JVM探究

什么是JVM

JVM是Java Virtual Machine（Java虚拟机）的缩写，是用来执行java字节码(二进制的形式)的虚拟计算机。包括一套字节码指令集、一组寄存器、一个栈、一个垃圾回收，堆 和 一个存储方法域。JVM 是运行在操作系统之上的，它与硬件没有直接的交互。

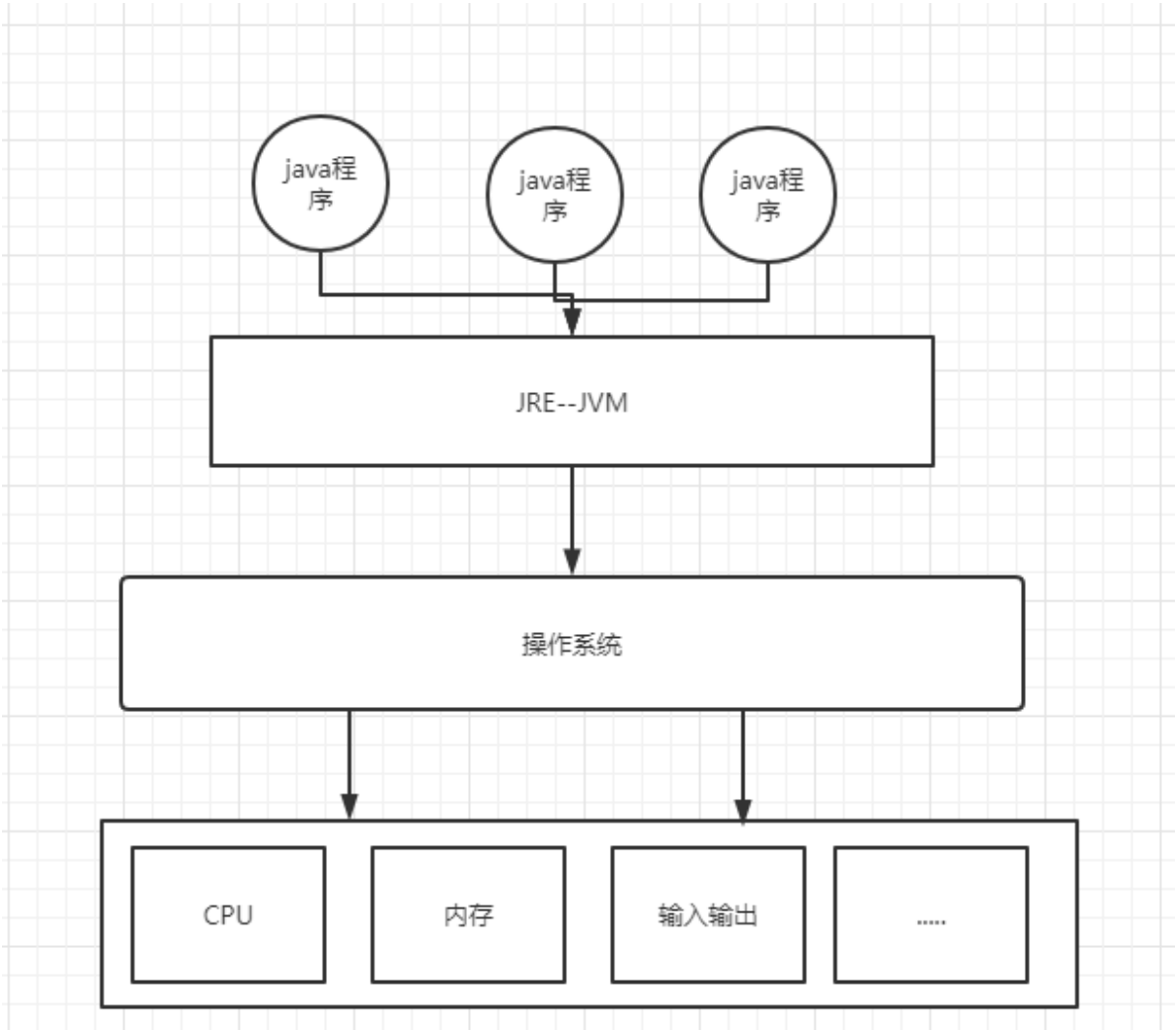
概述

Java虚拟机有自己完善的硬件架构，如处理器、堆栈等，还具有相应的指令系统。

Java虚拟机本质上就是一个程序，当它在命令行上启动的时候，就开始执行保存在某字节码文件中的指令。Java语言的可移植性正是建立在Java虚拟机的基础上。任何平台只要装有针对于该平台的Java虚拟机，字节码文件（.class）就可以在该平台上运行。这就是“一次编译，多次运行”。

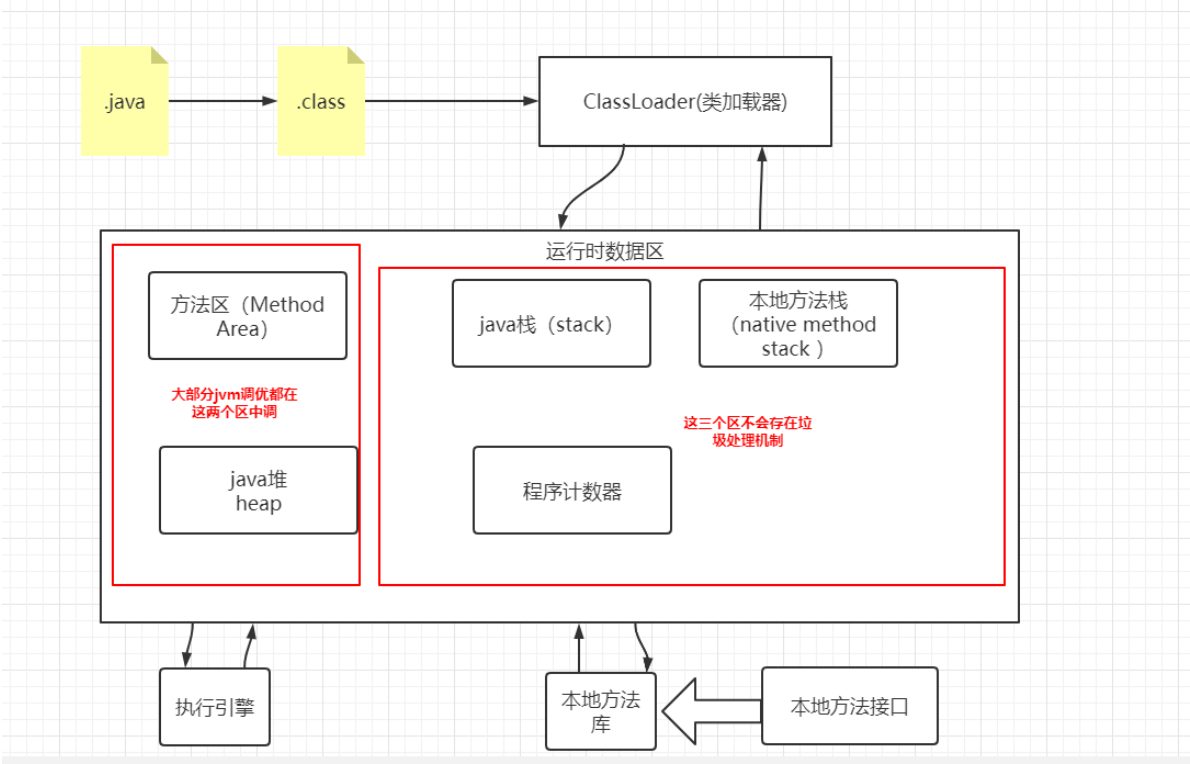
Java虚拟机不仅是一种跨平台的软件，而且是一种新的网络计算平台。该平台包括许多相关的技术，如符合开放接口标准的各种API、优化技术等。Java技术使同一种应用可以运行在不同的平台上。Java平台可分为两部分，即Java虚拟机（Java virtual machine，JVM）和Java API类库。

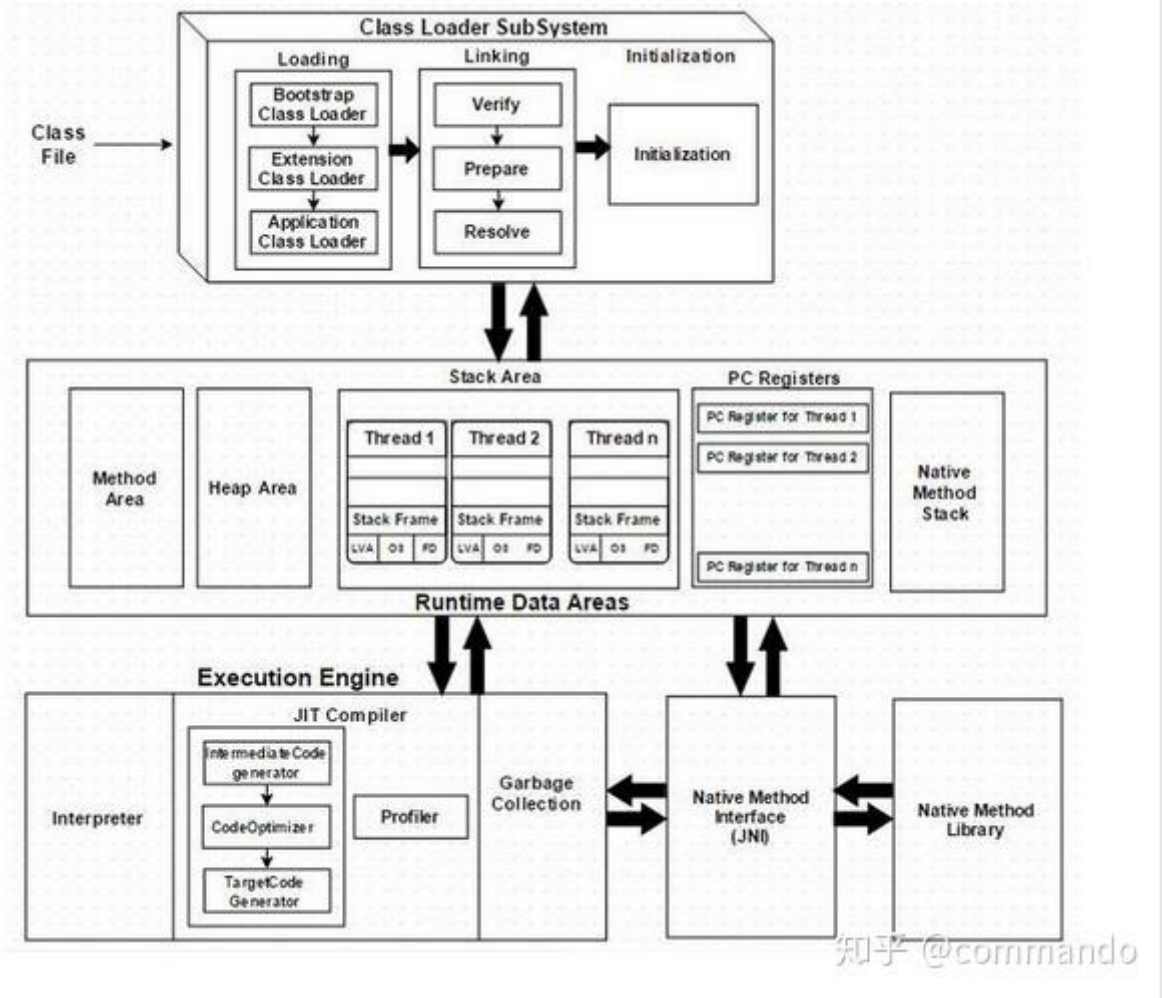
JVM的位置



JVM是运行在操作系统之上的，它与硬件没有直接的交互。

JVM的体系结构

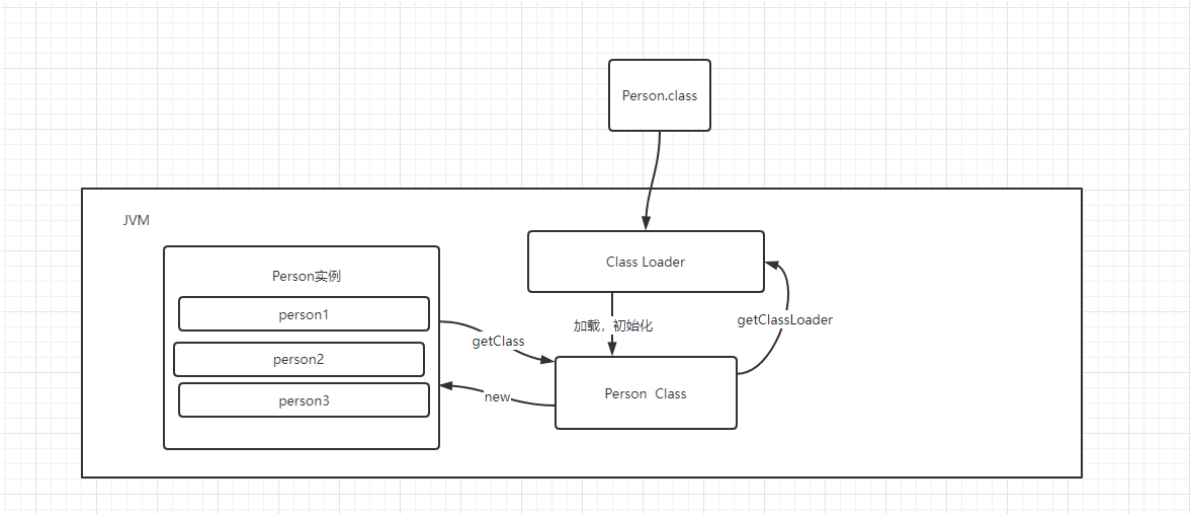




类加载器

作用

加载Class文件，（例如Pperson person = new Person()，new出来的引用在栈中，而具体的示例在堆中）



类加载器负责加载所有的类，其为所有被载入内存中的类生成一个java.lang.Class实例对象。一旦一个类被加载如JVM中，同一个类就不会被再次载入了。正如一个对象有一个唯一的标识一样，一个载入JVM的类也有一个唯一的标识。在Java中，一个类用其全限定类名（包括包名和类名）作为标识；但在JVM中，一个类用其全限定类名和其类加载器作为其唯一标识。

1、虚拟机自带的加载器

2、启动类加载器（根加载器）BootstrapClassLoader

3、扩展类加载器ExtClassLoader

4、程序加载器AppClassLoader

类加载器机制

JVM类加载器机制主要有3种：

- 全盘负责：所谓全盘负责，就是当一个类加载器负责加载某个Class时，该Class所依赖和引用其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入。
- 双亲委派：所谓的双亲委派，则是先让父类加载器试图加载该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。通俗的讲，就是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加载器，依次递归，如果父加载器可以完成类加载任务，就成功返回；只有父加载器无法完成此加载任务时，才自己去加载。
- 缓存机制。缓存机制将会保证所有加载过的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，存入缓冲区中。这就是为很么修改了Class后，必须重新启动JVM，程序所做的修改才会生效的原因。

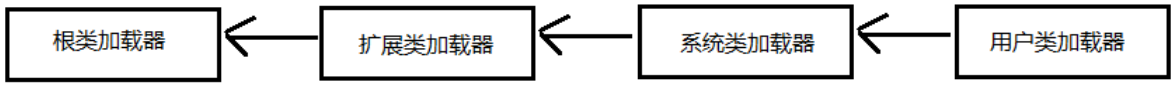
双亲委派机制

1、类加载器收到类加载的请求!

2、将这个请求向上委托给父类加载器去完成，一直向上委托，直到启动类加载器

3、启动加载器检查是否能够加载当前这个类，能加载就结束，使用当前的加载器，否则，抛出异常，通知子加载器进行加载

4、重复步骤3




```
public class Person {

    public static void main(String[] args) {

        //类是抽象的，对象是具体的
        Person person1 = new Person();
        Person person2 = new Person();
        Person person3 = new Person();
        System.out.println(person1.hashCode());//356573597
        System.out.println(person2.hashCode());//1735600054
        System.out.println(person3.hashCode());//21685669不同的对象指向不同的
hashCode

        Class<? extends Person> aClass1 = person1.getClass();
        Class<? extends Person> aClass2 = person2.getClass();
        System.out.println(aClass1.hashCode());//1956725890
        System.out.println(aClass2.hashCode());//1956725890指向的是同一个内存地址

        ClassLoader classLoader = aClass1.getClassLoader();//AppClassLoader
        System.out.println(classLoader);

        System.out.println(classLoader.getParent());//ExtClassLoader

        System.out.println(classLoader.getParent().getParent());//null,java调用不
到
    }
}
```

Native

定义

简单来讲，**一个Native Method就是一个java调用非java代码的接口**，一个Native Method 是这样一个java方法：该方法的实现由非Java语言实现，比如C。这个特征并非java特有，很多其他的编程语言都有这一机制，比如在C++ 中，你可以用extern “C” 告知C++ 编译器去调用一个C的函数。 在定义一个native method时，并不提供实现体（有些像定义一个Java interface），因为其实现体是由非java语言在外面实现的。 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。 标识符native可以与其他所有的java标识符连用，但是abstract除外。

```
/**
 * 本地方法
 */
public  class IHaveNatives {

    //abstract 没有方法体
    public abstract void abstractMethod(int x);

    //native 和 abstract不能共存，native是有方法体的，由C语言来实现
    public native void Native1(int x);
```

```
    /**
    使用native标识为本地方法接口
    */
    native static public long Native2();

    native synchronized private float Native3(Object o);

    native void Native4(int[] array) throws Exception;

}

public class NativeDemo {
    public static void main(String[] args) {
        new Thread(() -> {

        }).start();
    }

    /**
     *  native : 凡是带了native关键字的，就说明光是Java的作用范围达不到了，需要去调用底层C语言的库
     *  进入本地方法栈调用本地方法接口 JNI（java native interface）
     *  作用：扩展java的使用，融合不同的语言为java所用
     */

    private native void start0();
}
```

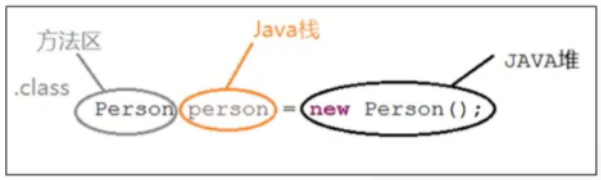
PC寄存器

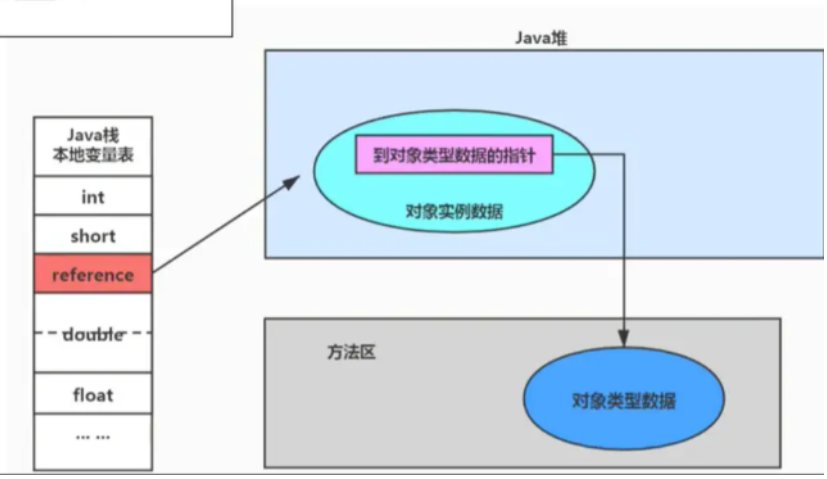
程序计数器：Program Counter Register

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（用来存储指向像一条指令的地址，也即将要执行的指令代码），在执行引擎读取下一条指令，是一个非常小的内存空间，几乎可以忽略不计

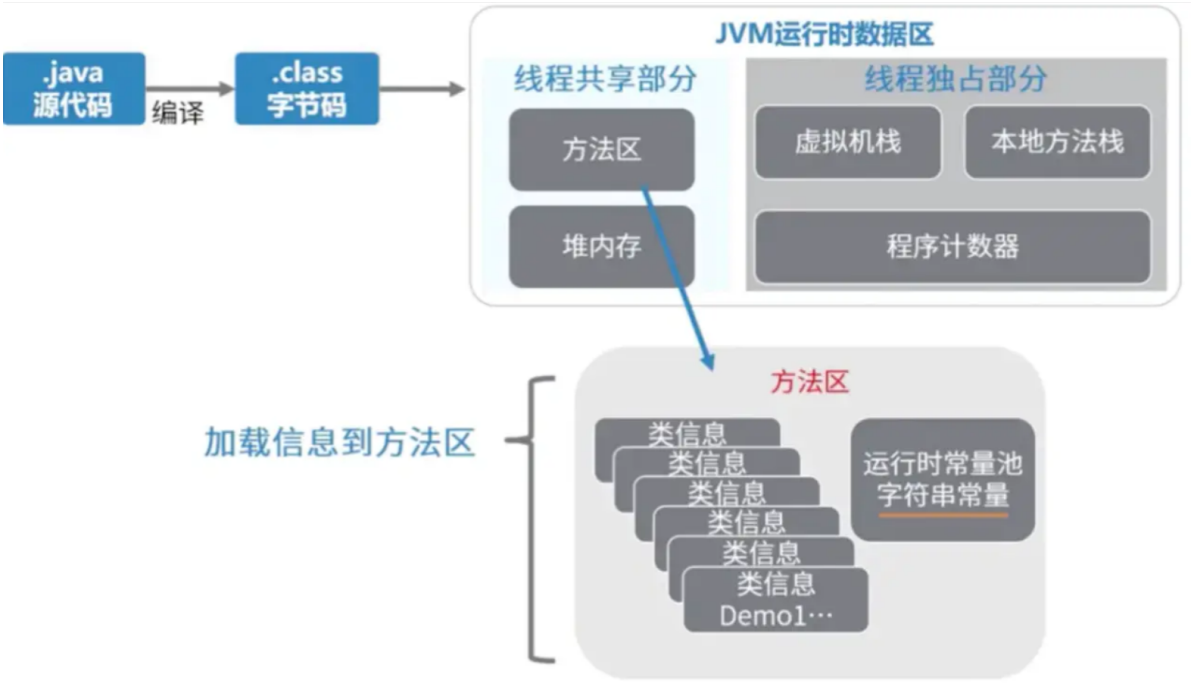
方法区

堆、栈、方法区的交互关系





方法区在运行时数据区中的位置



Method Area方法区：

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，此区域属于共享区间;

静态变量、常量、类信息(构造方法、接口定义)、运行时的常量池存在方法区中，但是实例变量存在堆内存中，和方法区无关。

方法区的垃圾回收

有些人认为方法区（如Hotspot，虚拟机中的元空间或者永久代）是没有垃圾收集行为的，其实不然。《Java 虚拟机规范》对方法区的约束是非常宽松的，提到过可以不要求虚拟机在方法区中实现垃圾收集。事实上也确实有未实现或未能完整实现方法区类型卸载的收集器存在（如 JDK11 时期的 2GC 收集器就不支持类卸载）。 一般来说这个区域的回收效果比较难令人满意，尤其是类型的卸载，条件相当苛刻。但是这部分区域的回收有时又确实是必要的。以前 Sun 公司的 Bug 列表中，曾出现过的若干个严重的 Bug 就是由于低版本的 Hotspot 虚拟机对此区域未完全回收而导致内存泄漏。　方法区的垃圾收集主要回收两部分内容：常量池中废弃的常量和不再使用的类型

- 先来说说方法区内常量池之中主要存放的两大类常量：字面量和符号引用。 字面量比较接近Java语言层次的常量概念，如文本字符串、被声明为final的常量值等。而符号引用则属于编译原理方面的概念，包括下面三类常量：
    - 1、类和接口的全限定名
    - 2、字段的名称和描述符
    - 3、方法的名称和描述符
- HotSpot虚拟机对常量池的回收策略是很明确的，只要常量池中的常量没有被任何地方引用，就可以被回收。
- 回收废弃常量与回收Java堆中的对象非常类似。
- 判定一个常量是否“废弃”还是相对简单，而要判定一个类型是否属于“不再被使用的类”的条件就比较苛刻了。需要同时满足下面三个条件：
    - 该类所有的实例都已经被回收，也就是Java堆中不存在该类及其任何派生子类的实例。
    - 加载该类的类加载器已经被回收，这个条件除非是经过精心设计的可替换类加载器的场景，如OSGi、JSP的重加载等，否则通常是很难达成的。

    - 该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。
- Java虚拟机被允许对满足上述三个条件的无用类进行回收，这里说的仅仅是“被允许”，而并不是和对象一样，没有引用了就必然会回收。关于是否要对类型进行回收，HotSpot虚拟机提供了一Xnoclassgc 参数进行控制，还可以使用一verbose：class以及一XX： +TraceClass一Loading、一XX：+TraceClassUnLoading查 看类加载和卸载信息
- 在大量使用反射、动态代理、CGLib等字节码框架，动态生成JSP以及oSGi这类频繁自定义类加载器的场景中，通常都需要Java虚拟机具备类型卸载的能力，以保证不会对方法区造成过大的内存压力。

三种JVM

- SUN公司：`Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)`大多数用的
- BEA公司：`JRocjkit`
- IBM：`J9 VM`

堆（Heap）

一个JVM只有一个堆内存，堆内存的大小是可以调节的。

类加载器读取了类文件后，一般会把什么东西放到堆中呢？类、方法、常量、变量，会保存所有引用类型的真实对象。栈中一般是对象和方法的引用。

堆内存中还要细分为三个区域：

- 新生区 Young/New
- 养老区 Old
- 永久区 Perm

GC垃圾回收主要是在伊甸园区和养老区回收，其中过渡的幸存区和永久区不会被回收。

加入内存满了，发生OOM，堆内存不够了。java.lang.OutOfMemoryError: Java heap space

JDK8以后，永久存储去改名为元空间

新生区

- 类：诞生和成长的地方，甚至是死亡；
- 伊甸园区 ： 所有对象都是在伊甸园区new出来的
- 幸存者区
    - 0区
    - 1区

老年区

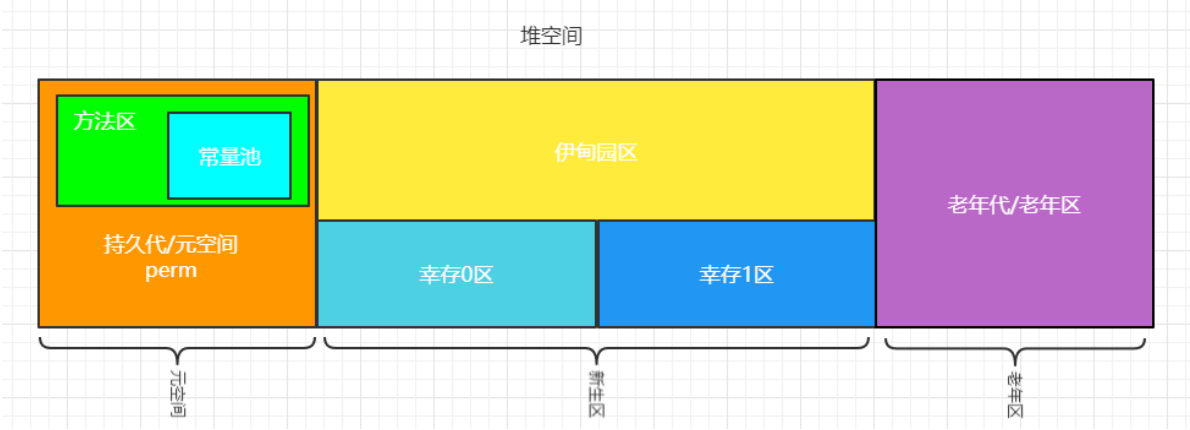
老年区是在对象经历新生区后，实例没有被回收的对象实例，如果老年区也满了会触发重量级的垃圾回收。（99%的对象都是临时对象，所以老年区对象实例一般很少）

永久区

这个区域是常驻内存的，用来存放JDK自身携带的Class对象，Interface元数据，存储的是JAVA运行时的一些环境或类信息，这个区域不存在垃圾回收，关闭JVM就会释放这个区中的内存；

- JDK1.6之前 ： 永久代，常量池是在方法区

- JDK1.7：永久代，但是慢慢退化了变为`去永久代`，常量池在堆中
- JDK1.8：无永久代，常量池在元空间




```
public class HeapSpaceDemo01 {
    public static void main(String[] args) {
        //返回虚拟机可使用的最大内存
        long maxMemory = Runtime.getRuntime().maxMemory();
        //返回虚拟机初始总内存
        long totalMemory = Runtime.getRuntime().totalMemory();

        //返回虚拟机剩余内存
        long freeMemory = Runtime.getRuntime().freeMemory();

        System.out.println("MaxMemory = " + maxMemory + "B，转为MB=>" + maxMemory
/ 1024 / 1024 + "MB");//将字节转化为MB
        System.out.println("TotalMemory = " + totalMemory + "B，转为MB=>" +
totalMemory / 1024 / 1024 + "MB");//将字节转化为MB
        System.out.println("FreeMemory = " + freeMemory + "B，转为MB=>" +
freeMemory / 1024 / 1024 + "MB");//将字节转化为MB
    }

    //设置虚拟机初始总内存1024M,虚拟机可获得最大内存2048m,并打印GC的详细信息
    //-Xms1024m -Xmx2048m -XX:+PrintGCDetails

    /**
     * 当出现OOM时：
     * 1、先考虑加大虚拟机初始总内存
     * 2、如果还是报错，应该检查程序本身逻辑问题导致OOM
     */
}
```

```
"C:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...
MaxMemory = 1908932608B，转为MB=>1820MB
TotalMemory = 1029177344B，转为MB=>981MB
FreeMemory = 1013071168B，转为MB=>966MB
Heap
 PSYoungGen      total 305664K, used 20971K [0x00000000d5580000, 0x00000000eaa80000, 0x0000000100000000)
  eden space 262144K, 8% used [0x00000000d5580000,0x00000000d69fafb8,0x00000000e5580000)
  from space 43520K, 0% used [0x00000000e8000000,0x00000000e8000000,0x00000000eaa80000)
  to   space 43520K, 0% used [0x00000000e5580000,0x00000000e5580000,0x00000000e8000000)
 ParOldGen       total 699392K, used 0K [0x0000000080000000, 0x00000000aab00000, 0x00000000d5580000)
  object space 699392K, 0% used [0x0000000080000000,0x0000000080000000,0x00000000aab00000)
 Metaspace       used 3489K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 386K, capacity 388K, committed 512K, reserved 1048576K
                                这里新生代内存+老年代内存/1024等于981刚好为最大内存，而下面的元空间可以说是在逻辑上存在，但在真实JVM内存中不存在

Process finished with exit code 0
```

在一个项目中，如果出现了OOM故障，应该如何排除问题，研究为什么出错?

- 能够看到代码第几行出错：内存快照分析工具，MAT(Eclipse)，Jprofiler
- DeBug，一行一行分析代码

MAT，Jprofiler作用：

- 分析Dump内存文件，快速定位内存泄露
- 获得堆中的数据
- 获得大的对象
- ...

JProfiler工具使用

idea中plugin安装jprofiler插件并在TOOL中指定本地安装bin/exe文件

对出错代码设置jvm参数：

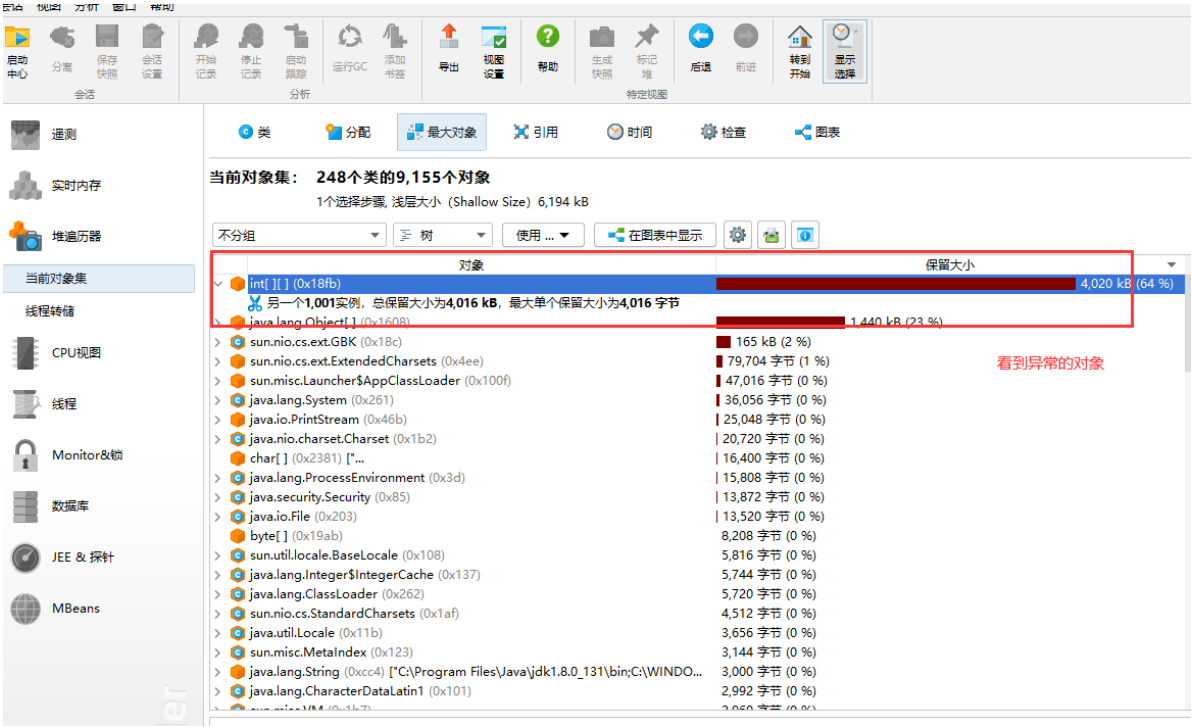
`-Xms8m -Xmx8m -XX:+HeapDumpOnOutOfMemoryError`

```
public class OOMDemo02 {
    public static void main(String[] args) {
//        int[][] array = new int[Integer.MAX_VALUE][Integer.MAX_VALUE];

        int[][] array = new int[1000][1000];
        ArrayList<int[][]> arrayList = new ArrayList<>();
        int count = 0;


        try {
            while (true) {
                arrayList.add(array);
                count++;
            }
        /*} catch (Exception e) {
            e.printStackTrace();
        }*/
        } catch (Error e) {//Error和Exception都是Throwable的子类
            System.out.println(count);
            e.printStackTrace();
        }


    }

    //-Xms8m -Xmx8m -XX:+HeapDumpOnOutOfMemoryError出现outofmemory时将信息打印进dump文件中
}
```

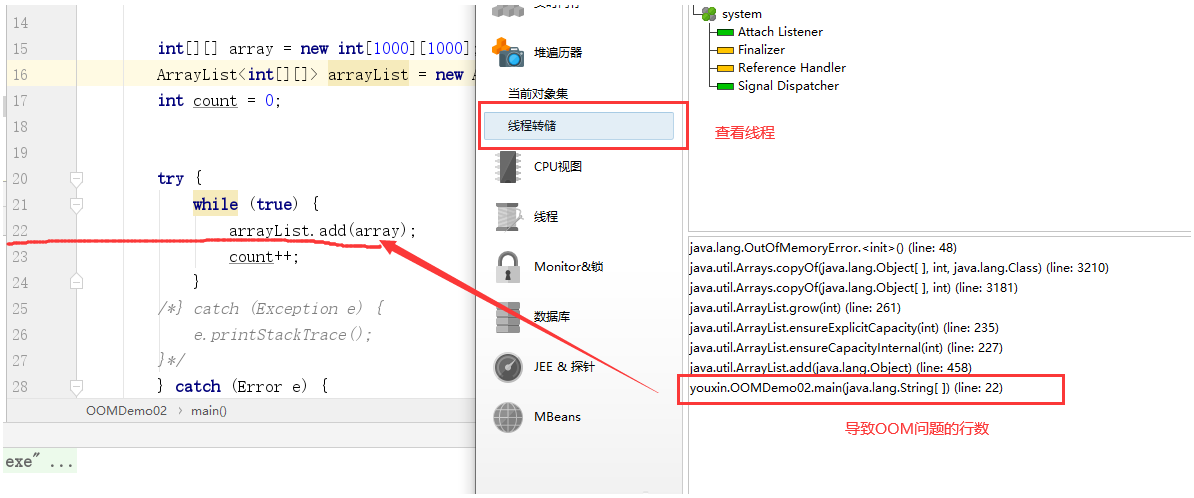
结果：

打开生成的对应文件：

查看对象：



查看线程：



GC（垃圾回收）

JVM在进行GC时，并不是对三个堆区域统一回收，大部分的时候回收的都是新生区。

GC两种类：轻GC（普通GC），重GC（全局GC）

GC算法：

引用计数法（一般不会使用）：

所谓的引用计数法就是给每个对象一个引用计数器，每当有一个地方引用它时，计数器就会加1；当引用失效时，计数器的值就会减1；任何时刻计数器的值为0的对象就是不可能再被使用的。

优点：

1、可以及时回收垃圾，在该方法中，每个对象始终知道自己是否有被引用，当被引用的计数器为0时，对象可以马上清除。

2、最大暂停时间短

3、没有必要沿着指针查找

缺点：

1、计数器的值的增减处理非常繁重

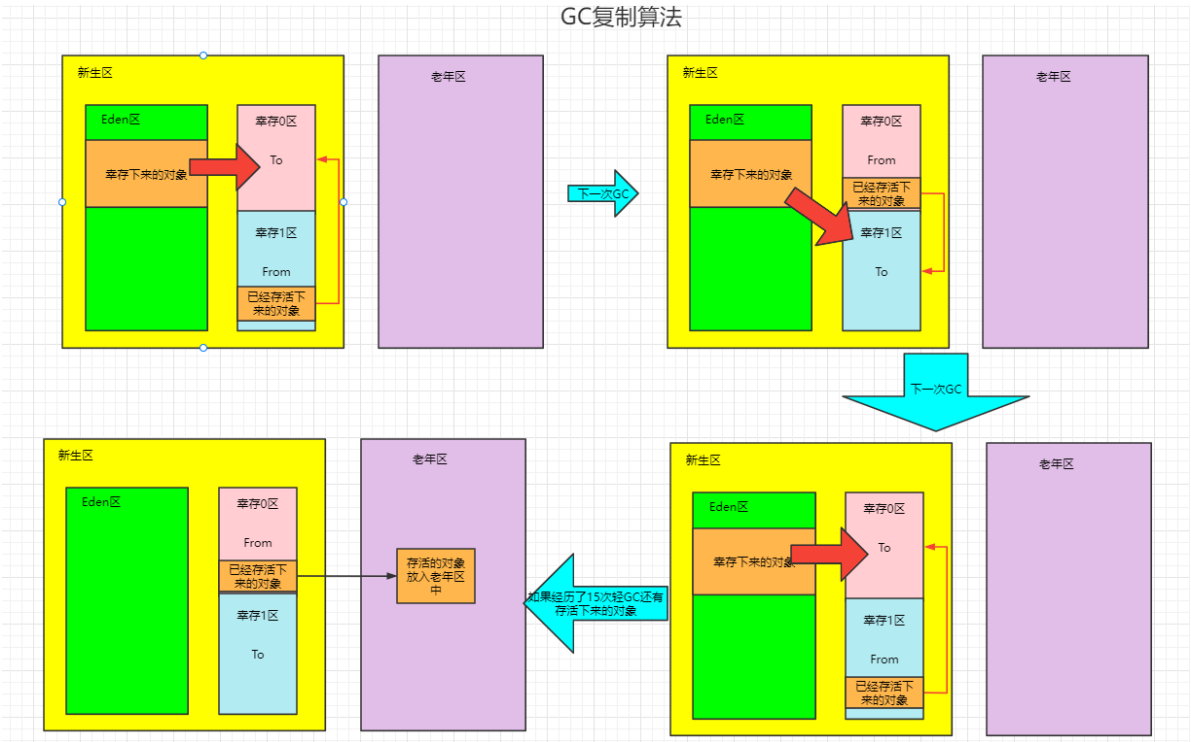
2、计数器需要占用很多位

3、实现繁琐

4、循环引用无法回收

复制算法

复制算法就是将内存空间按容量分成两块。当这一块内存用完的时候，就将还存活着的对象复制到另外一块上面，然后把已经使用过的这一块一次清理掉。这样使得每次都是对半块内存进行内存回收。内存分配时就不用考虑内存碎片等复杂情况，只要移动堆顶的指针，按顺序分配内存即可，实现简单，运行高效。



优点：

1、优秀的吞吐量

2、可实现高速分配

3、没有内存的碎片

4、与缓存兼容

缺点：

1、堆的使用效率低下，因为对于幸存区来说有一半的空间被浪费掉了（永远是to）

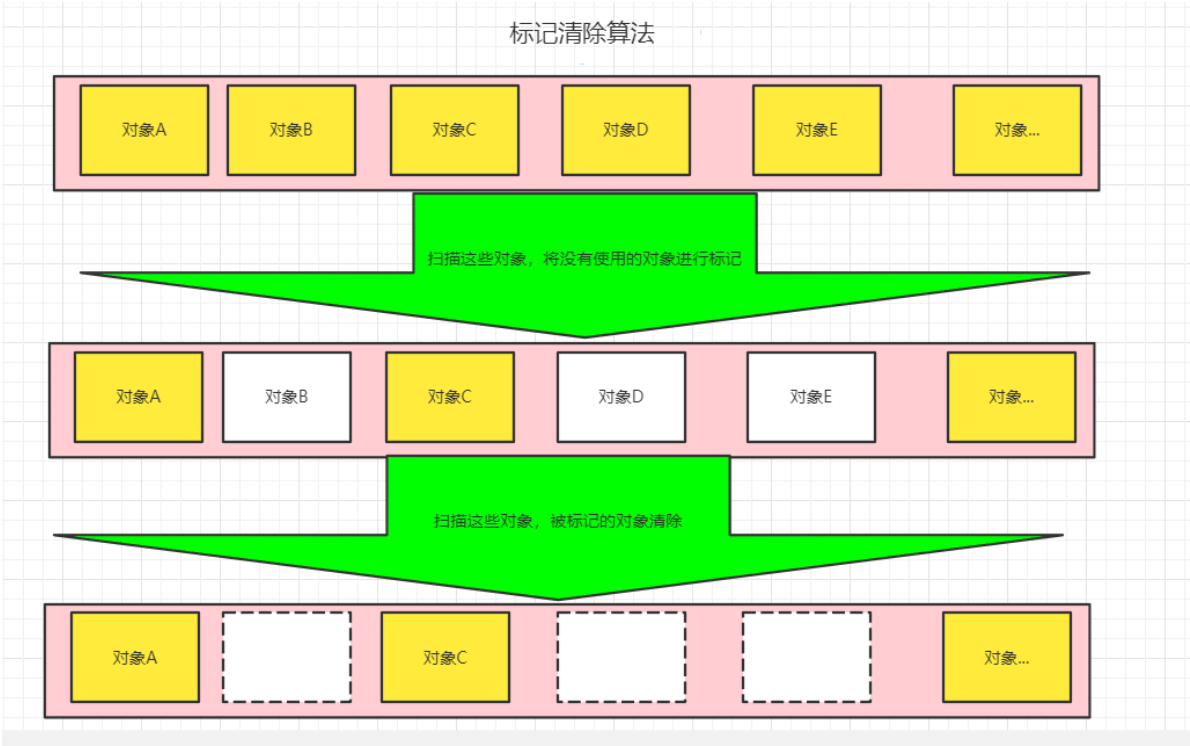
2、不兼容保守式GC算法

3、递归调用函数

复制算法的最佳使用场景：对象存活度较低的时候：新生区

标记清除算法

该算法分为标记和清除两个阶段。标记就是把所有活动对象都做上标记的阶段；清除就是将没有做上标记的对象进行回收的阶段。如下图所示。



优点：

1、实现简单

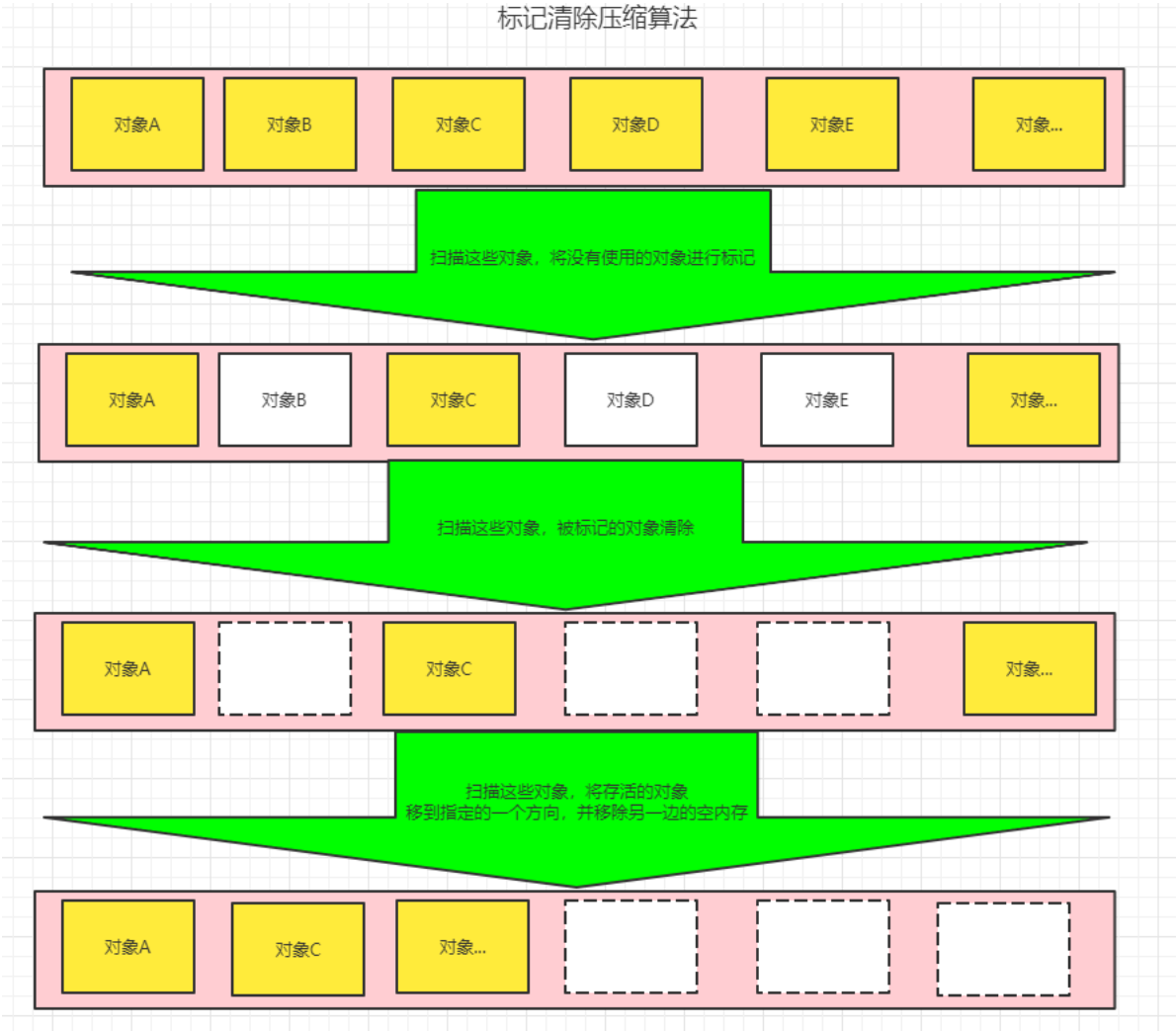
缺点：

1、内存产生碎片化

2、分配速度：因为分块不是连续的，因此每次分块都要遍历扫描，浪费时间

标记清除压缩算法

标记-压缩算法与标记-清理算法类似，只是后续步骤是让所有存活的对象移动到一端，然后直接清除掉端边界以外的内存。



优缺点：

该算法可以有效的利用堆，但是压缩需要花比较多的时间成本。

总结

内存效率：复制算法>标记清除算法>标记压缩算法（时间复杂度）

内存最优：复制算法=标记压缩算法>标记压缩算法

内存利用率：标记压缩算法=标记清除算法>复制算法

GC算法没有最优的算法，只有最合适的算法：分代收集算法

新生代：

- 存活率低
- 复制算法

老年代：

- 区域大，存活率高
- 标记清除算法（当内存碎片不是太多时）+标记压缩算法（内存碎片过多时）