓取设置，根据耗时范围抓取系统运行时出现在设置耗时范围的代码运行轨迹。

2、支持抓取特定的代码配置，方便对配置的特定方法进行抓取，过滤出关系的代码执行耗时情况。

3、支持APP层入口方法过滤，配置入口运行前的方法进行监控，相当于监控特有的方法耗时，进行方法专题分析。

4、支持入口方法参数输出功能，方便跟踪耗时高的时候对应的入参数。

5、提供WEB页面展示接口耗时展示、代码调用关系图展示、方法耗时百分比展示、可疑方法凸显功能。
```

那么，什么是类加载机制呢？

其实：JVM 的类加载机制是指 JVM 把描述类的数据从 .class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这就是 JVM 的类加载机制。

类加载的生命周期 类的生命周期总共分为7个阶段：加载、验证、准备、解析、初始化、使用和卸载。其中验证、准备、解析三个步骤又可统称为连接。 加载、验证、准备、初始化和卸载五个步骤的顺序都是确定的，解析阶段在某些情况下有可能发生在初始化之后，这是为了支持 Java 语言的运行期绑定的特性。 在 JVM 虚拟机规范中并没有规定加载的时机，但是却规定了初始化的时机，而加载、验证、准备三个步骤是在初始化之前。



类加载器 一般分为启动类加载器（Bootstrap ClassLoader），扩展类加载器（Extension ClassLoader），应用程序类加载器（Application ClassLoader）

JVM的类加载机制主要有如下三种机制：

①全盘负责， 当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显式使用另外一个类加载器来载入。

②父类委托， 先让parent（父）类加载器视图加载该Class，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类。（类加载器之间的父子关系并不是类继承上的父子关系，是类加载器实例之间的关系）

③缓存机制， 保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区中搜寻该Class，只有当缓存区中不存在该Class对象时，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区总。这就是为什么修改Class后，必须重启JVM，程序所做的修改才会生效的原因。

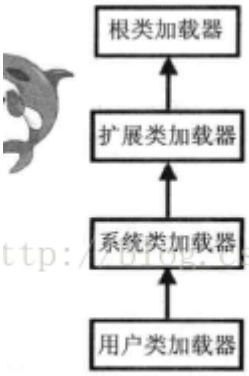


图 18.1　JVM 中 4 种类加载器的层次结构

```
1
2  import java.io.IOException;
3  import java.net.URL;
4  import java.util.Enumeration;
5
6  public class ClassLoaderPropTest {
7      public static void main(String[] args) throws IOException{
8          ClassLoader systemLoader = ClassLoader.getSystemClassLoader();
9          System.out.println("系统类加载器：" + systemLoader);
10         /*
11          获取系统类加载器的加载路径--通常CLASSPATH环境变量指定
12          如果操作系统没有指定ClASSPATH环境变量，则默认以当前路径作为系统类加载器的加载路径
13          */
14         Enumeration<URL> em1 = systemLoader.getResources("");
15         while(em1.hasMoreElements()){
16             System.out.println(em1.nextElement());
17         }
18         //获取系统类加载器的父加载器，得到扩展类加载器
19         ClassLoader extensionLoader = systemLoader.getParent();
20         System.out.println("扩展类加载器：" + extensionLoader);
21         System.out.println("扩展类加载器的加载路径：" + System.getProperty("java.ext.dirs"));
22         System.out.println("扩展类加载器的parent：" + extensionLoader.getParent());
23     }
24 }
25
```

输出

```
1  系统类加载器：sun.misc.Launcher$AppClassLoader@49aa95c
2  file:/D:/64liferay/front/javaspace/20160509JavaReflect/bin/
3  扩展类加载器：sun.misc.Launcher$ExtClassLoader@45e4d960
4  扩展类加载器的加载路径：C:\Program
   Files\Java\jdk1.7.0_80\jre\lib\ext;C:\Windows\Sun\Java\lib\ext
5  扩展类加载器的parent：null
```

1.类加载的时机

- 隐式加载 new 创建类的实例,
- 显式加载：loaderClass,forName等
- 访问类的静态变量，或者为静态变量赋值
- 调用类的静态方法
- 使用反射方式创建某个类或者接口对象的Class对象。
- 初始化某个类的子类
- 直接使用`java.exe`命令来运行某个主类

类加载的过程

我们编写的`java`文件都是保存着业务逻辑代码。`java`编译器将`.java`文件编译成扩展名为`.class`的文件。.class 文件中保存着java转换后，虚拟机将要执行的指令。当需要某个类的时候，java虚拟机会加载 .class 文件，并创建对应的class对象，将class文件加载到虚拟机的内存，这个过程被称为类的加载。

加载

类加载过程的一个阶段，ClassLoader通过一个类的完全限定名查找此类字节码文件，并利用字节码文件创建一个class对象。

验证

目的在于确保class文件的字节流中包含信息符合当前虚拟机要求，不会危害虚拟机自身的安全，主要包括四种验证：文件格式的验证，元数据的验证，字节码验证，符号引用验证。

准备

为类变量（static修饰的字段变量）分配内存并且设置该类变量的初始值，（如static int i = 5 这里只是将 i 赋值为0，在初始化的阶段再把 i 赋值为5)，这里不包含final修饰的static ，因为final在编译的时候就已经分配了。这里不会为实例变量分配初始化，类变量会分配在方法区中，实例变量会随着对象分配到Java堆中。

解析

这里主要的任务是把常量池中的符号引用替换成直接引用

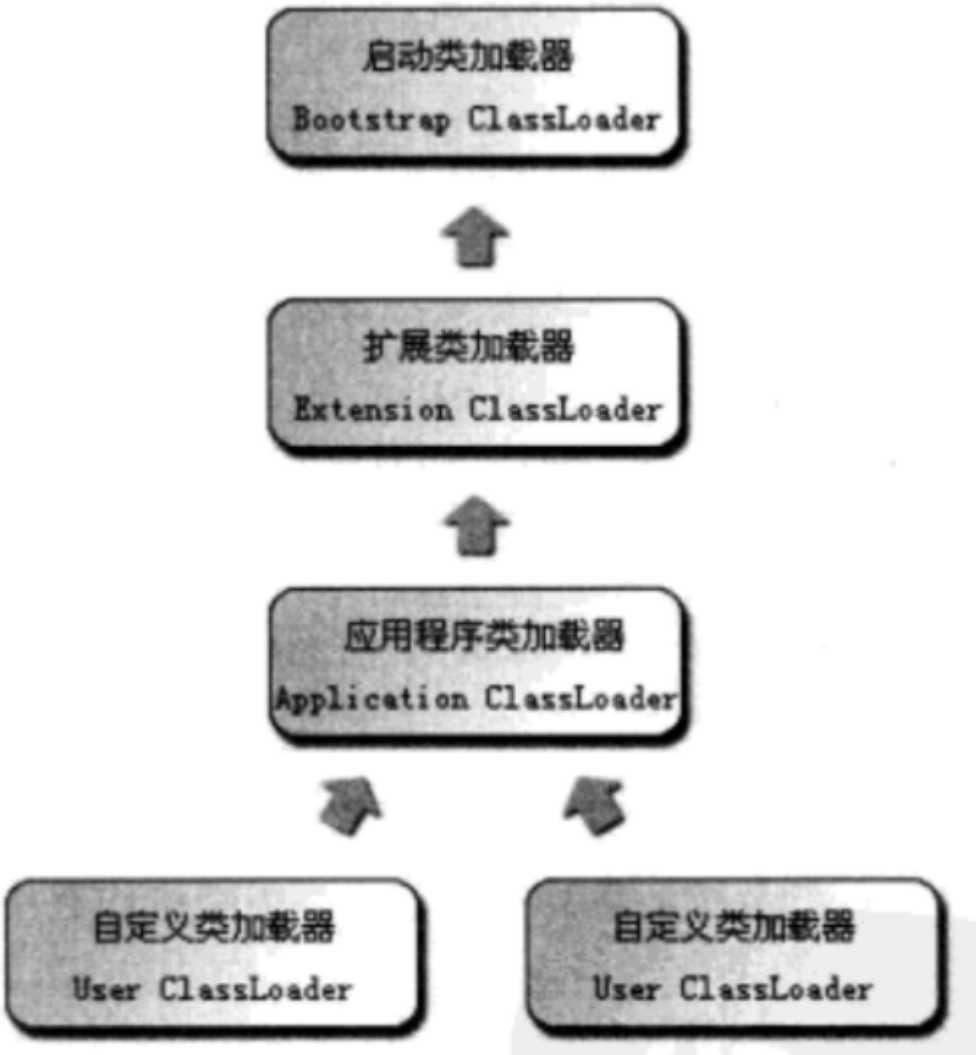
初始化

这里是类记载的最后阶段，如果该类具有父类就进行对父类进行初始化，执行其静态初始化器（静态代码块）和静态初始化成员变量。（前面已经对static 初始化了默认值，这里我们对它进行赋值，成员变量也将被初始化）

forName和loaderClass区别

- Class.forName()得到的class是已经初始化完成的。
- Classloader.loaderClass得到的class是还没有链接（验证，准备，解析三个过程被称为链接）的。

双亲委派



上图所示的这种关系我们就称之类加载器的双亲委派模型。在双亲委派模型中，除了顶层的 Bootstrap ClassLoader 之外，其他的类加载器都有自己的父加载器。

双亲委派模型的工作流程：

如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器，只有当父类加载器无法完成这个类加载请求时，才会让子类加载器去处理这个请求。

双亲委派模式要求除了顶层的启动类加载器之外，其余的类加载器都应该有自己的父类加载器，但是在双亲委派模式中父子关系采取的并不是继承的关系，而是采用组合关系来复用父类加载器的相关代码。

下面是关键性的源代码

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException {
    // 增加同步锁，防止多个线程加载同一类
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class<?> c = findLoadedClass(name);

        if (c == null) {
```

```
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else { // ExtClassLoader没有继承BootStrapClassLoader
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }

            if (c == null) {
                // If still not found, then invoke findClass in order
                // to find the class.
                long t1 = System.nanoTime();
                // AppClassLoader去我们项目中查找是否有这个文件，如有加载进来
                // 没有就到用户自定义ClassLoader中加载。如果没有就抛出异常
                c = findClass(name);

                // this is the defining class loader; record the stats
                sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);
                sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                sun.misc.PerfCounter.getFindClasses().increment();
            }
        }
        if (resolve) {
            resolveClass(c);
        }
        return c;
    }
}
```

工作原理

如果一个类收到了类加载的请求，它并不会自己先去加载，而是把这个请求委托给父类加载器去执行，如果父类加载器还存在父类加载器，则进一步向上委托，依次递归，请求最后到达顶层的启动类加载器，如果弗雷能够完成类的加载任务，就会成功返回，倘若父类加载器无法完成任务，子类加载器才会尝试自己去加载，这就是双亲委派模式。就是每个儿子都很懒，遇到类加载的活都给它爸爸干，直到爸爸说我也做不来的时候，儿子才会想办法自己去加载。

优势

采用双亲委派模式的好处就是Java类随着它的类加载器一起具备一种带有优先级的层次关系，通过这种层级关系可以避免类的重复加载，当父亲已经加载了该类的时候，就没有必要子类加载器（ClassLoader）再加载一次。其次是考虑到安全因素，Java核心API中定义类型不会被随意替换，假设通过网路传递一个名为java.lang.Integer的类，通过双亲委派的的模式传递到启动类加载器，而启动类加载器在核心Java API发现这个名字类，发现该类已经被加载，并不会重新加载网络传递过来的java.lang.Integer.而之际返回已经加载过的Integer.class，这样便可以防止核心API库被随意篡改。可能你会想，如果我们在calsspath路径下自定义一个名为java.lang.SingInteger?该类并不存在java.lang中，经过双亲委托模式，传递到启动类加载器中，由于父类加载器路径下并没有该类，所以不会加载，

将反向委托给子类加载器，最终会通过系统类加载器加载该类，但是这样做是不允许的，因为java.lang是核心的API包，需要访问权限，强制加载将会报出如下异常。

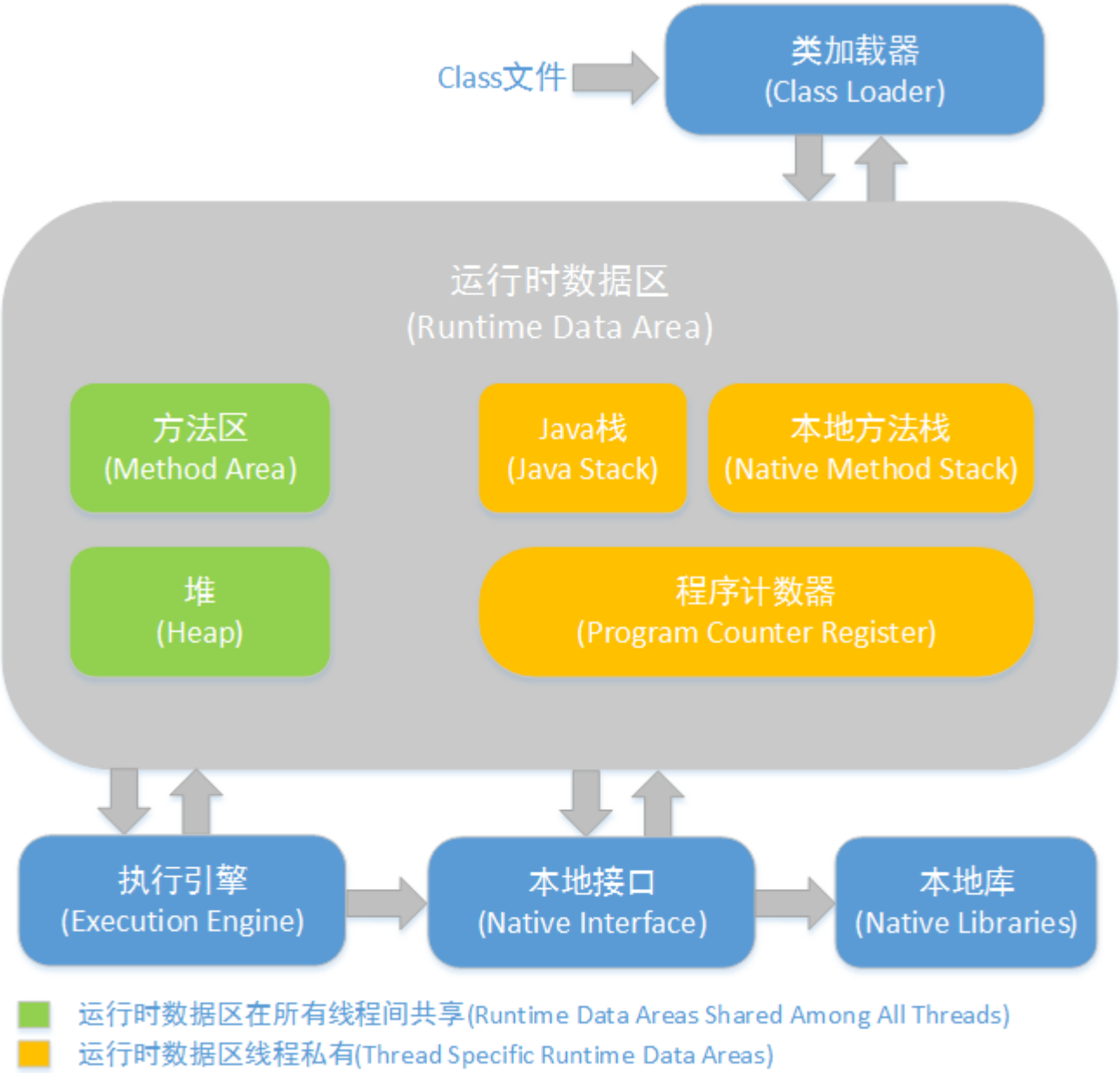
```
Java.lang.SecurityException:Prohibited package name: java.lang
```

类与类加载器

- 在JVM中标识两个Class对象，是否是同一个对象存在的两个必要条件
- 类的完整类名必须一致，包括包名。
- 加载这个ClassLoader（指ClassLoader实例对象）必须相同。

四、内存管理

1.Java内存结构



- **线程共享**：方法区、堆
- **线程私有**：java栈、本地方法栈、程序计数器

1-1.Java堆（Heap）

是Java虚拟机所管理的内存中最大的一块，在虚拟机启动时创建。线程共享，此内存区域的唯一目的就是**存放对象实例**。

1-2.方法区（Method Area）

线程共享，它用于存储已被虚拟机加载的**类信息**、**常量**、**静态变量**、即时编译器**编译后的代码**等数据。

1-3.程序计数器（Program Counter Register）

线程私有，是一块较小的内存空间，它的作用可以看做是当前线程所执行的**字节码的行号指示器**。

1-4.JVM栈（JVM Stacks）

线程私有，生命周期与线程相同。

虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于**存储局部变量表**、**操作栈**、**动态链接**、**方法出口**等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

1-5.本地方法栈（Native Method Stacks）

线程私有，与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是**为虚拟机使用到的Native方法服务**。

PS: Native Method就是一个java调用非java代码的接口

2.JVM的对象分配规则

- 对象优先分配在Eden区**【使用空间】**，如果Eden区没有足够的空间时，虚拟机执行一次Minor GC**【垃圾回收】**。
- 大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。
- 长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1次Minor GC（年轻代收集）那么对象会进入Survivor区，之后每经过一次Minor GC那么对象的年龄加1，知道达到阀值对象进入老年区。
- 动态判断对象的年龄。如果Survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。
- 空间分配担保。每次进行Minor GC时，JVM会计算Survivor区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次Full GC，如果小于检查HandlePromotionFailure设置，如果true则只进行Monitor GC,如果false则进行Full GC。

术语说明

- **Young Generation（新生代）**：分为：Eden区和Survivor区，Survivor区有分为大小相等的From Space和To Space。
- **Old Generation（老年代）**： Tenured区，当 Tenured区空间不够时， JVM 会在Tenured区进行 major collection。
- **Minor GC**：新生代GC，指发生在新生代的垃圾收集动作，因为java对象大多都具备朝生夕死的特性，所以Minor GC非常频繁，一般回收速度也比较快。
- **Major GC**：发生老年代的GC，对整个堆进行GC。出现Major GC，经常会伴随至少一次Minor GC（非绝对）。MajorGC的速度一般比minor GC慢10倍以上。
- **Full GC**：整个虚拟机，包括永久区、新生区和老年区的回收。

五、垃圾回收

了解垃圾回收之前，得先了解JVM是怎么分配内存的，然后识别哪些内存是垃圾需要回收，最后才是用什么方式回收。

Java的内存分配原理与C/C++不同，C/C++每次申请内存时都要malloc进行系统调用，而系统调用发生在内核空间，每次都要中断进行切换，这需要一定的开销，而Java虚拟机是先一次性分配一块较大的空间，然后每次new时都在该空间上进行分配和释放，减少了系统调用的次数，节省了一定的开销，这有点类似于内存池的概念；二是有了这块空间过后，如何进行分配和回收就跟GC机制有关了。

java一般内存申请有两种：**静态内存和动态内存**。很容易理解，编译时就能够确定的内存就是静态内存，即内存是固定的，系统一次性分配，比如int类型变量；动态内存分配就是在程序执行时才知道要分配的存储空间大小，比如java对象的内存空间。根据上面我们知道，java栈、程序计数器、本地方法栈都是线程私有的，线程生就生，线程灭就灭，栈中的栈帧随着方法的结束也会撤销，内存自然就跟着回收了。所以这几个区域的内存分配与回收是确定的，我们不需要管的。但是java堆和方法区则不一样，我们只有在程序运行期间才知道会创建哪些对象，所以这部分内存的分配和回收都是动态的。一般我们所说的垃圾回收也是针对的这一部分。

总之Stack的内存管理是顺序分配的，而且定长，不存在内存回收问题；而Heap 则是为java对象的实例随机分配内存，不定长度，所以存在内存分配和回收的问题；

垃圾检测、回收算法

垃圾收集器一般必须完成两件事：检测出垃圾；回收垃圾。怎么检测出垃圾？一般有以下几种方法：

引用计数法：给一个对象添加引用计数器，每当有个地方引用它，计数器就加1；引用失效就减1。

好了，问题来了，如果我有两个对象A和B，互相引用，除此之外，没有其他任何对象引用它们，实际上这两个对象已经无法访问，即是我们说的垃圾对象。但是互相引用，计数不为0，导致无法回收，所以还有另一种方法：

可达性分析算法：以根集对象为起始点进行搜索，如果有对象不可达的话，即是垃圾对象。这里的根集一般包括java栈中引用的对象、方法区常良池中引用的对象，本地方法中引用的对象等。

该算法的基本思路就是通过一些被称为引用链（GC Roots）的对象作为起点，从这些节点开始向下搜索，搜索走过的路径被称为（Reference Chain)，当一个对象到GC Roots没有任何引用链相连时（即从GC Roots节点到该节点不可达），则证明该对象是不可用的。

在Java中，可作为GC Root的对象包括以下几种：

- 虚拟机栈（栈帧中的本地变量表）中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中常量引用的对象
- 本地方法栈中JNI（即一般说的Native方法）引用的对象

总之，JVM在做垃圾回收的时候，会检查堆中的所有对象是否会被这些根集对象引用，不能够被引用的对象就会被垃圾收集器回收。一般回收算法也有如下几种：

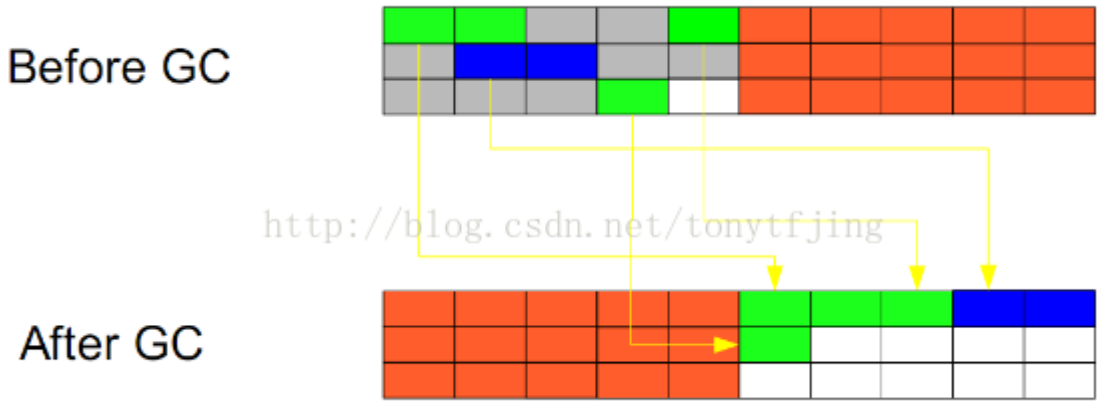
1.标记-清除（Mark-sweep）

算法和名字一样，分为两个阶段：标记和清除。标记所有需要回收的对象，然后统一回收。这是最基础的算法，后续的收集算法都是基于这个算法扩展的。

不足：效率低；标记清除之后会产生大量碎片。效果图如下：

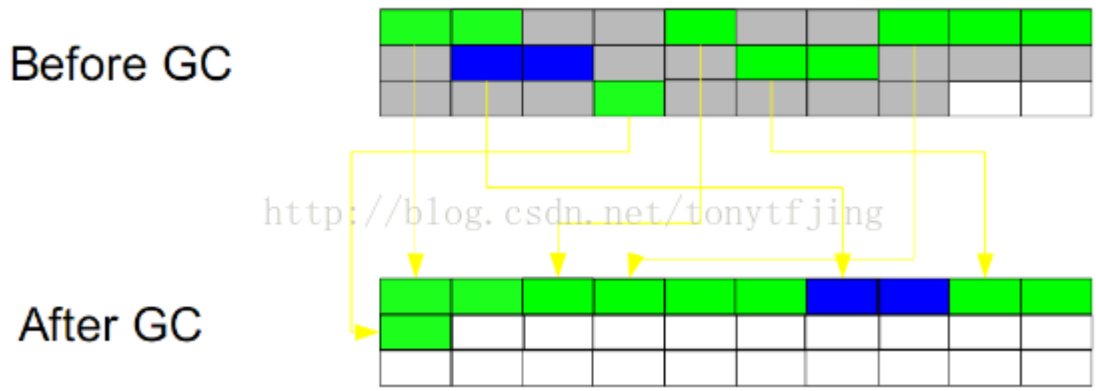
2.复制（Copying）

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。此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。效果图如下：



3.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。效果图如下：



4.分代收集算法

这是当前商业虚拟机常用的垃圾收集算法。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。

为什么要运用分代垃圾回收策略？在java程序运行的过程中，会产生大量的对象，因每个对象所能承担的职责不同所具有的功能不同所以也有着不一样的生命周期，有的对象生命周期较长，比如Http请求中的Session对象，线程，Socket连接等；有的对象生命周期较短，比如String对象，由于其不变类的特性，有的在使用一次后即可回收。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，那么消耗的时间相对会很长，而且对于存活时间较长的对象进行的扫描工作等都是徒劳。因此就需要引入分治的思想，所谓分治的思想就是因地制宜，将对象进行代的划分，把不同生命周期的对象放在不同的代上使用不同的垃圾回收方式。

如何划分？将对象按其生命周期的不同划分成：年轻代(Young Generation)、年老代(Old Generation)、持久代(Permanent Generation)。其中持久代主要存放的是类信息，所以与java对象的回收关系不大，与回收息息相关的是年轻代和年老代。这里有个比喻很形象

年轻代：是所有新对象产生的地方。年轻代被分为3个部分——Enden区和两个Survivor区（From和to）当Eden区被对象填满时，就会执行Minor GC。并把所有存活下来的对象转移到其中一个survivor区（假设为from区）。Minor GC同样会检查存活下来的对象，并把它们转移到另一个survivor区（假设为to区）。这样在一段时间内，总会有一个空的survivor区。经过多次GC周期后，仍然存活下来的对象会被转移到年老代内存空间。通常这是在年轻代有资格提升到年老代前通过设定年龄阈值来完成的。需要注意，Survivor的两个区是对称的，没先后关系，from和to是相对的。

年老代：在年轻代中经历了N次回收后仍然没有被清除的对象，就会被放到年老代中，可以说他们都是久经沙场而不亡的一代，都是生命周期较长的对象。对于年老代和永久代，就不能再采用像年轻代中那样搬移腾挪的回收算法，因为那些对于这些回收战场上的老兵来说是小儿科。通常会在老年代内存被占满时将会触发Full GC,回收整个堆内存。

持久代：用于存放静态文件，比如java类、方法等。持久代对垃圾回收没有显著的影响。

这里之所以最后讲分代，是因为分代里涉及了前面几种算法。

年轻代：涉及了复制算法；

年老代：涉及了“标记-整理（Mark-Sweep）”的算法。

总结JVM垃圾回收：

- 对新生代的对象的收集称为minor GC；
- 对老年代的对象的收集称为full GC;
- 程序中主动调用System.gc()强制执行的GC为full GC；
- 强引用：默认情况下，对象采用的均为强引用；
- 软引用：适用于缓存场景（只有在内存不够用的情况下才会被回收）
- 弱引用：在GC时一定会被GC回收
- 虚引用：用于判断对象是否被GC

垃圾收集算法：

- 标记—清除算法：有两点不足，一个效率问题，标记和清除过程都效率不高；一个是空间问题，标记清除后会产生大量不连续的内存碎片；
- 复制算法：解决了内存碎片问题，但是内存利用率低；

- 标记整理算法：解决了内存碎片问题

分代收集算法：

- 新生代：每次GC时都会有大量对象死去，少量存活，使用复制算法；新生代又分为Eden区、Survivor（Survivor from、Survivor to）大小比例默认为8:1:1；JVM给每个对象定义一个对象年龄计数器，如果对象在Eden出生并经过第一个Minor GC后仍然存活，并且能被Survivor容纳，将被移入Survivor并且年龄设定为1.每熬过一次Minor GC，年龄就加1，当它的年龄到了一定程度（默认15岁，可以通过XX:MaxTenuringThreshold来设定），就会移入老年代；如果Survivor相同年龄所有对象大小的总和大于Survivor的一半，年龄大于等于x的所有对象直接进入老年代，无需等到最大年龄要求。
- 老年代：老年代中的对象存活率高、没有额外空间进行分配，就是用标记—清除或标记—整理算法；大对象可以直接进入老年代，JVM可以配置对象达到阈值后进入老年代的大小（-XX:PretenureSizeThreshold）

垃圾收集器：

- Serial收集器：是最基本、历史最久的收集器，单线程，并且在收集是必须暂停所有的工作线程；
- ParNew收集器：是Serial收集器的多线程版本；
- Parallel Scavenge收集器：新生代收集器，并行的多线程收集器。它的目标是达到一个可控的吞吐量，这样可以高效率的利用CPU时间，尽快完成程序的运算任务，适合在后台运算；
- Serial Old收集器：Serial 收集器的老年代版本，单线程，主要是标记—整理算法来收集垃圾；
- Parallel Old收集器：Parallel Scavenge的老年代版本，多线程，主要是标记—整理算法来收集垃圾；Parallel Old 和 Serial Old 不能同时搭配使用，后者性能较差发挥不出前者的作用；
- CMS收集器：收集器是一种以获取最短回收停顿时间为目标的收集器；基于标记清除算法，并发收集、低停顿、运作过程复杂（初始标记、并发标记、重新标记、并发清除）。CMS收集器有3个缺点：1。对CPU资源非常敏感（占用资源）；2。无法处理浮动垃圾（在并发清除时，用户线程新产生的垃圾叫浮动垃圾），可能出现“Concurrent Mode Failure”失败；3。产生大量内存碎片
- G1收集器：
    - 特点：
        - 分代收集，G1可以不需要其他GC收集器的配合就能独立管理整个堆，采用不同的方式处理新生对象和已经存活一段时间的对象；
        - 空间整合：采用标记整理算法，局部采用复制算法（Region之间），不会有内存碎片，不会因为大对象找不到足够的连续空间而提前触发GC；
        - 可预测的停顿：能够让使用者明确指定一个时间片段内，消耗在垃圾收集上的时间不超过时间范围内；