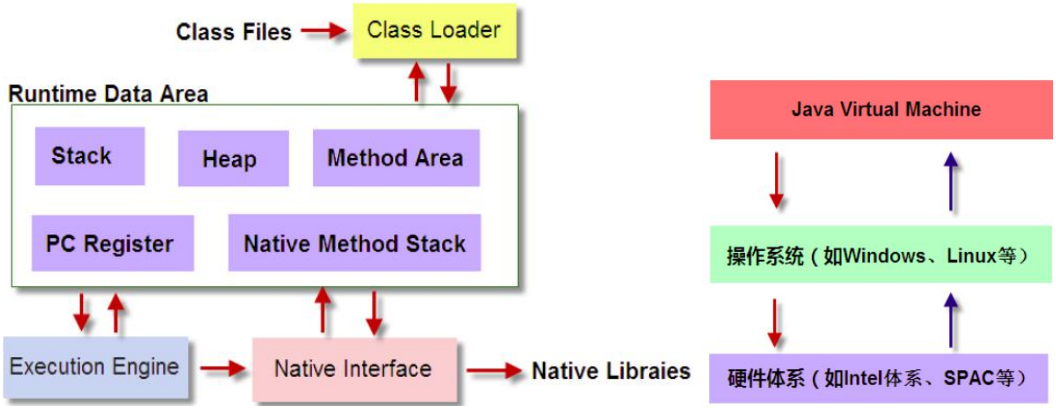
一、JVM

java 编译器面向 jvm，生成 jvm 能理解的以.class 后缀的字节码文件，通过 jvm 将每条指令翻译成不同的机器码，通过特定平台运行。

所以不是 JVM 生成的.class 文件，而是编译器将.java 结尾的源文件编译成.class 结尾的字节码文件。

1、JVM 组成



A.Class Loader:

类加载器本身也是一个类，而它的工作就是**把 class 文件从硬盘读取到内存中。**

类装载方式，有两种：

1.**隐式**装载， 程序在运行过程中当碰到通过 **new** 等方式生成对象时，隐式调用类装载器加载对应的类到 jvm 中，

2.**显式**装载， 通过 **class.forname()**等方法，显式加载需要的类，也就是反射

Java 类的加载是动态的，它并不会一次性将所有类全部加载后再运行，而是保证程序运行

的基础类(像是基类)完全加载到 jvm 中，至于**其他类，则在需要的时候才加载**。这当然就是为了节省内存开销。

B.Execution Engine 执行引擎

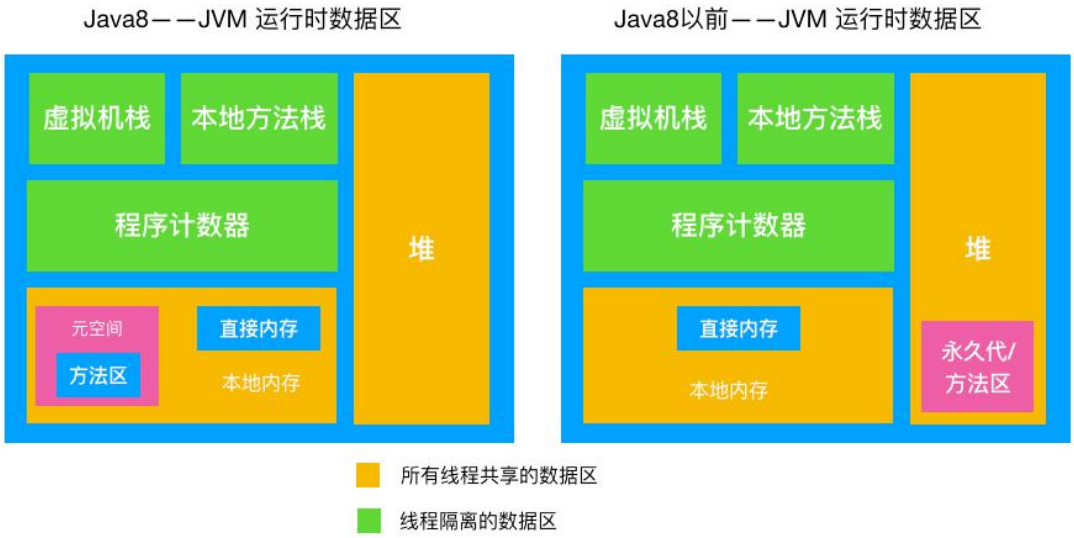
执行引擎也叫做解释器(Interpreter)，**负责解释命令，翻译成不同的机器码，通过特定平台运行。提交操作系统执行。**

C.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。

D.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。



本地栈、虚拟机栈和程序计数器线程私有，不需 GC，本地内存属于堆外内存，GC 管不到，所以 GC 的部分就只有堆。

1.组成结构：

程序计数器(PC 寄存器)

由于在 JVM 中，多线程是通过线程轮流切换来获得 CPU 执行时间的，因此，在任一具体时刻，一个 CPU 的内核只会执行一条线程中的指令，

因此，**为了能够使得每个线程都在线程切换后能够恢复在切换之前的程序执行位置，每个线程都需要有自己独立的程序计数器，并且不能互相被干扰，否则就会影响到程序的正常执行次序。**

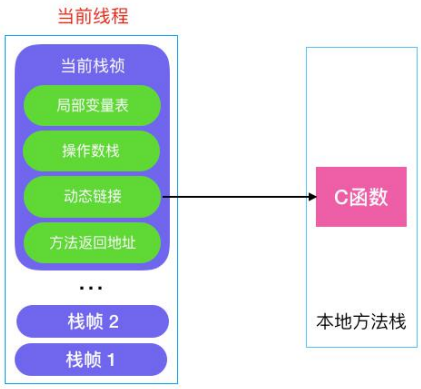
因此，可以这么说，程序计数器是每个线程所私有的。由于程序计数器中存储的数据所占空间的大小不会随程序的执行而发生改变，因此，对于程序计数器是不会发生内存溢出现象(OutOfMemory)的， 那么也就**不需要 GC**。

当执行 java 方法时候，计数器中保存的是字节码文件的行号；当执行 Native 方法时，计数器的值为空。

java 栈

Java 栈中存放的是一个个的栈帧，**每个栈帧对应一个被调用的方法**，在栈帧中 包括局部变量表(Local Variables)、操作数栈(Operand Stack)、动态链接、方法返回地址(Return Address)和一些额外的附加信息。

当线程执行一个方法时，就会随之创建一个对应的栈帧，并将建立的栈帧压栈。当方法执行完毕之后，便会将栈帧出栈，因此也是不需要 GC 的。



局部变量表：基本数据类型值和抽象数据类型的引用，如 student s = new student()

操作数栈：抽象类型的引用先回存入这里，再弹出，存入局部变量表。

动态链接：调用本地方法

方法返回地址

本地方法栈

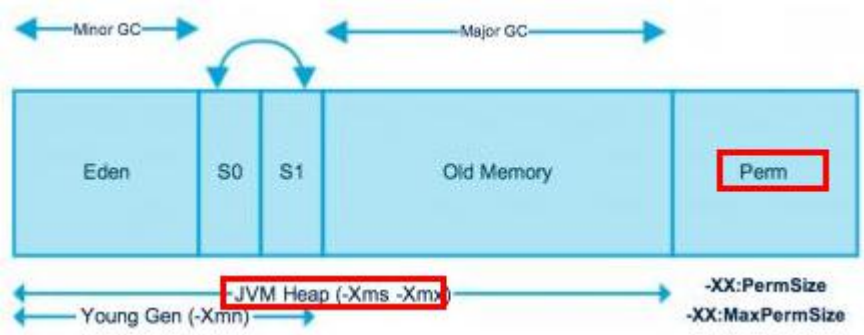
本地方法栈与 Java 栈的作用和原理非常相似。区别只不过是 **Java 栈是为执行 Java 方法服务的，而本地方法栈则是为执行本地方法（Native Method）服务的。**

一 堆

Java 中的堆是**用来存储对象本身的以及数组**（数组引用是存放在 Java 栈中的）。堆是被所有线程共享的，在 JVM 中只有一个堆，在虚拟机启动时创建。

堆内存是所有线程共有的，可以分为两个部分：年轻代和老年代。

下图中的 Perm 代表的是永久代，但是注意永久代并不属于堆内存中的一部分，同时 jdk1.8 之后永久代已经被移除，而是被放入堆外内存，即本地内存中。



默认的:(可通过参数修改以下配比）

Young gen : Old = 1 : 2 // 大对象的创建直接分配到老年代，所以老年代需要大一点。

Eden : from : to = 8 : 1 : 1 // 之所以这个配比是因为 Eden 区将满时，98% 的对象都会被标记不可达，存活对象很少。

还有一个两者都具备的原因就是：就是推迟 Full GC 的到来

那么 GC 也就分为年轻代的 Minor GC 和老年代的 Major GC（Full GC）。

静态变量随着类的加载而加载，随着类的消失而消失，所以静态变量放在方法区。

成员变量的生命周期随着对象的创建而存在，随着对象的消失而消失，所以成员变量应该放在堆中

局部变量的生命周期随着方法的调用而存在，随着方法的调用完毕而消失，所以局部变量放在栈中

方法调用完后会弹栈，表明生命周期结束，那么方法里面的局部变量的生命周期也会结束，所以如果要想使其生命周期延长，最好用 final 声明为常量，就会保存在方法区

状态修饰符分为 final 和 static，声明为 final 后变量不会随着声明周期的结束而消失

本地内存（方法区所在）

1、直接内存

2、元空间（jdk8 实现方法区）

与堆一样，是被线程共享的区域，包含堆中分出来的方法区。

这里要注意一点：针对 HotPot JVM

jdk8 实现方法区的方法是元空间

Jdk8 以前实现方法区的方法是永久代

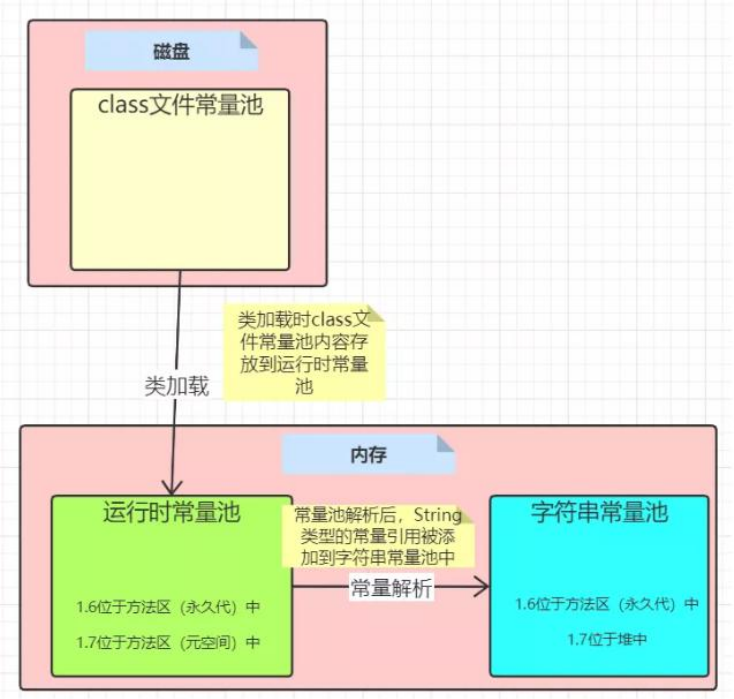
在方法区中，**存储:**

每个类的信息（包括类的名称、方法信息、字段信息）、

静态变量、常量（运行时常量池）

以及编译器编译后的 Class 文件。

- 三种常量池



- Class 文件常量池

位于 class 文件中，而 class 文件位于磁盘。

Class 文件是一组以 8 个字节为基础单位的二进制流，非常紧凑，没有任何分割符，存储的内容几乎全是程序运行必须的参数：

常量池可以理解为 class 文件的资源仓库，存放编译器生成的各种字面量

（literal）和符号引用（区别于直接引用：本地指针、相对偏移量、句柄）

- 字符串常量池

Jdk7 之后存在于堆中，之前存在于堆中的方法区内

本身是一张 hash 表 StringTable，存储的是字符串实例的直接引用。

在解析运行时运行时常量池中的符号引用时，会去查询字符串常量池，确保运行时常量池中解析的直接引用和字符传常量池中的引用是一致的。

- 运行时常量池

存在于方法区内，jdk8 以前方法区存在于堆，之后存在于本地内存的元空间。

jvm 在执行某个类的时候，必须经过**加载、连接、初始化**，而连接又包括验证、准备、解析三个阶段。而当类加载到内存中后，**jvm 就会将 class 常量池中的内容存放到运行时常量池中**，由此可知，**运行时常量池也是每个类都有一个**。在上面我也说了，class 常量池中存的是字面量和符号引用，也就是说他们存的并不是对象的实例，而是对象的符号引用值。而经过解析（resolve）之后，也就是把符号引用替换为直接引用，**解析的过程会去查询全局字符串池，也就是我们上面所说的 StringTable，以保证运行时常量池所引用的字符串与全局字符串池中所引用的是一致的**。

所以简单来说，**运行时常量池就是用来存放 class 常量池中的内容的。**

请注意：Sring 的 intern 方法是将字符串实例引用放入字符串常量池，而不是放入运行时常量池。

实战举例 1

String s1 = new String(“1”) 创建了几个对象？

明确创建对象的几种方法：

1、new　2、反射（newInstance）3、反序列化 4、clone 方法

答：两个对象，

A、首先去字符串常量池查找有没有字符串“1”的引用时，发现字符串常量池中不存在该字符串的引用，故又创建了一个字符串实例，然后将对其的引用存入字符串常量池。

B、New 在堆中开辟的普通对象

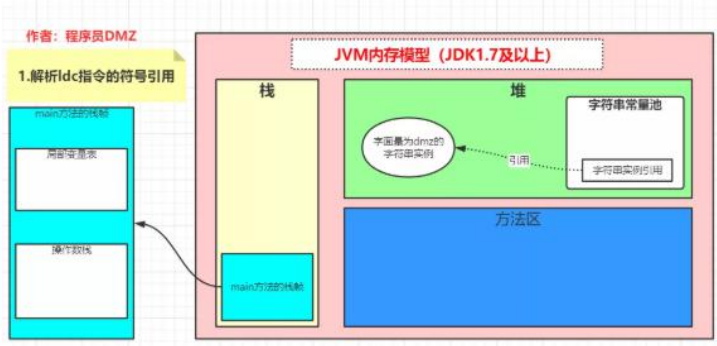
实战举例 2

```
public class Main {
    public static void main(String[] args) {
            String name = "dmz";
    }
}
```

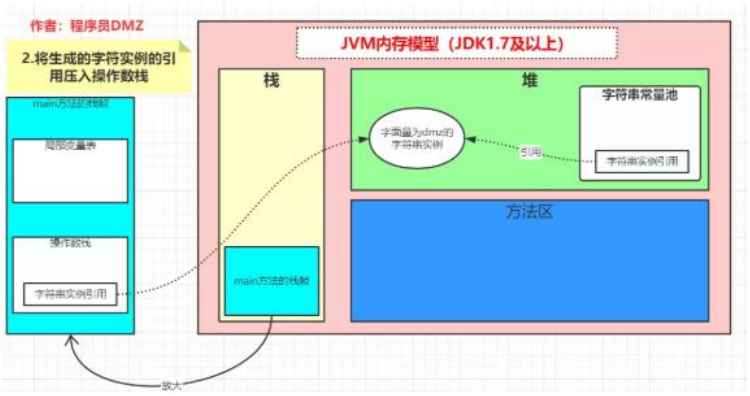
以上 Main 方法在 JVM 中的表现：

1、Main 方法栈帧进栈；

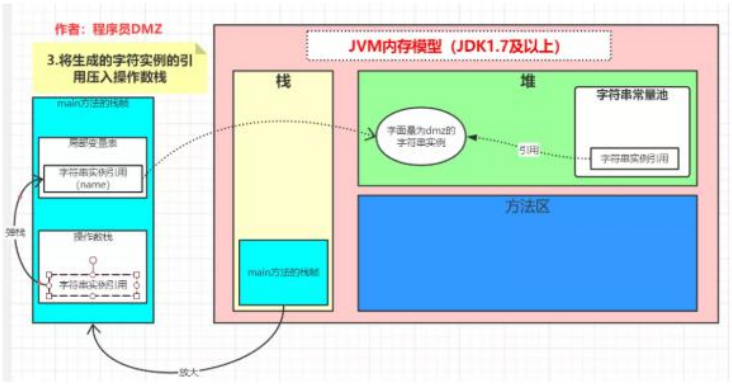
2、先到堆中的字符串常量池中寻找是否存在“dmz”的引用，如果有的话，直接返回，如果没有，则在堆中创建该字符串的实例，然后将引用存入字符串常量池（其实是一张 hash 表），然后返回该引用。



3、将得到的引用压入操作数栈，此时这个字符串实例同时被操作数栈和字符串常量池所引用。



4、操作数栈中的引用弹出，并赋值给局部变量表中的 1 号位置元素，到这一步其实执行完了 String name = "dmz"这行代码。此时局部变量表中储存着一个指向堆中字符串实例的引用，并且这个字符串实例同时也被字符串常量池引用。

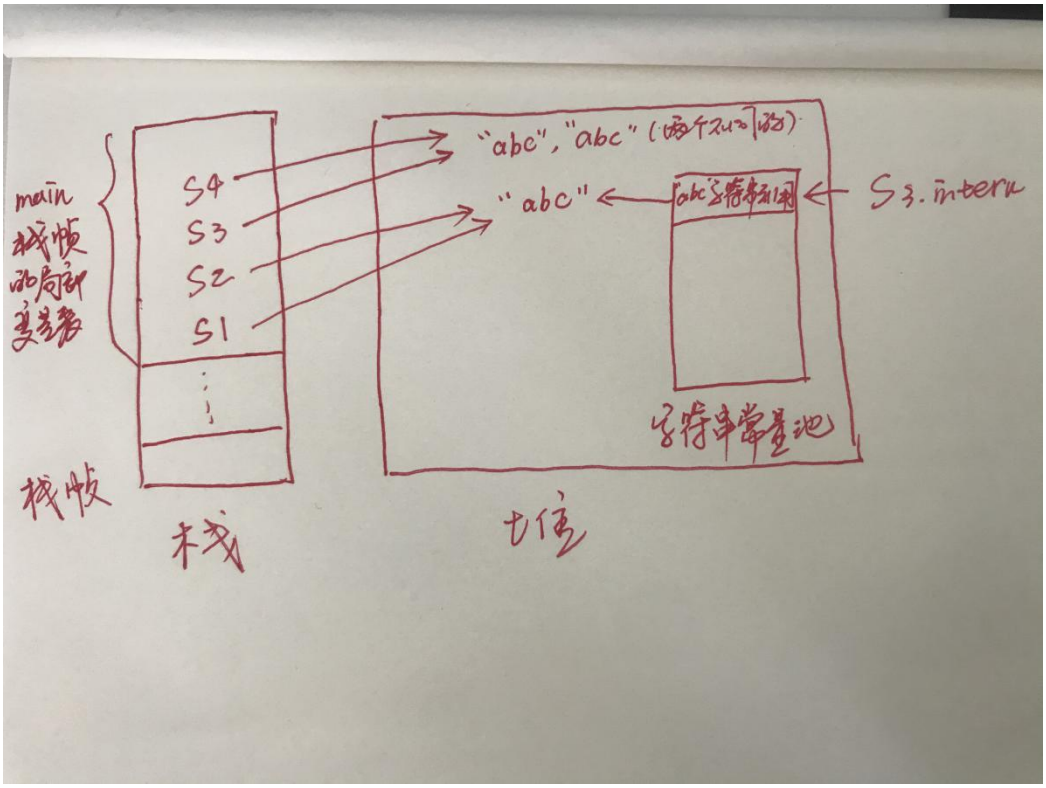


5、方法执行完成，Main 栈帧弹栈

实战举例 3

```java
package com.test5;

public class Test {
    public static void main(String[] args) {
        String s1 = "abc";
        String s2 = "abc";
        System.out.println(s1 == s2);
        String s3 = new String("abc");
        String s4 = new String("abc");
        System.out.println(s1 == s2);
        //基本数据类型和抽象数据类型都会放在局部变量表中（虚拟机栈的栈帧中）
        //String 是抽象数据类型，抽象数据类型存放的是引用，
        System.out.println("s1 == s2? :" + (s1 == s2));
        //System.out.println("s1 == s2?" + s1 == s2);返回结果为 false为什么?
        System.out.println("s1 == s3? :" + (s1 == s3));
        System.out.println("s1 == s3? :" + (s1 == s3.intern()));
        System.out.println("s3 == s4? :" + (s3 == s4));
    }
}
```

```
true
true
s1 == s2? :true
s1 == s3? :false
s1 == s3? :true
s3 == s4? :false
```



实战举例 4

Integer 对象会表现出上面的性质吗？

```
public class Main {
    public static void main(String[] args) {
        String str1 = "abc";
        String str2 = new String( original: "abc");
        System.out.println(str1 == str2);  // False  两者指向不同的地址
        System.out.println(str1 == str2.intern());  // True

        int i0 = 1;
        Integer i1 = new Integer( value: 1);
        System.out.println(i0 == i1);      // True  i0存在vitrual stack的局部变量表中，i1存在堆中，所以i1会隐式转换为字面量
    }
}
```

因为没有整型常量池的概念，而且 i0 是基本数据类型，基本数据类型和抽象数据类型比较，会将抽象数据类型转化为基本数据类型再比较。

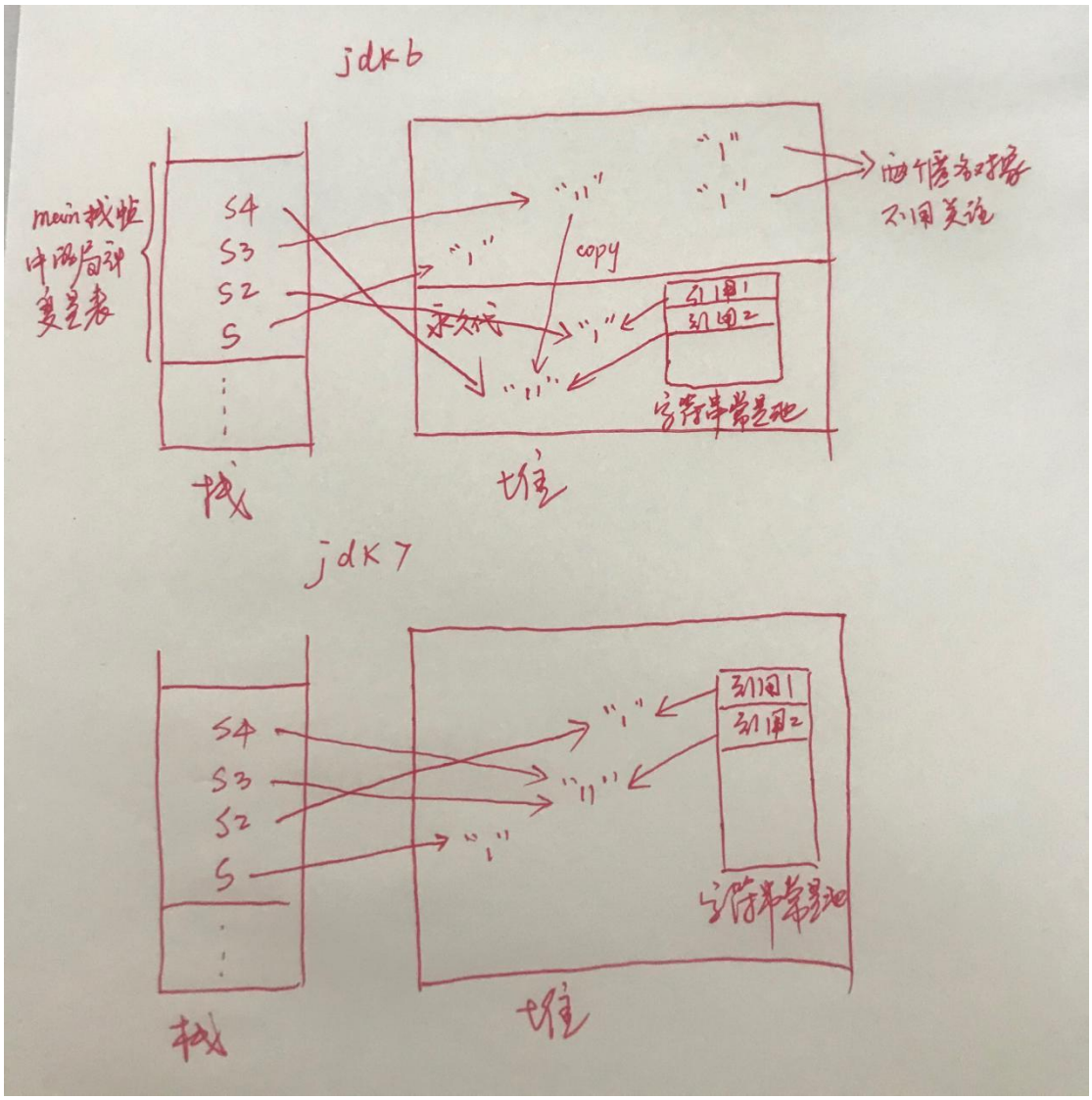
实战举例 5

```
public static void main(String[] args) {
    String s = new String("1");
    s.intern();
    String s2 = "1";
    System.out.println(s == s2);

    String s3 = new String("1") + new String("1");
    s3.intern();
    String s4 = "11";
    System.out.println(s3 == s4);
}
```

Jdk6：false false

Jdk7：false true

s.intern()只是将 s 的字符串引用加入到了字符串常量池中，对 s 本身无任何影响，如果堆中不存在该字符串，那么将字符串常量池中的引用也将指向同一个对象。



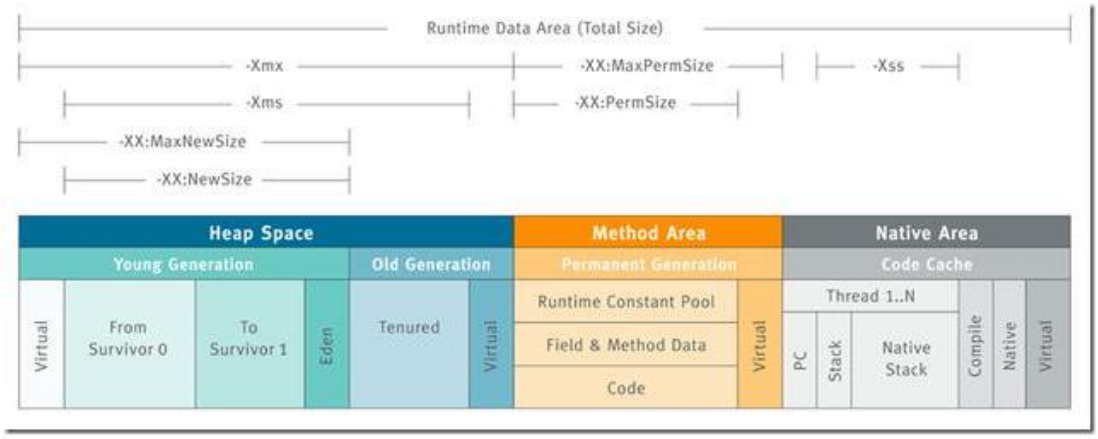

```
public static void main(String[] args) {
    String s = new String("1");
    String sintern = s.intern();
    String s2 = "1";
    System.out.println(sintern == s2);

    String s3 = new String("1") + new String("1");
    String s3intern = s3.intern();
    String s4 = "11";
    System.out.println(s3intern == s4);
}
```

均返回 True，True

```java
// 环境1.7及以上
public class Clazz {
    public static void main(String[] args) {
        String s1 = new StringBuilder().append("ja").append("va1").toString();
        String s2 = s1.intern();
        System.out.println(s1==s2);

        String s5 = "dmz";
        String s3 = new StringBuilder().append("d").append("mz").toString();
        String s4 = s3.intern();
        System.out.println(s3 == s4);

        String s7 = new StringBuilder().append("s").append("pring").toString();
        String s8 = s7.intern();
        String s6 = "spring";
        System.out.println(s7 == s8);
    }
}
```

True false true

2.jvm 内存参数设置



-Xms 设置堆的最小空间大小。

-Xmx 设置堆的最大空间大小。

-Xmn:设置年轻代大小

-XX:NewSize 设置新生代最小空间大小。

-XX:MaxNewSize 设置新生代最大空间大小。

-XX:PermSize 设置永久代最小空间大小。

-XX:MaxPermSize 设置永久代最大空间大小。

-Xss 设置每个线程的堆栈大小

-XX:+UseParallelGC:选择垃圾收集器为并行收集器。此配置仅对年轻代有效。即上述配置下,年轻代使用并发收集,而年老代仍旧使用串行收集。

-XX:ParallelGCThreads=20:配置并行收集器的线程数,即:同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。

典型 JVM 参数配置参考:

```
java-Xmx3550m-Xms3550m-Xmn2g-Xss128k
-XX:ParallelGCThreads=20
-XX:+UseConcMarkSweepGC-XX:+UseParNewGC
```

-Xmx3550m:设置 JVM 最大可用内存为 3550M。

-Xms3550m:设置 JVM 初始内存为 3550m。此值可以设置与-Xmx 相同,以避免每次垃圾回收完成后 JVM 重新分配内存。

-Xmn2g:设置年轻代大小为 2G。整个堆大小=年轻代大小+年老代大小+持久代大小。持久代一般固定大小为 64m,**所以增大年轻代后,将会减小年老代大小**。此值对系统性能影响较大,**官方推荐配置为整个堆的 3/8。**

-Xss128k:设置每个线程的堆栈大小。**JDK5.0 以后每个线程堆栈大小为 1M**,以前每个线程堆栈大小为 256K。更具应用的线程所需内存大 小进行调整。**在相同物理内存下,减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的,不能无限生成,经验值在 3000~5000 左右。**

二、垃圾回收机制

Java 只提供了 new 操作符在堆中开辟空间，却没有 delete 操作符来释放空间，所以就有了自动的垃圾回收机制，自动的检测垃圾，自动的收集垃圾。

垃圾回收算法需要做的基本事情：

发现无用对象

回收被无用对象占用的内存空间，使该空间可被程序再次使用

1、如何检测垃圾

1.1 引用计数法（Reference Counting Collector）

引用计数是垃圾收集器中的早期策略。

此方法中，堆中的每个对象都会添加一个引用计数器。每当一个地方引用这个对象时，计数器值 +1；当引用失效时，计数器值 -1。任何时刻计数值为 0 的对象就是不可能再被使用的。

这种算法无法解决对象之间相互引用的情况。

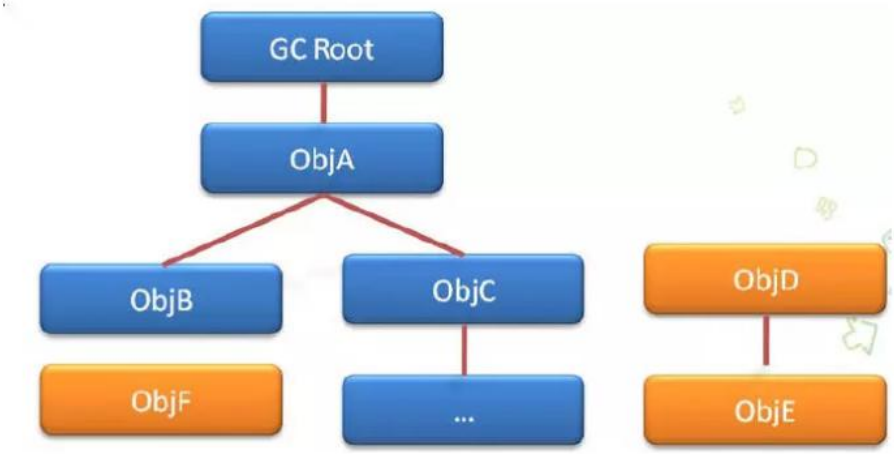
比如对象有一个对子对象的引用，子对象反过来引用父对象，它们的引用计数永远不可能为 0。

```
public class Main {
    public static void main(String[] args) {
        MyObject object1 = new MyObject();
        MyObject object2 = new MyObject();

        object1.object = object2;
        object2.object = object1;

        object1 = null;
        object2 = null;
    }
}
```

最后面两句将 object1 和 object2 赋值为 null，也就是说 object1 和 object2 指向的对象已经不可能再被访问，但是由于它们互相引用对方，导致它们的引用计数器都不为 0，那么垃圾收集器就永远不会回收它们。

1.2 根搜索算法（可达性分析算法）

由于引用计数法存在缺陷，所有现在一般使用根搜索算法。



根搜索算法图解

根搜索算法是从离散数学中的图论引入的，程序把所有的引用关系看作一张图，从一个节点 GC ROOT 开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，**当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即无用的节点。**

如上图中的 ObjF、ObjD、ObjE 通过 GC Root 是无法找到的，所以它们是无用节点。Java 中可作为 GC Root 的对象：

虚拟机栈中引用的对象（局部变量表）

方法区中静态属性引用的对象

方法区中常量引用的对象

本地方法栈中引用的对象（Native 对象）

大部分应该都是局部变量表中引用的对象。

虚拟机栈（栈帧中的本地变量表）中引用的对象

如下代码所示，a 是栈帧中的本地变量，当 a = null 时，由于此时 a 充当了 **GC Root** 的作用，a 与原来指向的实例 **new Test()** 断开了连接，所以对象会被回收。

```
public class Test {
    public static  void main(String[] args) {
    Test a = new Test();
    a = null;
    }
}
```

方法区中类静态属性引用的对象

如下代码所示，当栈帧中的本地变量 a = null 时，由于 a 原来指向的对象与 GC Root (变量 a) 断开了连接，所以 a 原来指向的对象会被回收，而由于我们给 s 赋值了变量的引用，s 在此时是类静态属性引用，充当了 GC Root 的作用，它指向的对象依然存活!

```
public class Test {
    public static Test s;
    public static  void main(String[] args) {
        Test a = new Test();
        a.s = new Test();
        a = null;
    }
}
```

方法区中常量引用的对象

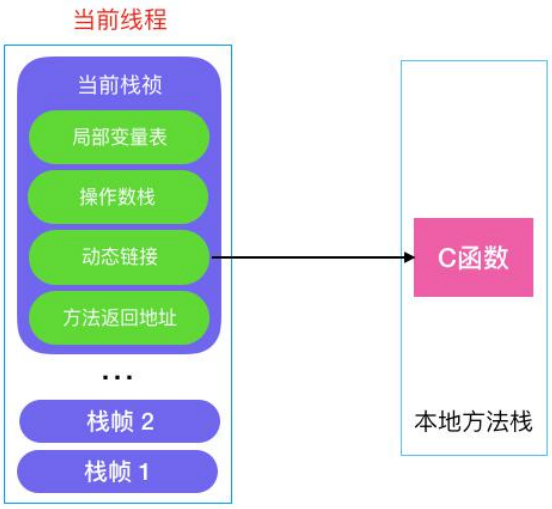
如下代码所示，常量 s 指向的对象并不会因为 a 指向的对象被回收而回收

```
public class Test {
    public static final Test s = new Test();
        public static  void main(String[] args) {
        Test a = new Test();
            a = null;
        }
}
```

本地方法栈中 JNI 引用的对象

这是简单给不清楚本地方法为何物的童鞋简单解释一下：所谓本地方法就是一个 java 调用非 java 代码的接口，该方法并非 Java 实现的，可能由 C 或 Python等其他语言实现的， Java 通过 JNI 来调用本地方法， 而本地方法是以库文件的形式存放的（在 WINDOWS 平台上是 DLL 文件形式，在 UNIX 机器上是 SO 文件形式）。通过调用本地的库文件的内部方法，使 JAVA 可以实现和本地机器的紧密联系，调用系统级的各接口方法，还是不明白？见文末参考，对本地方法定义与使用有详细介绍。

当调用 Java 方法时，虚拟机会创建一个栈桢并压入 Java 栈，而当它调用的是本地方法时，虚拟机会保持 Java 栈不变，不会在 Java 栈祯中压入新的祯，虚拟机只是简单地动态连接并直接调用指定的本地方法。




```
JNIEXPORT void JNICALL Java_com_pecuyu_jnirefdemo_MainActivity_newStringNative(JNIEnv *env, jobject instance, jstring
...
    // 缓存String的class
    jclass jc = (*env)->FindClass(env, STRING_PATH);
}
```

如上代码所示，当 java 调用以上本地方法时，jc 会被本地方法栈压入栈中，jc 就是我们说的本地方法栈中 JNI 的对象引用，因此只会在此本地方法执行完成后才会被释放。

小结：无论是引用计数法还是跟搜索法，都是为了找到可回收的对象（内存块），所以搜索的也都是对象，对象在内存中，调用它的时候就需要引用，这些引用大部分都存在于 virtual stack 中局部变量表中，当然你如果定义对象的引用为静态属性或常量属性，那这些引用将会存在方法区。还有一种特殊的就是本地方法栈中引用的对象。

所以，四种对象分别来自栈的两种和来自方法区的两种。

2、垃圾收集算法

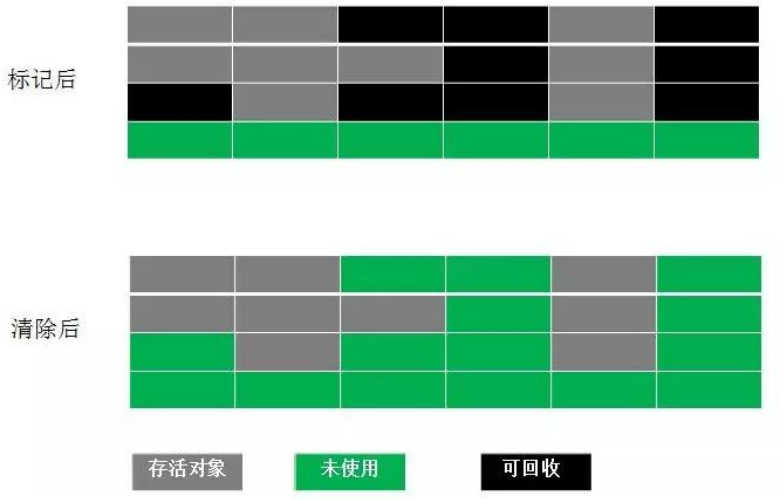
在确定了哪些垃圾可以被回收后，垃圾收集器要做的就是进行垃圾的回收，有下面的几中算法：

2.1 标记-清除（Mark-Sweep）算法

标记-清除算法分为两个阶段：

标记阶段：标记出需要被回收的对象。

清除阶段：回收被标记的可回收对象的内部空间。



标记-清除算法实现较容易，不需要移动对象，但是存在较严重的问题：

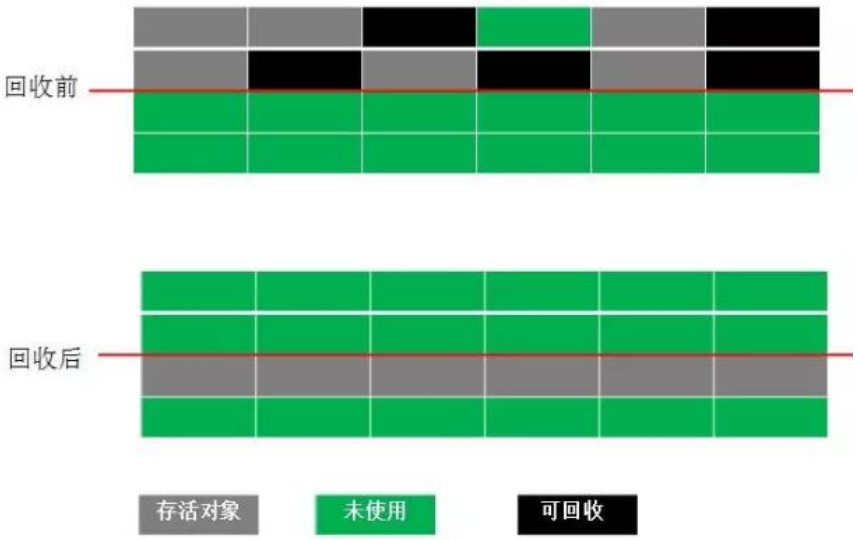
算法过程需要**暂停整个应用，效率不高**。

标记清除后会**产生大量不连续的内存碎片**，碎片太多可能会导致后续过程中需要为大对象分配空间时无法找到足够的空间而提前触发新的一次垃圾收集动作。

2.2 复制（Copying）算法

为了解决标志-清除算法的缺陷，由此有了复制算法。

复制算法将可用内存分为两块，每次只用其中一块，当这一块内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已经使用过的内存空间一次性清理掉。



小结：

优点：实现简单，不易产生内存碎片，每次只需要对半个区进行内存回收。

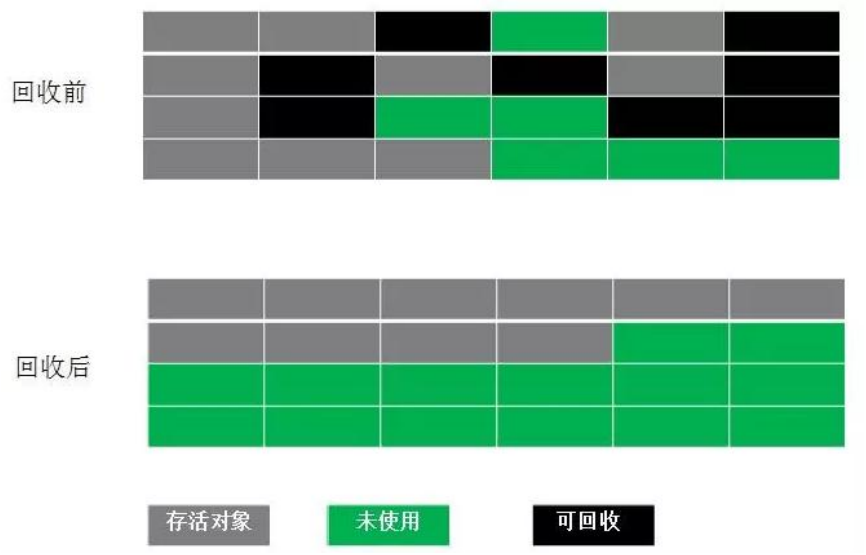
缺点：**内存空间缩减为原来的一半**；算法的效率和存活对象的数目有关，**存活对象越多，效率越低。**

2.3 标记-整理（Mark-Compact）算法

为了更充分利用内存空间，提出了标记-整理算法。

此算法结合了“标记-清除”和“复制”两个算法的优点。

该算法标记阶段和“标志-清除”算法一样，但是在完成标记之后，它不是直接清理可回收对象，而是将存活对象都向一端移动，然后清理掉端边界以外的内存。



总结前三种算法

对象分为可回收、存活对象。

1、标记清除算法：直接标记可回收对象，直接清除，产生很多碎片空间。

2、复制算法：为解决上述问题，每次只用一半内存空间，清除时，将还存活的对象复制到未使用的内存空间，然后直接清除另一半内存，这样的话，不产生碎片，但是效率取决于存活对象的多少。

3、标记-整理算法：结合上面两种，将已存活对象向一端移动，将存活对象聚集在一起，然后清理掉边界以外的内存。

2.4 分代收集（Generational Collection）算法

分代收集算法是目前大部分 JVM 的垃圾收集器采用的算法。

老年代的特点是每次垃圾收集时只有少量对象需要被回收，

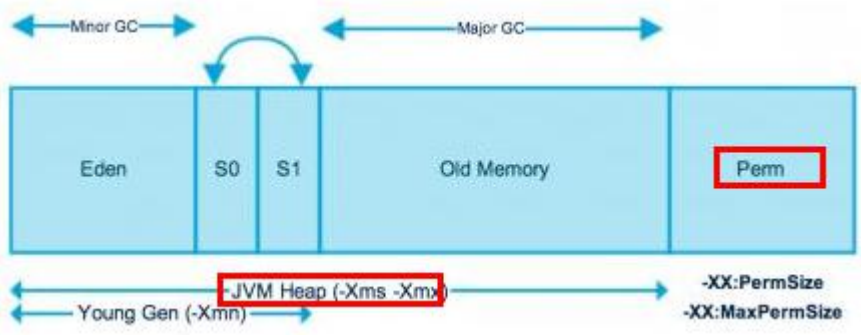
而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，**那么就可以根据不同代的特点采取最适合的收集算法。**

区域划分：

年轻代（Young Generation）

所有新生成的对象（除大对象需要分配连续大内存）首先都是放在年轻代的，大部分对象在 Eden 区中生成。

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，故年轻代的存活对像往往比较少，故采取的是复制算法。



1、当 Eden 区将满时，出发 Minor GC，回收时先将 eden 区存活对象复制到一个 survivor0 区，**所有存活的对象的年龄（经历了 Minor GC 的次数）加 1**，然后清空 eden 区；

2、当 Eden 区再度将满时，则将 eden 区和 survivor0 区存活对象复制到另一个 survivor1 区，所有存活对象年龄加 1，然后清空 eden 和这个 survivor0 区；

重复 2，每次 Eden 区将满时，将 Eden 区和另一个 survivor 区的存活对象存入空 survivor 区。**在这个过程中当有对象的年龄达到了给定阈值，移入老年代。**

还有一种情况也会让对象晋升到老年代，即在 S0（或 S1） 区相同年龄的对象大小之和大于 S0（或 S1）空间一半以上时，则年龄大于等于该年龄的对象也会晋升到老年代。

也就是前者是因为年龄太多，后者是因为空间不够了。

年老代（Old Generation）

在年轻代中经历了 N 次 Minor GC 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。**

内存比新生代也大很多(**大概比例是 1:2**)，**当老年代内存满时触发 Major GC 即 Full GC**，同时挥手新生代和老年代，Full GC 发生频率比较低，老年代对象存活时间比较长，存活率标记高。

但是一旦发生，就会 Stop The World，即在 GC 期间，只能有垃圾回收线程在工作，其他线程都需挂起。采取的是标记整理算法。

由于 Full GC（或Minor GC） 会影响性能，所以我们要在一个合适的时间点发起 GC，这个时间点被称为 Safe Point，这个时间点的选定既不能太少以让 GC 时间太长导致程序过长时间卡顿，也不能过于频繁以至于过分增大运行时的负荷。一般当线程在这个时间点上状态是可以确定的，如确定 GC Root 的信息等，可以使 JVM 开始安全地 GC。Safe Point 主要指的是以下特定位置：

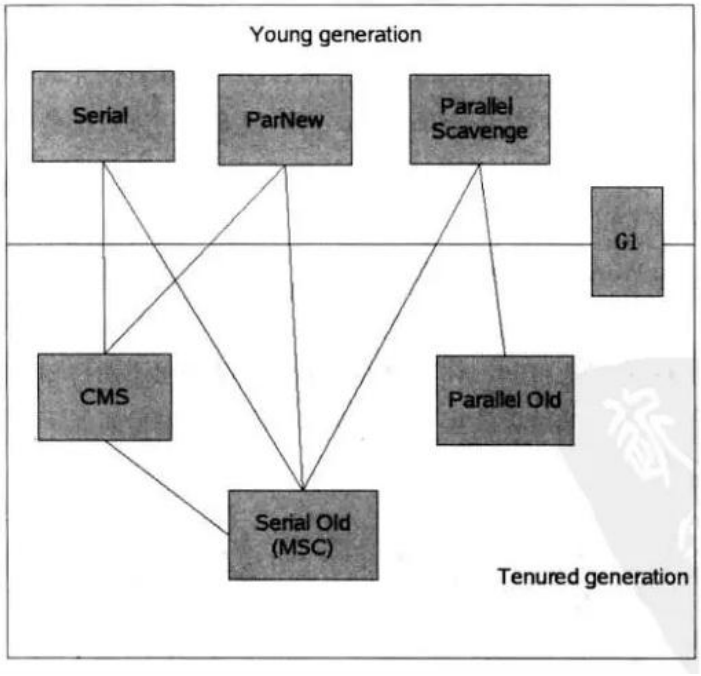
- 循环的末尾
- 方法返回前
- 调用方法的 call 之后
- 抛出异常的位置 另外需要注意的是由于新生代的特点（大部分对象经过 Minor GC后会消亡）， Minor GC 用的是复制算法，而在老生代由于对象比较多，占用的空间较大，使用复制算法会有较大开销（复制算法在对象存活率较高时要进行多次复制操作，同时浪费一半空间）所以根据老生代特点，在老年代进行的 GC 一般采用的是标记整理法来进行回收。

3、垃圾收集器（GC）

不同虚拟机所提供的垃圾收集器可能会有很大差别，下面的例子是 HotSpot。

新生代收集器使用的收集器：Serial、PraNew、Parallel Scavenge。

老年代收集器使用的收集器：Serial Old、Parallel Old、CMS。



Serial 收集器（复制算法)

新生代单线程收集器，标记和清理都是单线程，优点是简单高效。

Serial Old 收集器(标记-整理算法)

老年代单线程收集器，Serial 收集器的老年代版本。

ParNew 收集器(停止-复制算法)

新生代收集器，可以认为是 Serial 收集器的多线程版本，在多核 CPU 环境下有着比 Serial 更好的表现。

Parallel Scavenge 收集器(停止-复制算法)

并行收集器，追求高吞吐量，高效利用 CPU。吞吐量一般为 99%， 吞吐量 = 用户线程时间 /(用户线程时间 + GC 线程时间)。适合后台应用等对交互相应要求不高的场景。

Parallel Old 收集器(停止-复制算法)

Parallel Scavenge 收集器的老年代版本，并行收集器，吞吐量优先。

CMS(Concurrent Mark Sweep) 收集器（标记-清理算法）

高并发、低停顿，追求最短 GC 回收停顿时间，cpu 占用比较高，响应时间快，停顿时间短，多核 cpu 追求高响应时间的选择。

根据对象的生命周期的不同将内存划分为几块 ,然后根据各块的特点采用最适当的收集算法。

大批对象死去、少量对象存活的（新生代），使用复制算法，复制成本低；

对象存活率高、没有额外空间进行分配担保的（老年代），采用标记-清理算法或者标记-整理算法。

4、堆外内存的回收

JVM 启动时分配的内存，称为堆内存，与之相对的，在代码中还可以使用堆外内存，比如 Netty，广泛使用了堆外内存，但是这部分的内存并不归 JVM 管理，GC 算法并不会对它们进行回收，所以在使用堆外内存时，要格外小心，防止内存一直得不到释放，造成线上故障。

https://www.jianshu.com/p/35cf0f348275