JVM相关面试题

1 JVM组成

1.1 JVM由那些部分组成，运行流程是什么？

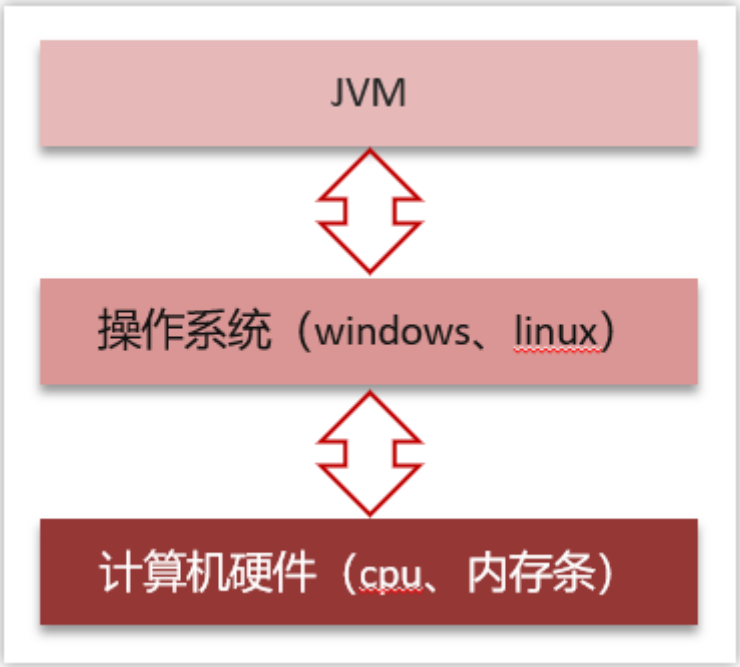
> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

JVM是什么

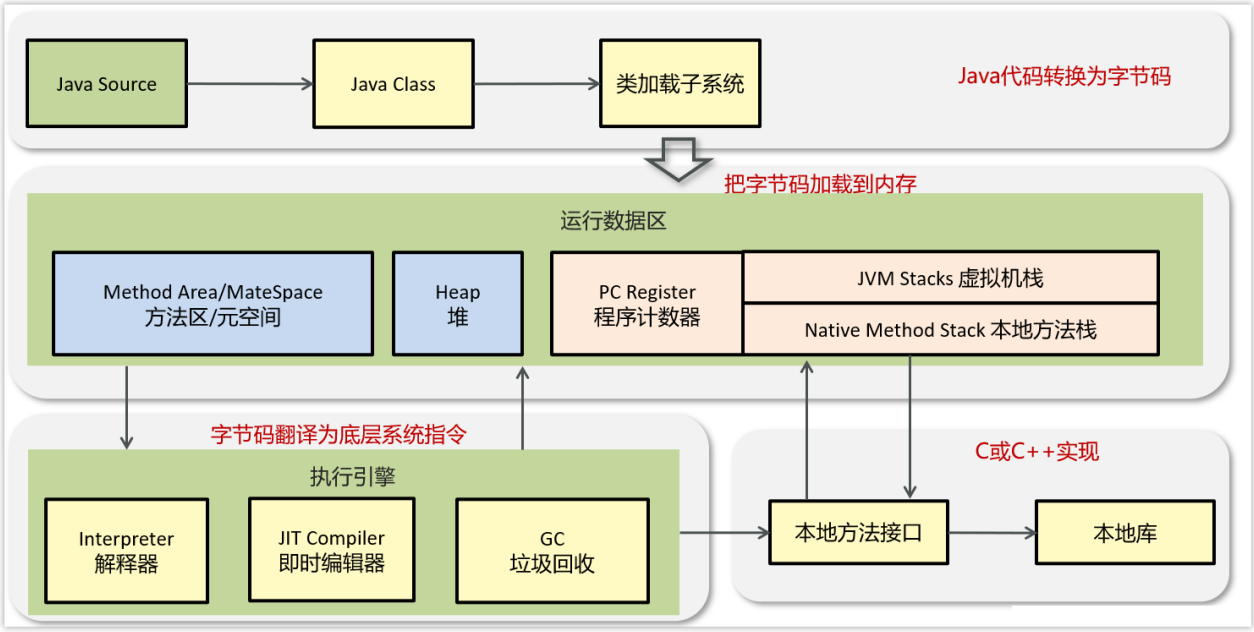
Java Virtual Machine Java程序的运行环境（java二进制字节码的运行环境）

好处：

- 一次编写，到处运行
- 自动内存管理，垃圾回收机制



JVM由哪些部分组成，运行流程是什么？



从图中可以看出 JVM 的主要组成部分

- ClassLoader（类加载器）
- Runtime Data Area（运行时数据区，内存分区）
- Execution Engine（执行引擎）
- Native Method Library（本地库接口）

运行流程：

（1）类加载器（ClassLoader）把Java代码转换为字节码

（2）运行时数据区（Runtime Data Area）把字节码加载到内存中，而字节码文件只是JVM的一套指令集规范，并不能直接交给底层系统去执行，而是有执行引擎运行

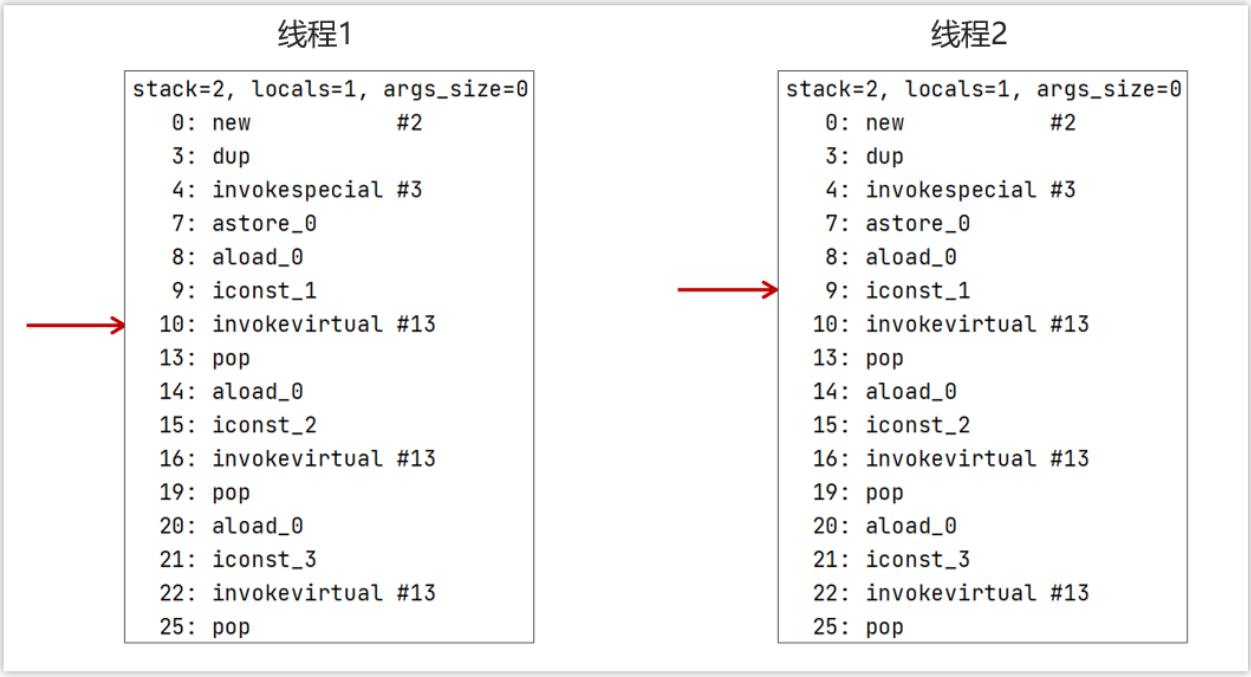
（3）执行引擎（Execution Engine）将字节码翻译为底层系统指令，再交由CPU执行去执行，此时需要调用其他语言的本地库接口（Native Method Library）来实现整个程序的功能。

1.2 什么是程序计数器？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

程序计数器：线程私有的，内部保存的字节码的行号。用于记录正在执行的字节码指令的地址。

> javap -verbose xx.class 打印堆栈大小，局部变量的数量和方法的参数。



java虚拟机对于多线程是通过线程轮流切换并且分配线程执行时间。在任何的一个时间点上，一个处理器只会处理执行一个线程，如果当前被执行的这个线程它所分配的执行时间用完了【挂起】。处理器会切换到另外的一个线程上来进行执行。并且这个线程的执行时间用完了，接着处理器就会又来执行被挂起的这个线程。

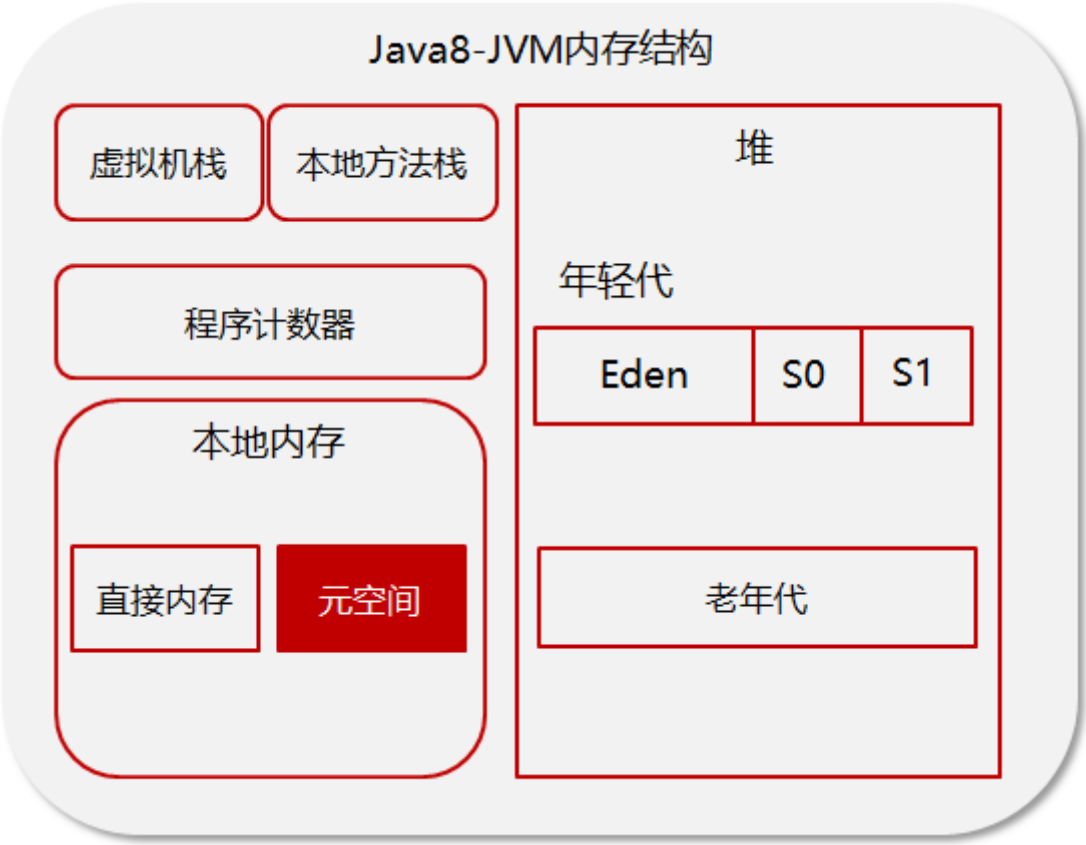
那么现在有一个问题就是，当前处理器如何能够知道，对于这个被挂起的线程，它上一次执行到了哪里？那么这时就需要从程序计数器中来回去到当前的这个线程他上一次执行的行号，然后接着继续向下执行。

程序计数器是JVM规范中唯一一个没有规定出现OOM的区域，所以这个空间也不会进行GC。

1.3 你能给我详细的介绍Java堆吗?

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

线程共享的区域：主要用来保存对象实例，数组等，当堆中没有内存空间可分配给实例，也无法再扩展时，则抛出OutOfMemoryError异常。



- 年轻代被划分为三部分，Eden区和两个大小严格相同的Survivor区，根据JVM的策略，在经过几次垃圾收集后，任然存活于Survivor的对象将被移动到老年代区间。
- 老年代主要保存生命周期长的对象，一般是一些老的对象
- 元空间保存的类信息、静态变量、常量、编译后的代码

为了避免方法区出现OOM，所以在java8中将堆上的方法区【永久代】给移动到了本地内存上，重新开辟了一块空间，叫做**元空间**。那么现在就可以避免掉OOM的出现了。

元空间(MetaSpace)介绍

在 HotSpot JVM 中，永久代（≈方法区）中用于存放类和方法的元数据以及常量池，比如Class 和 Method。每当一个类初次被加载的时候，它的元数据都会放到永久代中。

永久代是有大小限制的，因此如果加载的类太多，很有可能导致永久代内存溢出，即OutOfMemoryError，为此不得不对虚拟机做调优。

那么，Java 8 中 PermGen 为什么被移出 HotSpot JVM 了？

官网给出了解释：http://openjdk.java.net/jeps/122

```
This is part of the JRockit and Hotspot convergence effort. JRockit customers do not need to configure the permanent generation (since JRockit does not have a permanent generation) and are accustomed to not configuring the permanent generation.

移除永久代是为融合HotSpot JVM与 JRockit VM而做出的努力，因为JRockit没有永久代，不需要配置永久代。
```

1）由于 PermGen 内存经常会溢出，引发OutOfMemoryError，因此 JVM 的开发者希望这一块内存可以更灵活地被管理，不要再经常出现这样的 OOM。

2）移除 PermGen 可以促进 HotSpot JVM 与 JRockit VM 的融合，因为 JRockit 没有永久代。

准确来说，Perm 区中的字符串常量池被移到了堆内存中是在 Java7 之后，Java 8 时，PermGen 被元空间代替，其他内容比如**类元信息、字段、静态属性、方法、常量**等都移动到元空间区。比如 java/lang/Object 类元信息、静态属性 System.out、整型常量等。

元空间的本质和永久代类似，都是对 JVM 规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制。

1.4 什么是虚拟机栈

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

Java Virtual machine Stacks (java 虚拟机栈)

- 每个线程运行时所需要的内存，称为虚拟机栈，先进后出
- 每个栈由多个栈帧（frame）组成，对应着每次方法调用时所占用的内存
- 每个线程只能有一个活动栈帧，对应着当前正在执行的那个方法

1. 垃圾回收是否涉及栈内存？

   垃圾回收主要指就是堆内存，当栈帧弹栈以后，内存就会释放

2. 栈内存分配越大越好吗？

   未必，默认的栈内存通常为1024k

   栈帧过大会导致线程数变少，例如，机器总内存为512m，目前能活动的线程数则为512个，如果把栈内存改为2048k，那么能活动的栈帧就会减半

3. 方法内的局部变量是否线程安全？

   - 如果方法内局部变量没有逃离方法的作用范围，它是线程安全的
   - 如果是局部变量引用了对象，并逃离方法的作用范围，需要考虑线程安全
   - 比如以下代码：

```
public static void main(String[] args) {
    StringBuilder sb = new StringBuilder();
    sb.append(1);
    sb.append(2);
    new Thread(()->{
        m2(sb);
    }).start();
}
public static void m1(){
    StringBuilder sb = new StringBuilder();
    sb.append(1);
    sb.append(2);
    System.out.println(sb.toString());
}
public static void m2(StringBuilder sb){
    sb.append(3);
    sb.append(4);
    System.out.println(sb.toString());
}
public static StringBuilder m3(){
    StringBuilder sb = new StringBuilder();
    sb.append(5);
    sb.append(6);
    return sb;
}
```

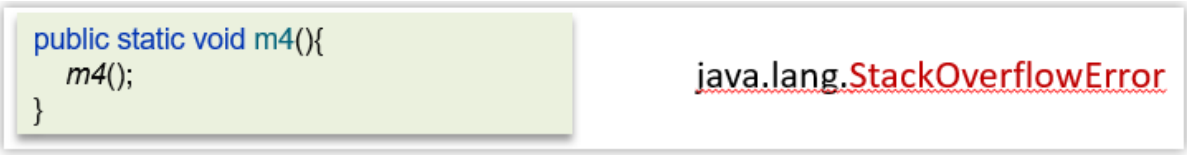
线程安全

线程不安全

线程不安全

栈内存溢出情况

- 栈帧过多导致栈内存溢出，典型问题：递归调用



- 栈帧过大导致栈内存溢出

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

组成部分：堆、方法区、栈、本地方法栈、程序计数器

1、堆解决的是对象实例存储的问题，垃圾回收器管理的主要区域。
2、方法区可以认为是堆的一部分，用于存储已被虚拟机加载的信息，常量、静态变量、即时编译器编译后的代码。
3、栈解决的是程序运行的问题，栈里面存的是栈帧，栈帧里面存的是局部变量表、操作数栈、动态链接、方法出口等信息。
4、本地方法栈与栈功能相同，本地方法栈执行的是本地方法，一个Java调用非Java代码的接口。
5、程序计数器（PC寄存器）程序计数器中存放的是当前线程所执行的字节码的行数。JVM工作时就是通过改变这个计数器的值来选取下一个需要执行的字节码指令。

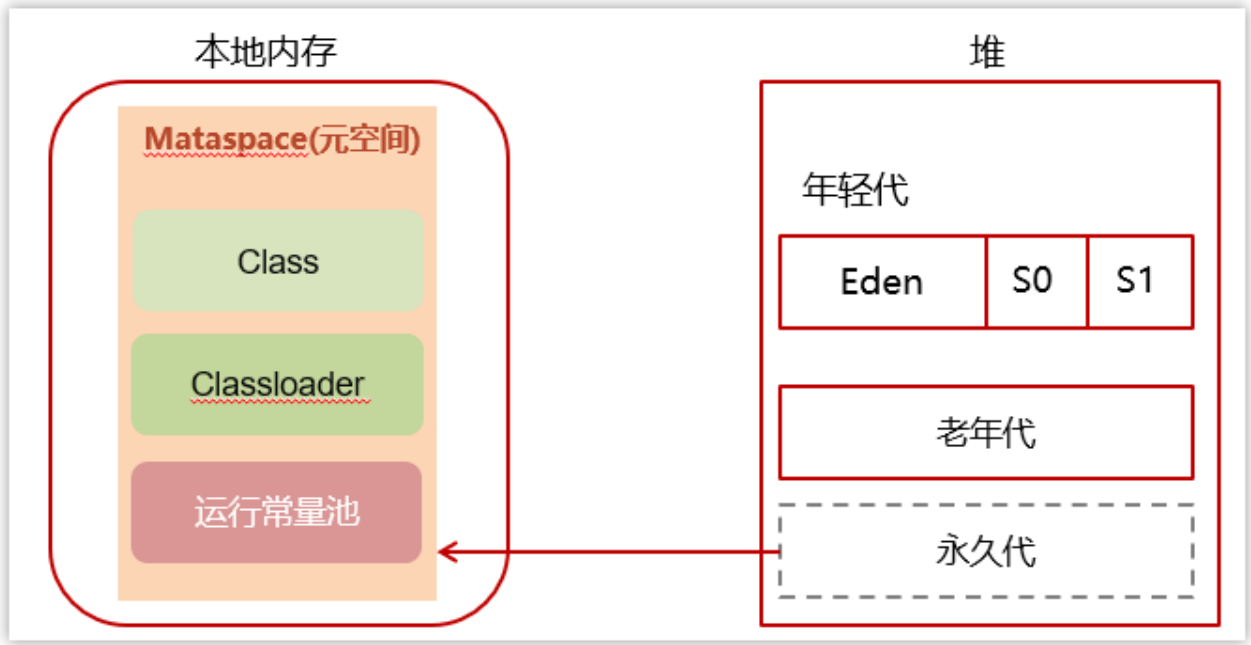
1.5 能不能解释一下方法区？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

1.5.1 概述

- 方法区(Method Area)是各个线程共享的内存区域
- 主要存储类的信息、运行时常量池
- 虚拟机启动的时候创建，关闭虚拟机时释放

- 如果方法区域中的内存无法满足分配请求，则会抛出OutOfMemoryError: Metaspace



1.5.2 常量池

可以看作是一张表，虚拟机指令根据这张常量表找到要执行的类名、方法名、参数类型、字面量等信息

查看字节码结构（类的基本信息、常量池、方法定义） `javap -v xx.class`

比如下面是一个Application类的main方法执行，源码如下：

```
public class Application {
    public static void main(String[] args) {
        System.out.println("hello world");
    }
}
```

找到类对应的class文件存放目录，执行命令： `javap -v Application.class` 查看字节码结构

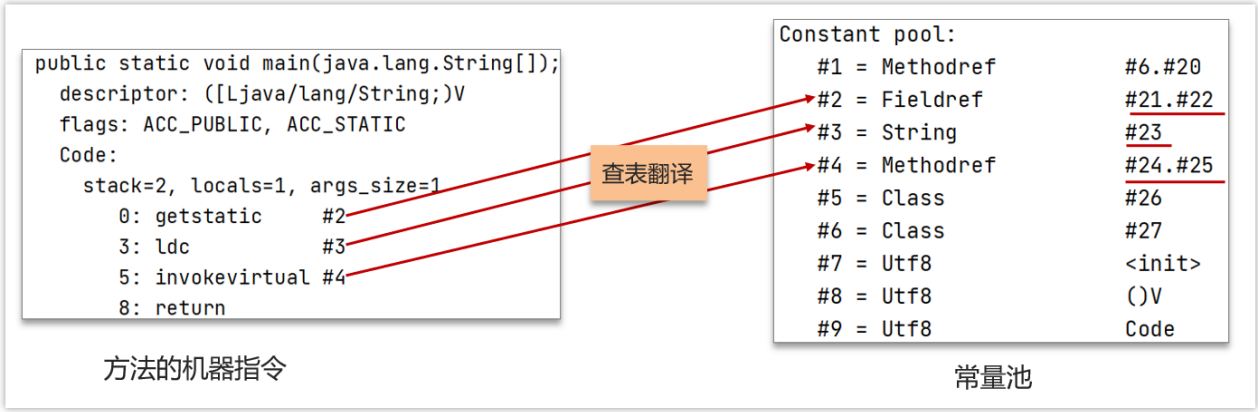
```
D:\code\jvm-demo\target\classes\com\heima\jvm>javap -v
Application.class
Classfile /D:/code/jvm-
demo/target/classes/com/heima/jvm/Application.class
  Last modified 2023-05-07; size 564 bytes     //最后修改的时间
  MD5 checksum c1b64ed6491b9a16c2baab5061c64f88    //签名
```

```
  Compiled from "Application.java"   //从哪个源码编译
public class com.heima.jvm.Application   //包名，类名
  minor version: 0
  major version: 52     //jdk版本
  flags: ACC_PUBLIC, ACC_SUPER  //修饰符
Constant pool:   //常量池
   #1 = Methodref          #6.#20         // java/lang/Object."
<init>":()V
   #2 = Fieldref           #21.#22        //
java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #23            // hello world
   #4 = Methodref          #24.#25        //
java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = Class              #26            //
com/heima/jvm/Application
   #6 = Class              #27            // java/lang/Object
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               LocalVariableTable
  #12 = Utf8               this
  #13 = Utf8               Lcom/heima/jvm/Application;
  #14 = Utf8               main
  #15 = Utf8               ([Ljava/lang/String;)V
  #16 = Utf8               args
  #17 = Utf8               [Ljava/lang/String;
  #18 = Utf8               SourceFile
  #19 = Utf8               Application.java
  #20 = NameAndType        #7:#8          // "<init>":()V
  #21 = Class              #28            // java/lang/System
  #22 = NameAndType        #29:#30        //
out:Ljava/io/PrintStream;
  #23 = Utf8               hello world
  #24 = Class              #31            // java/io/PrintStream
  #25 = NameAndType        #32:#33        // println:
(Ljava/lang/String;)V
  #26 = Utf8               com/heima/jvm/Application
  #27 = Utf8               java/lang/Object
  #28 = Utf8               java/lang/System
  #29 = Utf8               out
```

```
  #30 = Utf8               Ljava/io/PrintStream;
  #31 = Utf8               java/io/PrintStream
  #32 = Utf8               println
  #33 = Utf8               (Ljava/lang/String;)V
{
  public com.heima.jvm.Application();  //构造方法
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method
java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 3: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this   Lcom/heima/jvm/Application;

  public static void main(java.lang.String[]);  //main方法
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2                  // Field
java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #3                  // String hello world
         5: invokevirtual #4                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
         8: return
      LineNumberTable:
        line 7: 0
        line 8: 8
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       9     0  args   [Ljava/lang/String;
}
SourceFile: "Application.java"
```

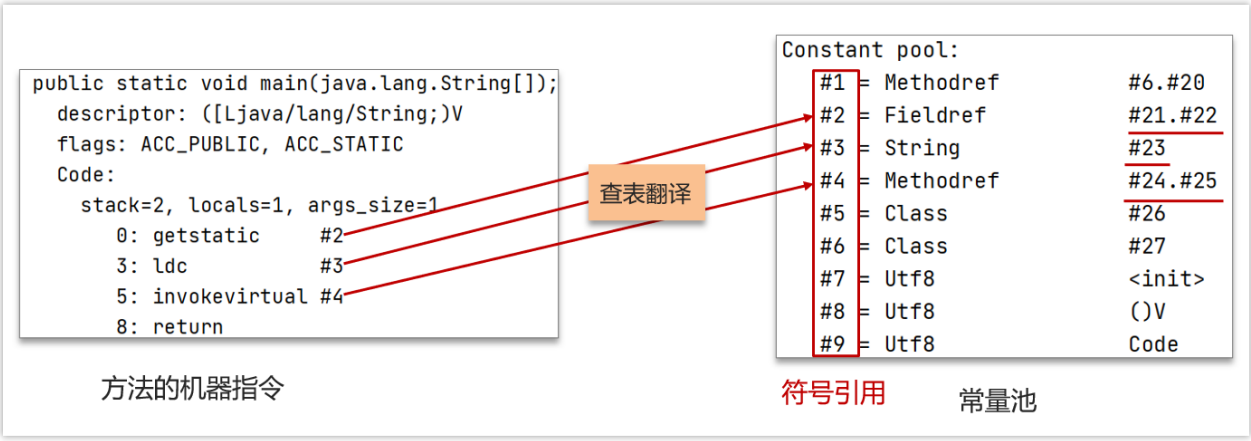
下图，左侧是main方法的指令信息，右侧constant pool 是常量池

main方法按照指令执行的时候，需要到常量池中查表翻译找到具体的类和方法地址去执行



1.5.3 运行时常量池

常量池是 *.class 文件中的，当该类被加载，它的常量池信息就会放入运行时常量池，并把里面的符号地址变为真实地址



1.6 你听过直接内存吗？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

不受 JVM 内存回收管理，是虚拟机的系统内存，常见于 NIO 操作时，用于数据缓冲区，分配回收成本较高，但读写性能高，不受 JVM 内存回收管理

举例：

需求，在本地电脑中的一个较大的文件（超过100m）从一个磁盘挪到另外一个磁盘

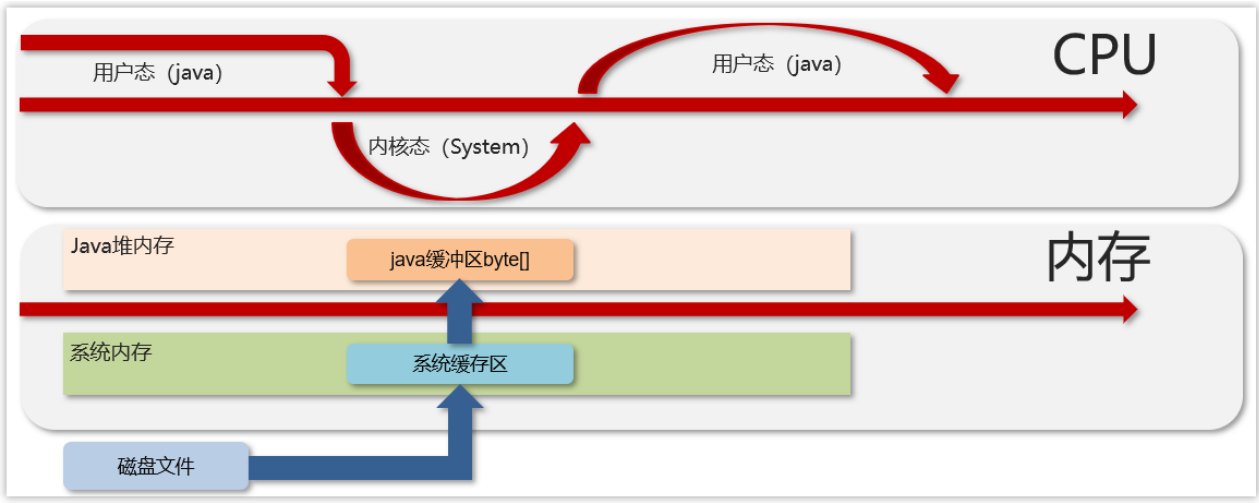
代码如下：

```
/**
 * 演示 ByteBuffer 作用
 */
public class Demo1_9 {
    static final String FROM = "E:\\编程资料\\第三方教学视频
\\youtube\\Getting Started with Spring Boot-sbPSjI4tt10.mp4";
    static final String TO = "E:\\a.mp4";
    static final int _1Mb = 1024 * 1024;

    public static void main(String[] args) {
        io(); // io 用时: 1535.586957 1766.963399 1359.240226
        directBuffer(); // directBuffer 用时: 479.295165 702.291454
562.56592
    }

    private static void directBuffer() {
        long start = System.nanoTime();
        try (FileChannel from = new
FileInputStream(FROM).getChannel();
             FileChannel to = new FileOutputStream(TO).getChannel();
        ) {
            ByteBuffer bb = ByteBuffer.allocateDirect(_1Mb);
            while (true) {
                int len = from.read(bb);
                if (len == -1) {
```

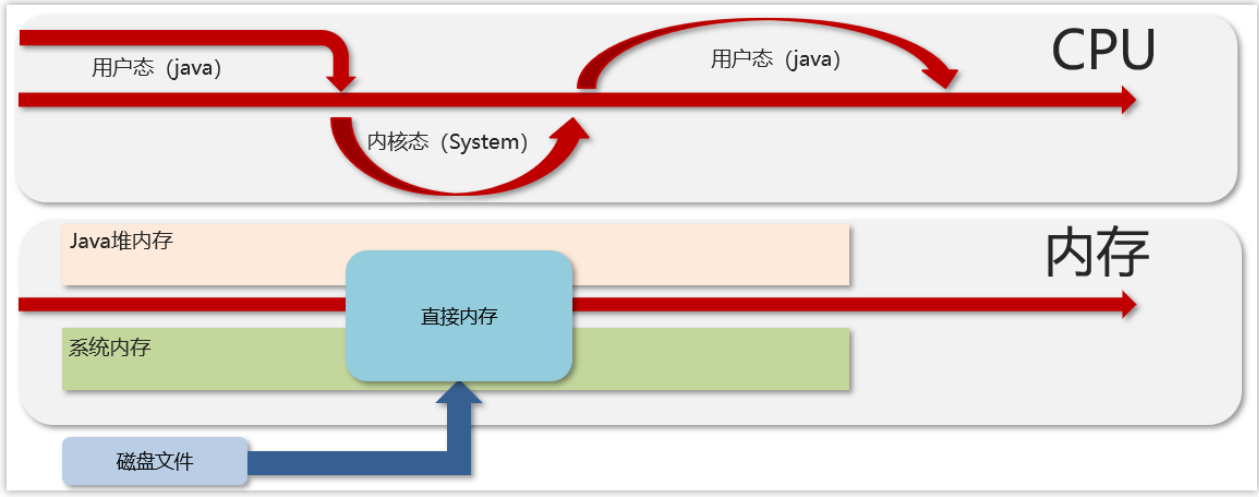
```
                    break;
                }
                bb.flip();
                to.write(bb);
                bb.clear();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        long end = System.nanoTime();
        System.out.println("directBuffer 用时: " + (end - start) /
1000_000.0);
    }

    private static void io() {
        long start = System.nanoTime();
        try (FileInputStream from = new FileInputStream(FROM);
             FileOutputStream to = new FileOutputStream(TO);
        ) {
            byte[] buf = new byte[_1Mb];
            while (true) {
                int len = from.read(buf);
                if (len == -1) {
                    break;
                }
                to.write(buf, 0, len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        long end = System.nanoTime();
        System.out.println("io 用时: " + (end - start) /
1000_000.0);
    }
}
```

可以发现，使用传统的IO的时间要比NIO操作的时间长了很多了，也就说NIO的读性能更好。

这个是跟我们的JVM的直接内存是有一定关系，如下图，是传统阻塞IO的数据传输流程



下图是NIO传输数据的流程，在这个里面主要使用到了一个直接内存，不需要在堆中开辟空间进行数据的拷贝，jvm可以直接操作直接内存，从而使数据读写传输更快。



1.7 堆栈的区别是什么？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

1、栈内存一般会用来存储局部变量和方法调用，但堆内存是用来存储Java对象和数组的的。堆会GC垃圾回收，而栈不会。

2、栈内存是线程私有的，而堆内存是线程共有的。

3,、两者异常错误不同，但如果栈内存或者堆内存不足都会抛出异常。

栈空间不足：java.lang.StackOverFlowError。

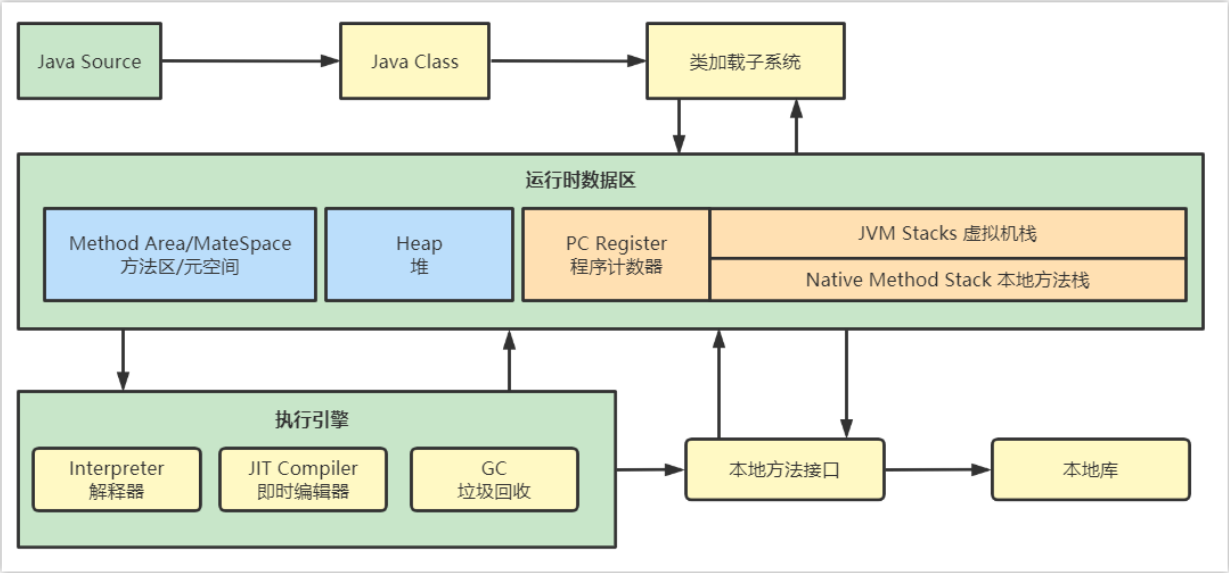
堆空间不足：java.lang.OutOfMemoryError。

2 类加载器

2.1 什么是类加载器，类加载器有哪些?

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

要想理解类加载器的话，务必要先清楚对于一个Java文件，它从编译到执行的整个过程。



- 类加载器：用于装载字节码文件(.class文件)
- 运行时数据区：用于分配存储空间
- 执行引擎：执行字节码文件或本地方法
- 垃圾回收器：用于对JVM中的垃圾内容进行回收

类加载器

JVM只会运行二进制文件，而类加载器（ClassLoader）的主要作用就是将**字节码文件加载到JVM中**，从而让Java程序能够启动起来。现有的类加载器基本上都是java.lang.ClassLoader的子类，该类的只要职责就是用于将指定的类找到或生成对应的字节码文件，同时类加载器还会负责加载程序所需要的资源

类加载器种类

类加载器根据各自加载范围的不同，划分为四种类加载器：

- **启动类加载器(BootStrap ClassLoader)：**

  该类并不继承ClassLoader类，其是由C++编写实现。用于加载**JAVA_HOME/jre/lib**目录下的类库。

- **扩展类加载器(ExtClassLoader)：**

  该类是ClassLoader的子类，主要加载**JAVA_HOME/jre/lib/ext**目录中的类库。

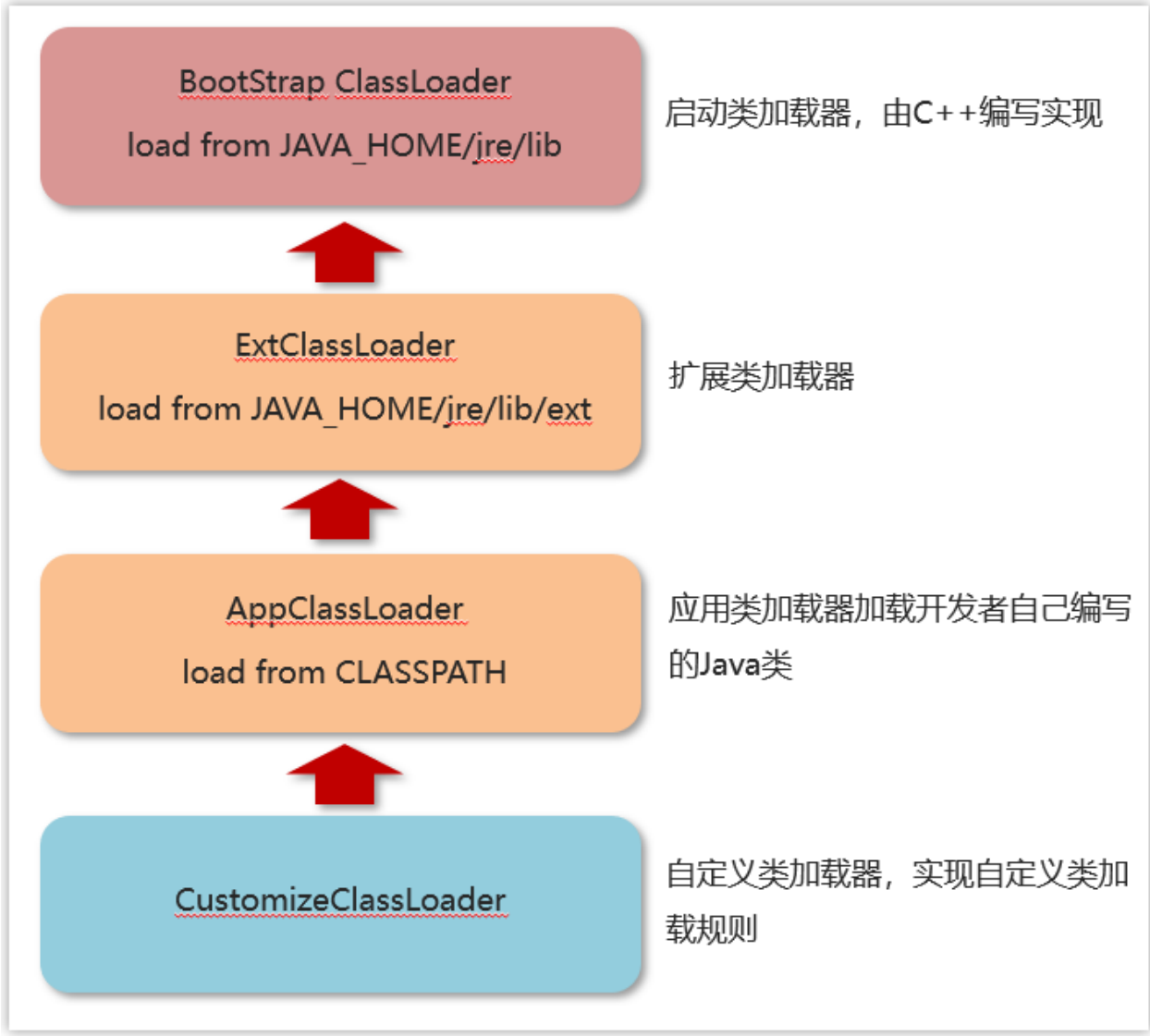
- **应用类加载器(AppClassLoader)：**

  该类是ClassLoader的子类，主要用于加载**classPath**下的类，也就是加载开发者自己编写的Java类。

- **自定义类加载器：**

  开发者自定义类继承ClassLoader，实现自定义类加载规则。

上述三种类加载器的层次结构如下如下：

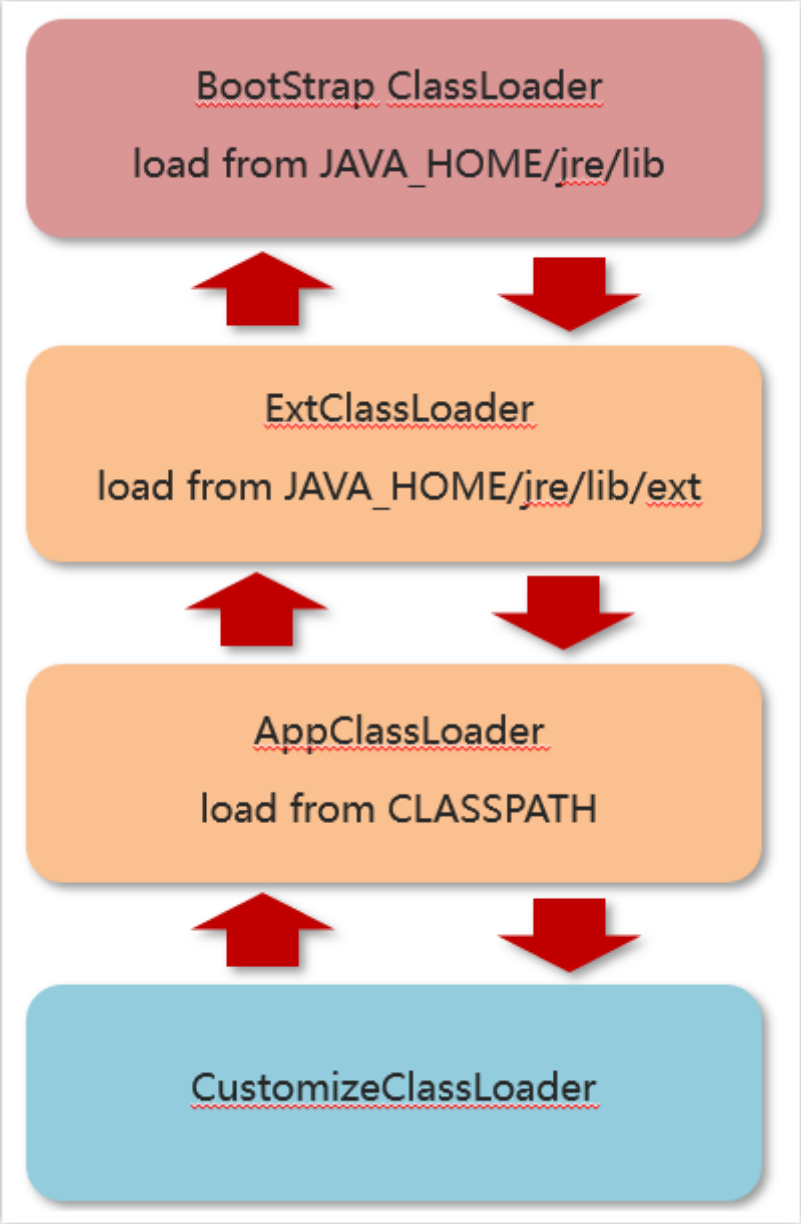


类加载器的体系并不是“继承”体系，而是**委派体系**，类加载器首先会到自己的parent中查找类或者资源，如果找不到才会到自己本地查找。类加载器的委托行为动机是为了避免相同的类被加载多次。

2.2 什么是双亲委派模型？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆☆

如果一个类加载器在接到加载类的请求时，它首先不会自己尝试去加载这个类，而是把这个请求任务委托给父类加载器去完成，依次递归，如果父类加载器可以完成类加载任务，就返回成功；只有父类加载器无法完成此加载任务时，才由下一级去加载。



2.3 JVM为什么采用双亲委派机制

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

（1）通过双亲委派机制可以避免某一个类被重复加载，当父类已经加载后则无需重复加载，保证唯一性。

（2）为了安全，保证类库API不会被修改

在工程中新建java.lang包，接着在该包下新建String类，并定义main函数

```
public class String {

    public static void main(String[] args) {

        System.out.println("demo info");
    }
}
```

此时执行main函数，会出现异常，在类 java.lang.String 中找不到 main 方法

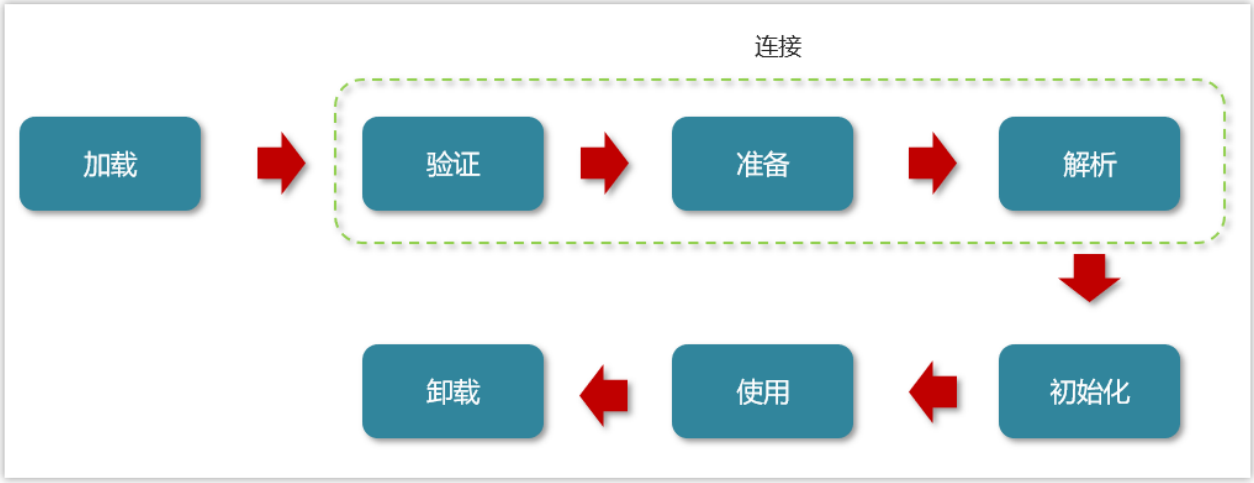
```
错误: 在类 java.lang.String 中找不到 main 方法, 请将 main 方法定义为:
   public static void main(String[] args)
否则 JavaFX 应用程序类必须扩展javafx.application.Application
```

出现该信息是因为由双亲委派的机制，java.lang.String的在启动类加载器(Bootstrap classLoader)得到加载，因为在核心jre库中有其相同名字的类文件，但该类中并没有main方法。这样就能防止恶意篡改核心API库。

2.4 说一下类装载的执行过程？

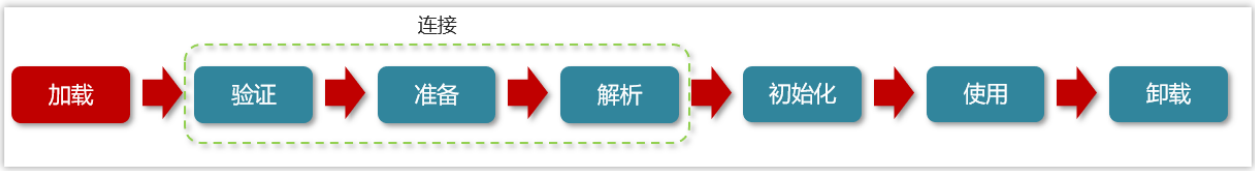
> 难易程度：☆☆☆☆☆
>
> 出现频率：☆☆☆

类从加载到虚拟机中开始，直到卸载为止，它的整个生命周期包括了：加载、验证、准备、解析、初始化、使用和卸载这7个阶段。其中，验证、准备和解析这三个部分统称为连接（linking）。

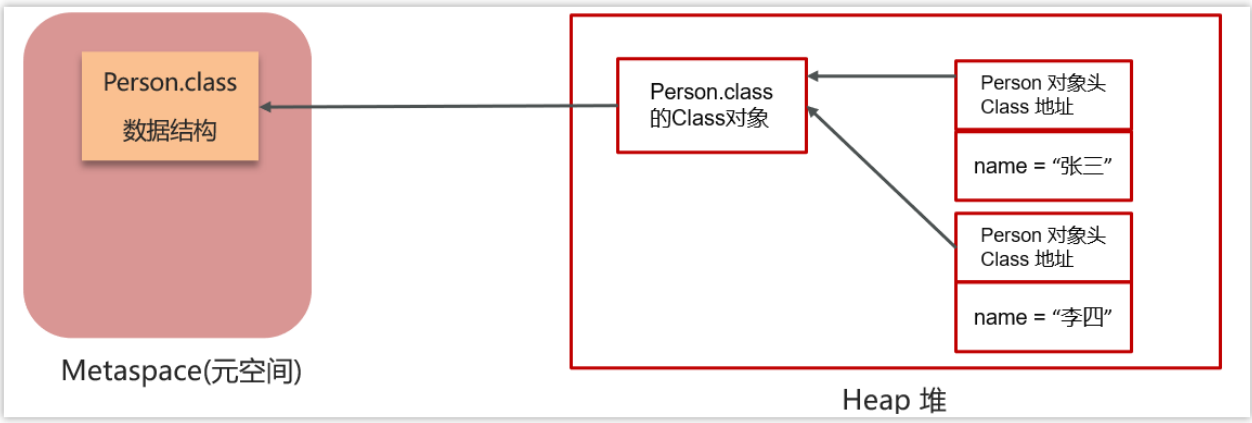


类加载过程详解

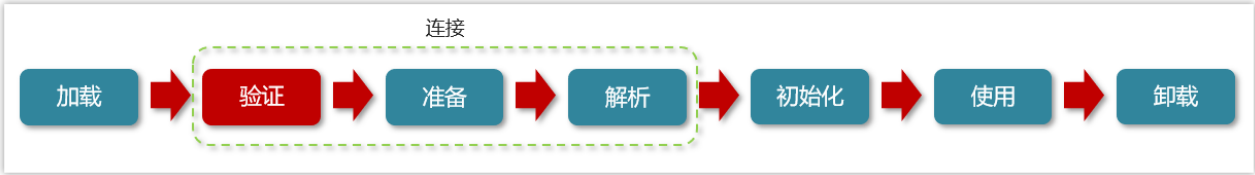
1.加载



- 通过类的全名，获取类的二进制数据流。
- 解析类的二进制数据流为方法区内的数据结构（Java类模型）
- 创建java.lang.Class类的实例，表示该类型。作为方法区这个类的各种数据的访问入口



2.验证



验证类是否符合JVM规范，安全性检查

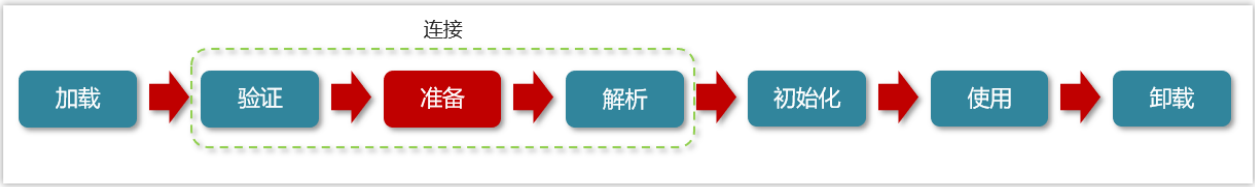
(1)文件格式验证:是否符合Class文件的规范
(2)元数据验证
　这个类是否有父类（除了Object这个类之外，其余的类都应该有父类）
　这个类是否继承（extends）了被final修饰过的类（被final修饰过的类表示类不能被继承）
　类中的字段、方法是否与父类产生矛盾。（被final修饰过的方法或字段是不能覆盖的）
(3)字节码验证

主要的目的是通过对数据流和控制流的分析，确定程序语义是合法的、符合逻辑的。

(4)符号引用验证：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量

> 比如：int i = 3;
> 字面量：3
> 符号引用：i

3.准备

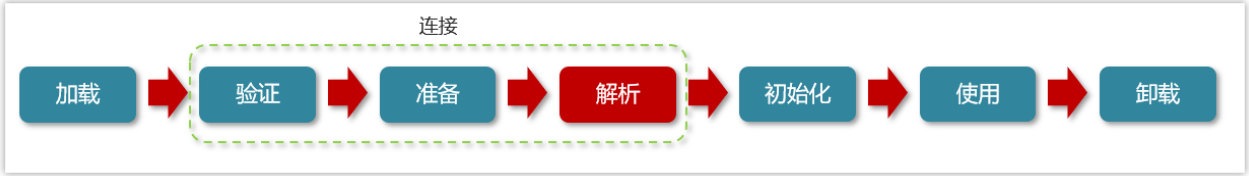


为类变量分配内存并设置类变量初始值

- static变量，分配空间在准备阶段完成（设置默认值），赋值在初始化阶段完成
- static变量是final的基本类型，以及字符串常量，值已确定，赋值在准备阶段完成
- static变量是final的引用类型，那么赋值也会在初始化阶段完成

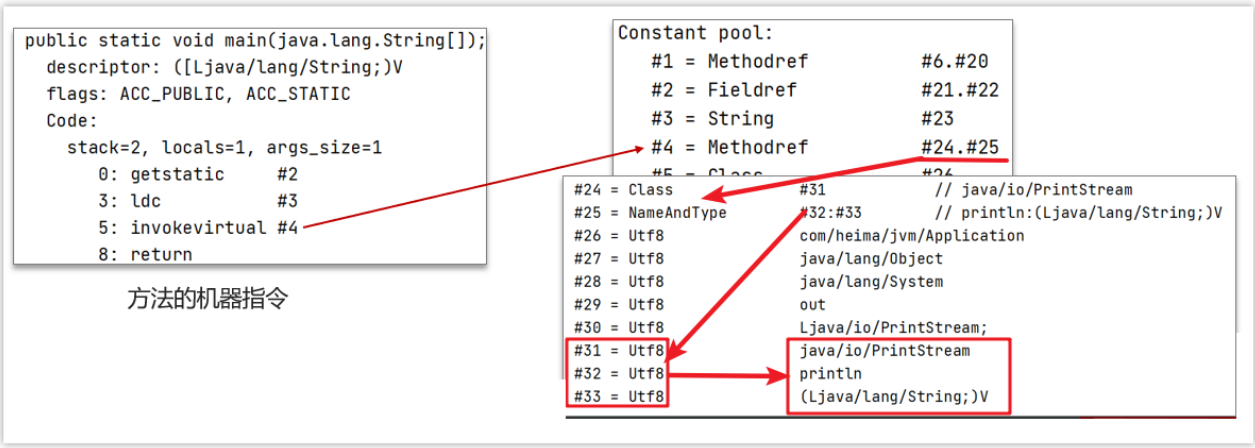
```
public class Application {

    static int b = 10;
    static final int c = 20;
    static final String d = "hello";
    static final Object obj = new Object();

}
```

4.解析

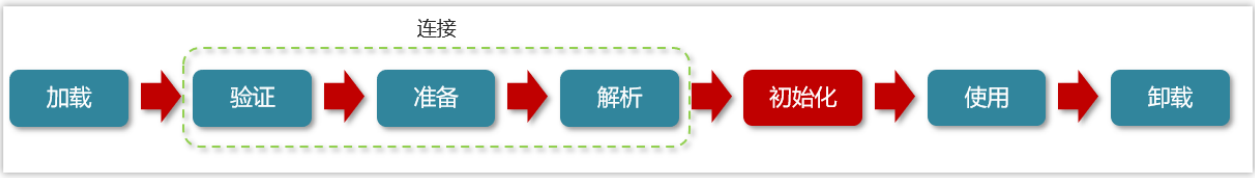


把类中的符号引用转换为直接引用

比如：方法中调用了其他方法，方法名可以理解为符号引用，而直接引用就是使用指针直接指向方法。



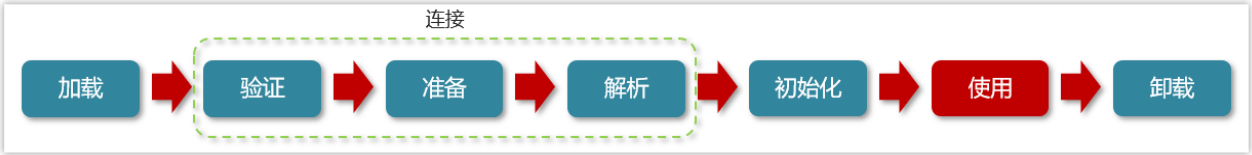
5.初始化



对类的静态变量，静态代码块执行初始化操作

- 如果初始化一个类的时候，其父类尚未初始化，则优先初始化其父类。
- 如果同时包含多个静态变量和静态代码块，则按照自上而下的顺序依次执行。

6.使用



JVM 开始从入口方法开始执行用户的程序代码

- 调用静态类成员信息（比如：静态字段、静态方法）
- 使用new关键字为其创建对象实例

7.卸载

当用户程序代码执行完毕后，JVM 便开始销毁创建的 Class 对象，最后负责运行的 JVM 也退出内存

3 垃圾收回

3.1 简述Java垃圾回收机制？（GC是什么？为什么要GC）

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

为了让程序员更专注于代码的实现，而不用过多的考虑内存释放的问题，所以，在Java语言中，有了自动的垃圾回收机制，也就是我们熟悉的GC(Garbage Collection)。

有了垃圾回收机制后，程序员只需要关心内存的申请即可，内存的释放由系统自动识别完成。

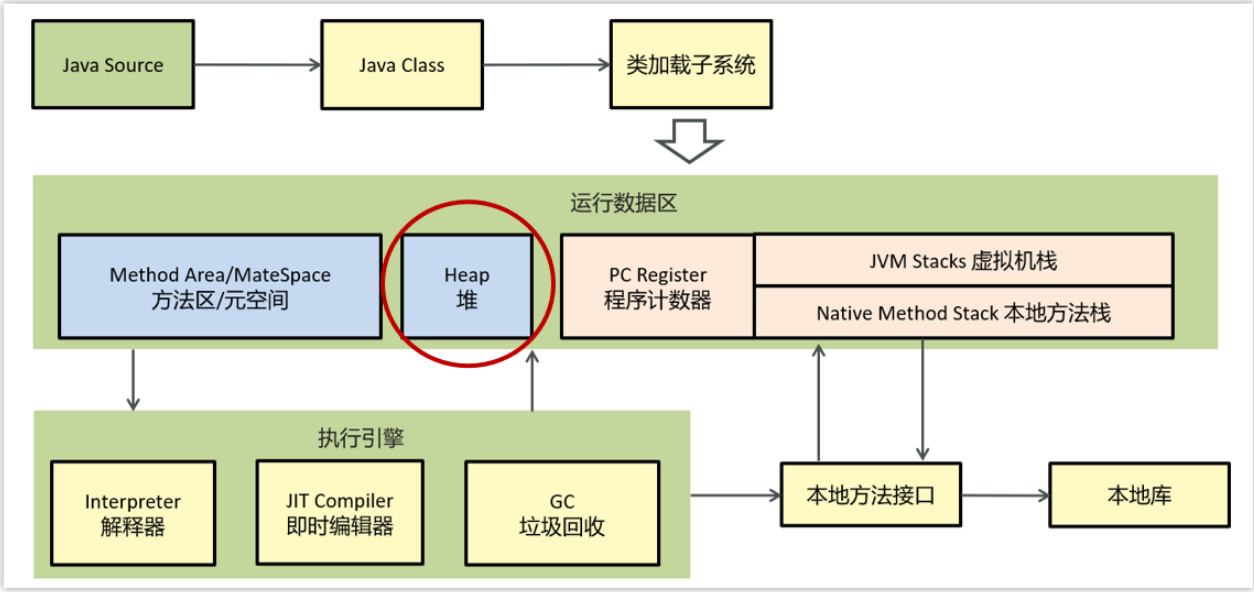
在进行垃圾回收时，不同的对象引用类型，GC会采用不同的回收时机

换句话说，自动的垃圾回收的算法就会变得非常重要了，如果因为算法的不合理，导致内存资源一直没有释放，同样也可能会导致内存溢出的。

当然，除了Java语言，C#、Python等语言也都有自动的垃圾回收机制。

3.2 对象什么时候可以被垃圾器回收

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆☆



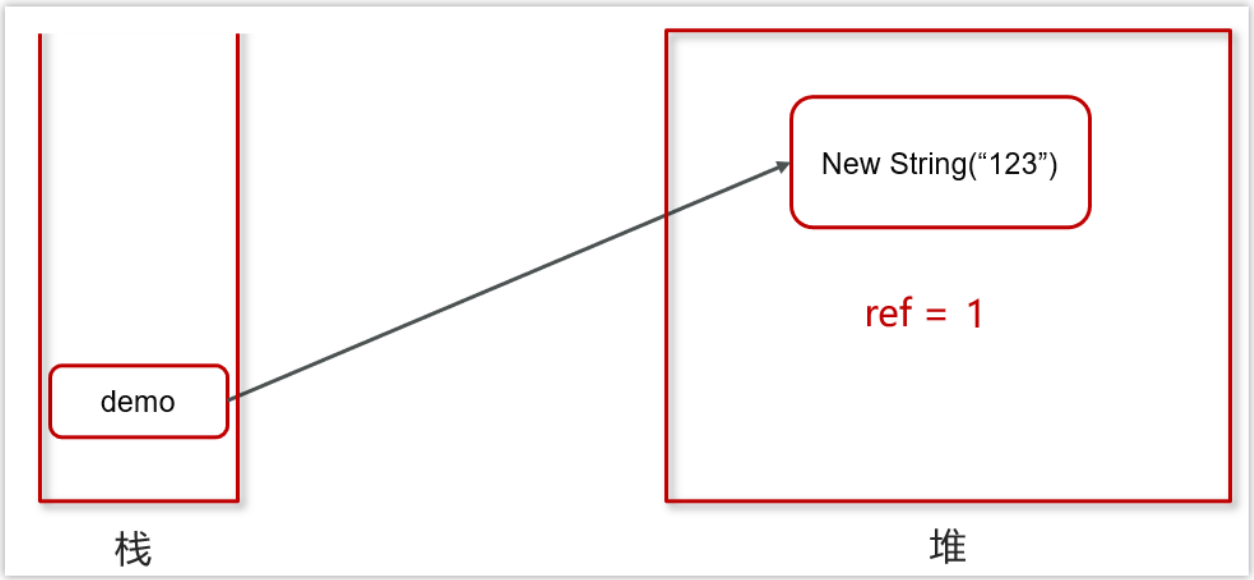
简单一句就是：如果一个或多个对象没有任何的引用指向它了，那么这个对象现在就是垃圾，如果定位了垃圾，则有可能会被垃圾回收器回收。

如果要定位什么是垃圾，有两种方式来确定，第一个是引用计数法，第二个是可达性分析算法

3.2.1 引用计数法

一个对象被引用了一次，在当前的对象头上递增一次引用次数，如果这个对象的引用次数为0，代表这个对象可回收

```
String demo = new String("123");
```



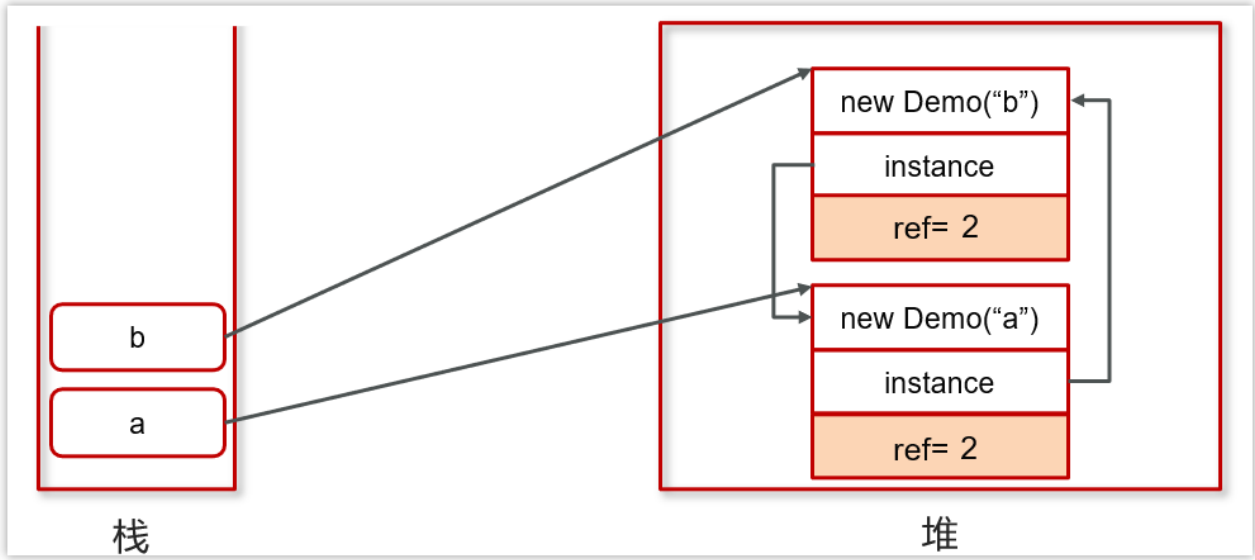

```
String demo = null;
```

当对象间出现了循环引用的话，则引用计数法就会失效

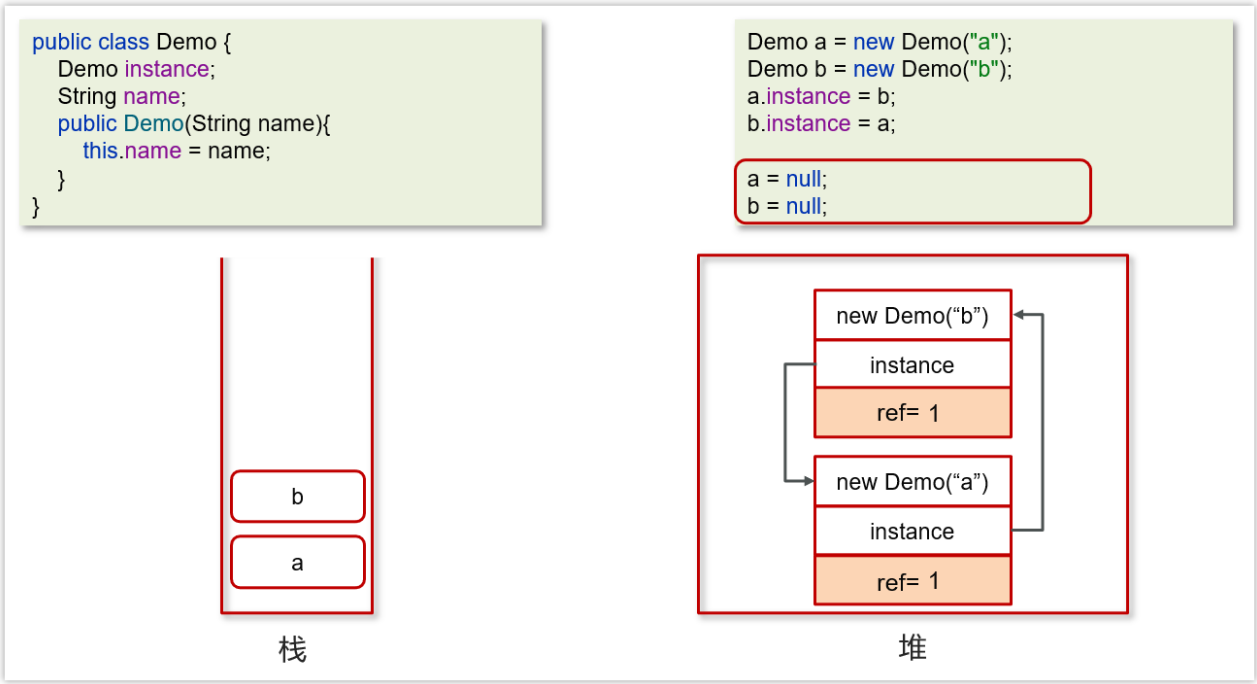
```
public class Demo {
    Demo instance;
    String name;
    public Demo(String name){
        this.name = name;
    }
}
```

```
Demo a = new Demo("a");
Demo b = new Demo("b");
a.instance = b;
b.instance = a;

a = null;
b = null;
```

先执行右侧代码的前4行代码



目前上方的引用关系和计数都是没问题的，但是，如果代码继续往下执行，如下图



虽然a和b都为null，但是由于a和b存在循环引用，这样a和b永远都不会被回收。

优点：

- 实时性较高，无需等到内存不够的时候，才开始回收，运行时根据对象的计数器是否为0，就可以直接回收。
- 在垃圾回收过程中，应用无需挂起。如果申请内存时，内存不足，则立刻报OOM错误。
- 区域性，更新对象的计数器时，只是影响到该对象，不会扫描全部对象。

缺点：

- 每次对象被引用时，都需要去更新计数器，有一点时间开销。
- **浪费CPU资源**，即使内存够用，仍然在运行时进行计数器的统计。
- **无法解决循环引用问题，会引发内存泄露**。（最大的缺点）

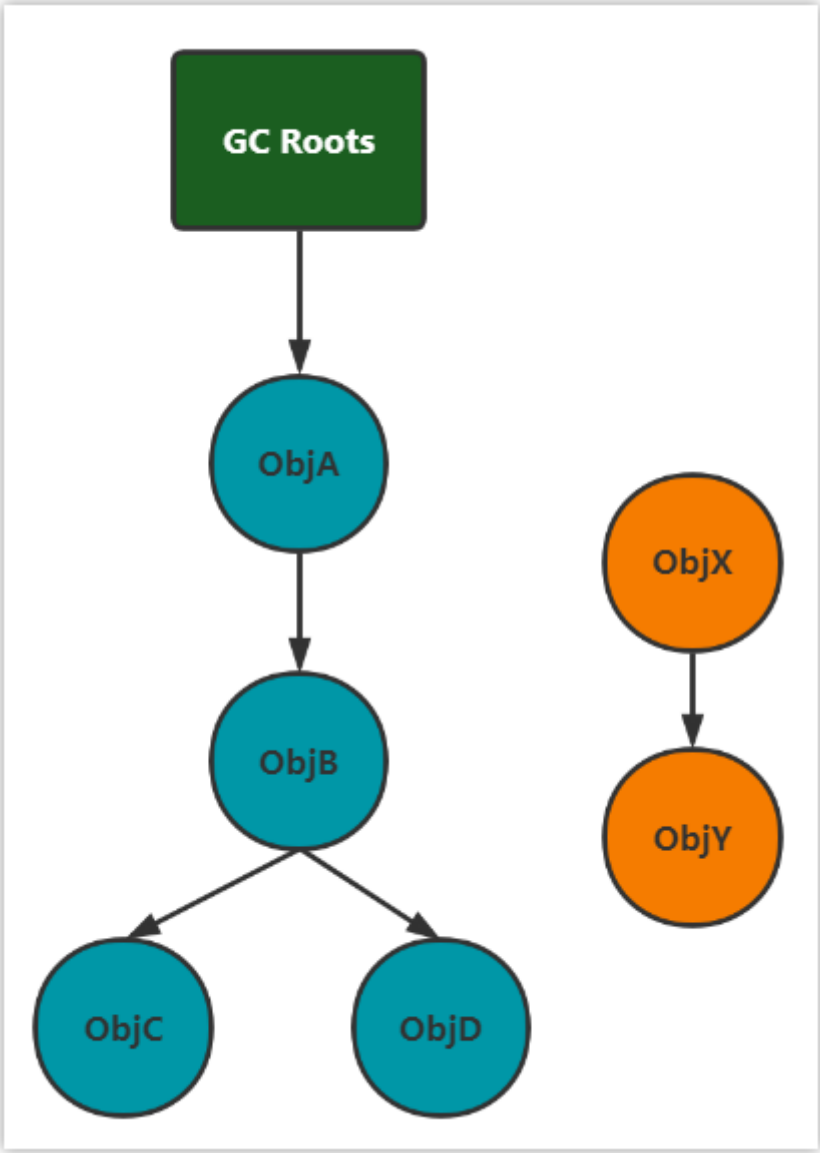
3.2.2 可达性分析算法

现在的虚拟机采用的都是通过可达性分析算法来确定哪些内容是垃圾。

会存在一个根节点【GC Roots】，引出它下面指向的下一个节点，再以下一个节点节点开始找出它下面的节点，依次往下类推。直到所有的节点全部遍历完毕。

> 根对象是那些肯定不能当做垃圾回收的对象，就可以当做根对象

> 局部变量，静态方法，静态变量，类信息
>
> 核心是：判断某对象是否与根对象有直接或间接的引用，如果没有被引用，则可以当做垃圾回收



X,Y这两个节点是可回收的，但是**并不会马上的被回收！！** 对象中存在一个方法【finalize】。当对象被标记为可回收后，当发生GC时，首先**会判断这个对象是否执行了finalize方法**，如果这个方法还没有被执行的话，那么就会先来执行这个方法，接着在这个方法执行中，可以设置当前这个对象与GC ROOTS产生关联，那么这个方法执行完成之后，GC会再次判断对象是否可达，如果仍然不可达，则会进行回收，如果可达了，则不会进行回收。

finalize方法对于每一个对象来说，只会执行一次。如果第一次执行这个方法的时候，设置了当前对象与RC ROOTS关联，那么这一次不会进行回收。 那么等到这个对象第二次被标记为可回收时，那么该对象的finalize方法就不会再次执行了。

GC ROOTS：

- 虚拟机栈（栈帧中的本地变量表）中引用的对象

```
/**
 * demo是栈帧中的本地变量，当 demo = null 时，由于此时 demo 充当了 GC Root 的作用，demo与原来指向的实例 new Demo() 断开了连接，对象被回收。
 */
public class Demo {
    public static  void main(String[] args) {
        Demo demo = new Demo();
        demo = null;
    }
}
```

- 方法区中类静态属性引用的对象

```
/**
 * 当栈帧中的本地变量 b = null 时，由于 b 原来指向的对象与 GC Root (变量 b) 断开了连接，所以 b 原来指向的对象会被回收，而由于我们给 a 赋值了变量的引用，a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 Demo {
    public static Demo a;
    public static  void main(String[] args) {
        Demo b = new Demo();
        b.a = new Demo();
        b = null;
    }
}
```

- 方法区中常量引用的对象

```java
/**
 * 常量 a 指向的对象并不会因为 demo 指向的对象被回收而回收
 */
public class Demo {

    public static final Demo a = new Demo();

    public static  void main(String[] args) {
        Demo demo = new Demo();
        demo = null;
    }
}
```

- 本地方法栈中 JNI（即一般说的 Native 方法）引用的对象

3.3 JVM 垃圾回收算法有哪些？

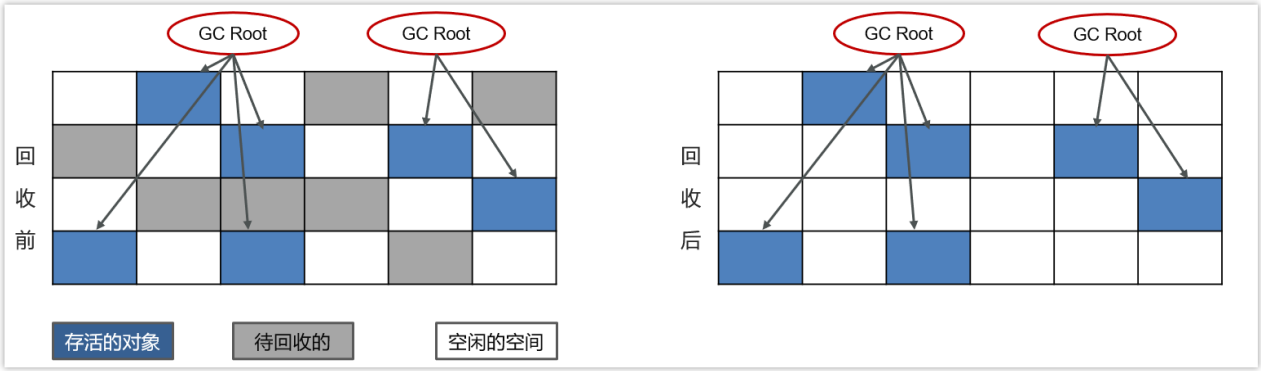
> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

3.3.1 标记清除算法

标记清除算法，是将垃圾回收分为2个阶段，分别是**标记和清除**。

1.根据可达性分析算法得出的垃圾进行标记

2.对这些标记为可回收的内容进行垃圾回收



可以看到，标记清除算法解决了引用计数算法中的循环引用的问题，没有从root节点引用的对象都会被回收。

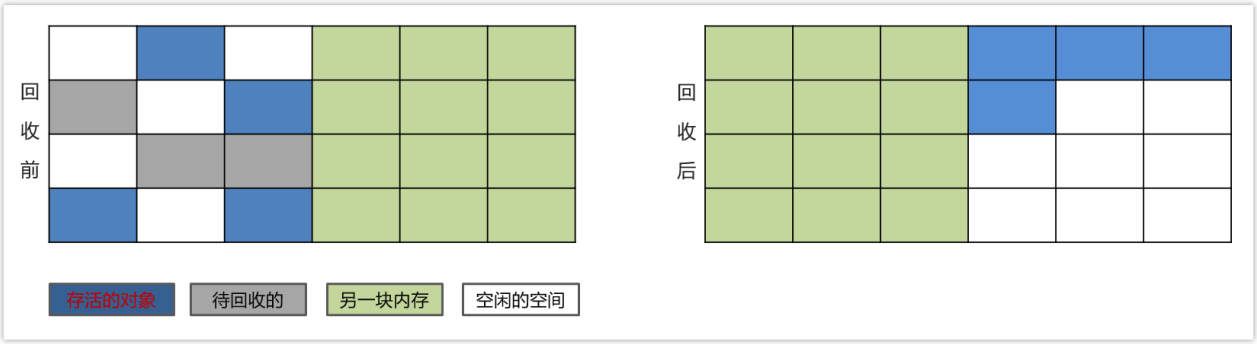
同样，标记清除算法也是有缺点的：

- 效率较低，**标记和清除两个动作都需要遍历所有的对象**，并且在GC时，**需要停止应用程序**，对于交互性要求比较高的应用而言这个体验是非常差的。
- （**重要**）通过标记清除算法清理出来的内存，碎片化较为严重，因为被回收的对象可能存在于内存的各个角落，所以清理出来的内存是不连贯的。

3.3.2 复制算法

复制算法的核心就是，**将原有的内存空间一分为二，每次只用其中的一块**，在垃圾回收时，将正在使用的对象复制到另一个内存空间中，然后将该内存空间清空，交换两个内存的角色，完成垃圾的回收。

如果内存中的垃圾对象较多，需要复制的对象就较少，这种情况下适合使用该方式并且效率比较高，反之，则不适合。



1）将内存区域分成两部分，每次操作其中一个。

2）当进行垃圾回收时，将正在使用的内存区域中的存活对象移动到未使用的内存区域。当移动完对这部分内存区域一次性清除。

3）周而复始。

优点：

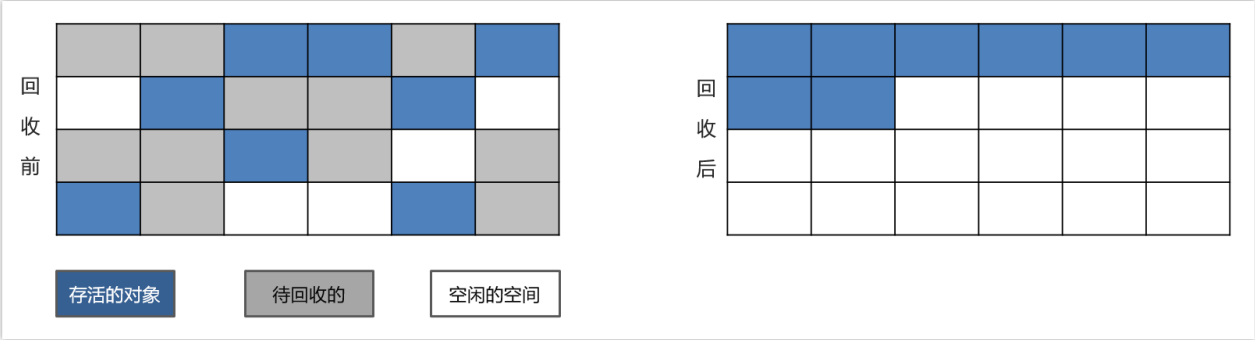
- 在垃圾对象多的情况下，效率较高
- 清理后，内存无碎片

缺点：

- 分配的2块内存空间，在同一个时刻，只能使用一半，内存使用率较低

3.3.3 标记整理算法

标记压缩算法是在标记清除算法的基础之上，做了优化改进的算法。和标记清除算法一样，也是从根节点开始，对对象的引用进行标记，在清理阶段，并不是简单的直接清理可回收对象，而是将存活对象都向内存另一端移动，然后清理边界以外的垃圾，从而解决了碎片化的问题。



1）标记垃圾。

2）需要清除向右边走，不需要清除的向左边走。

3）清除边界以外的垃圾。

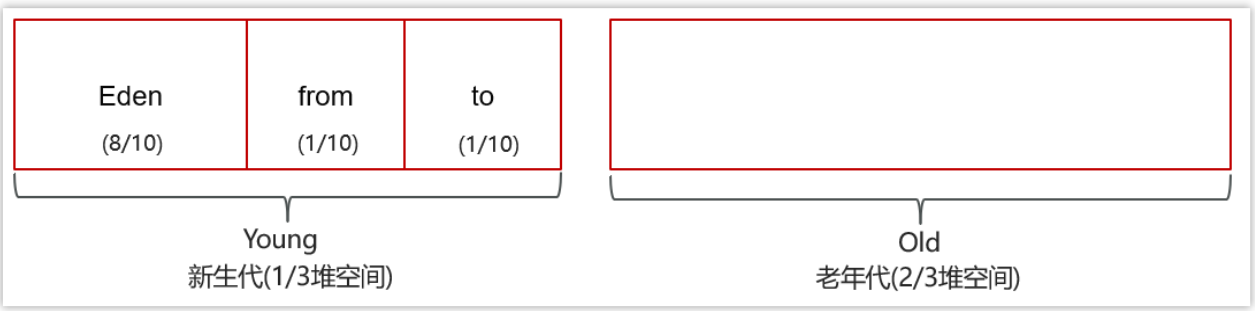
优缺点同标记清除算法，解决了标记清除算法的碎片化的问题，同时，标记压缩算法多了一步，对象移动内存位置的步骤，其效率也有有一定的影响。

与复制算法对比：复制算法标记完就复制，但标记整理算法得等把所有存活对象都标记完毕，再进行整理

3.4 分代收集算法

3.4.1 概述

在java8时，堆被分为了两份：**新生代和老年代【1：2】**，在java7时，还存在一个永久代。



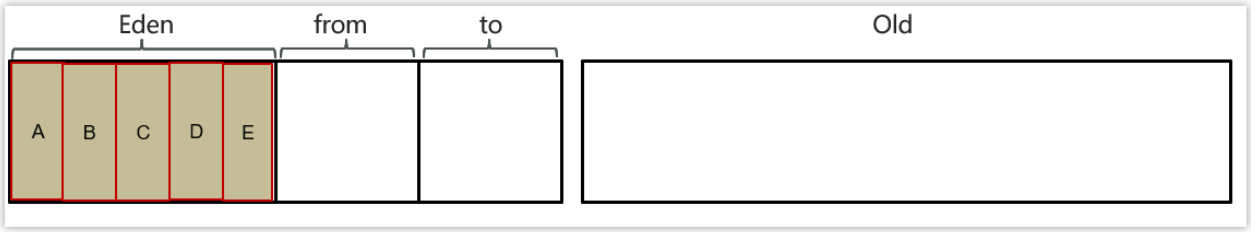
对于新生代，内部又被分为了三个区域。Eden区，S0区，S1区【8：1：1】

当对新生代产生GC：MinorGC【young GC】

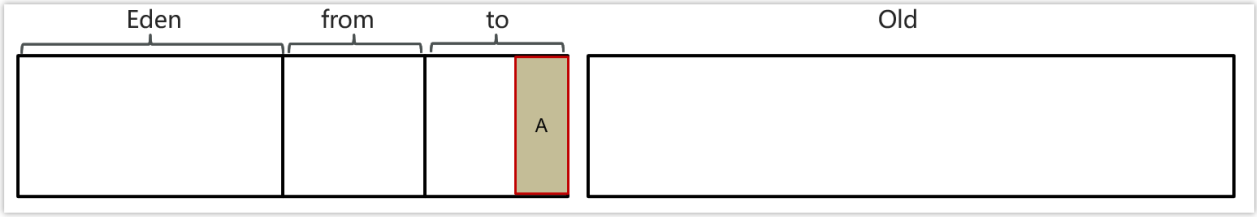
当对老年代代产生GC：Major GC

当对新生代和老年代产生FullGC： 新生代 + 老年代完整垃圾回收，暂停时间长，**应尽力避免**

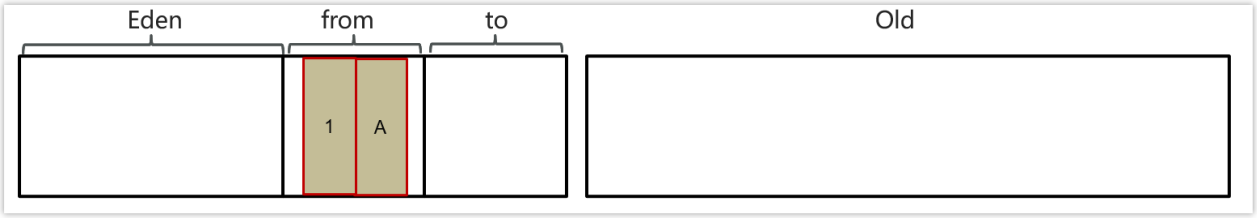
3.4.2工作机制



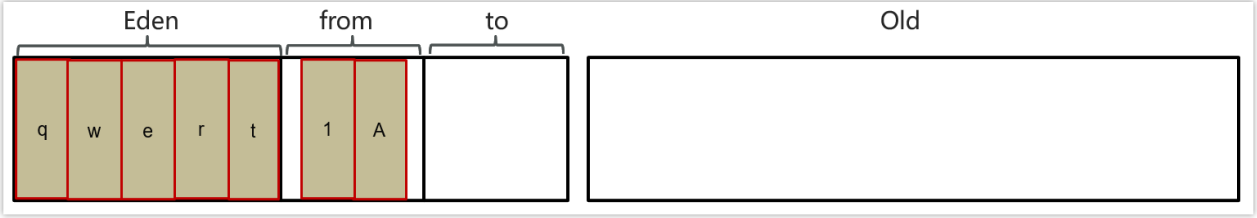
- 新创建的对象，都会先分配到eden区

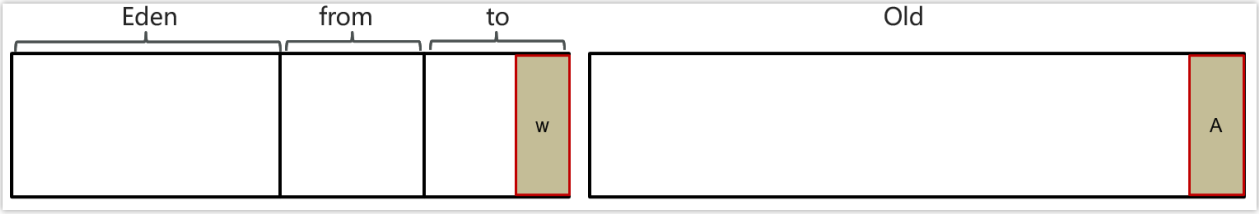


- 当伊甸园内存不足，标记伊甸园与 from（现阶段没有）的存活对象
- 将存活对象采用复制算法复制到 to 中，复制完毕后，伊甸园和 from 内存都得到释放



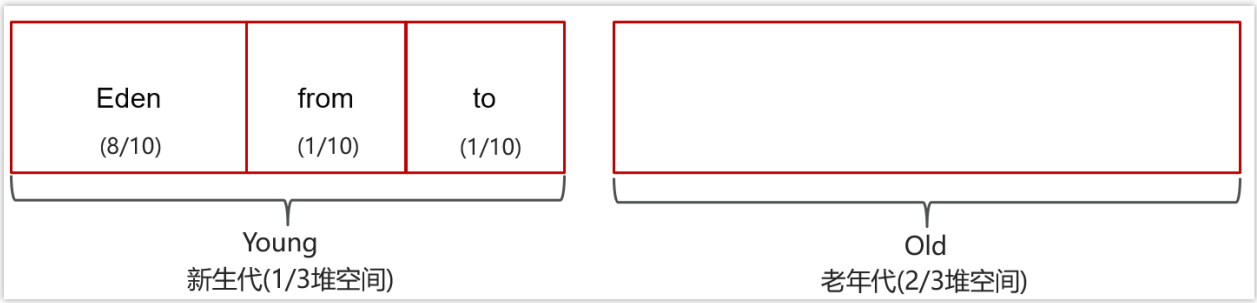
- 经过一段时间后伊甸园的内存又出现不足，标记eden区域to区存活的对象，将存活的对象复制到from区





- 当幸存区对象熬过几次回收（最多15次），晋升到老年代（幸存区内存不足或大对象会导致提前晋升）

MinorGC、 Mixed GC 、 FullGC的区别是什么



- MinorGC【young GC】发生在新生代的垃圾回收，暂停时间短（STW）
- Mixed GC 新生代 + 老年代部分区域的垃圾回收，G1 收集器特有
- FullGC： 新生代 + 老年代完整垃圾回收，暂停时间长（STW），应尽力避免？

> 名词解释：
>
> STW（Stop-The-World）：暂停所有应用程序线程，等待垃圾回收的完成

3.5 说一下 JVM 有哪些垃圾回收器？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆☆

在jvm中，实现了多种垃圾收集器，包括：

- 串行垃圾收集器
- 并行垃圾收集器
- CMS（并发）垃圾收集器

- G1垃圾收集器

3.5.1 串行垃圾收集器

Serial和Serial Old串行垃圾收集器，是指使用单线程进行垃圾回收，堆内存较小，适合个人电脑

- Serial 作用于新生代，采用复制算法
- Serial Old 作用于老年代，采用标记-整理算法

垃圾回收时，只有一个线程在工作，并且java应用中的所有线程都要暂停（STW），等待垃圾回收的完成。

3.5.2 并行垃圾收集器

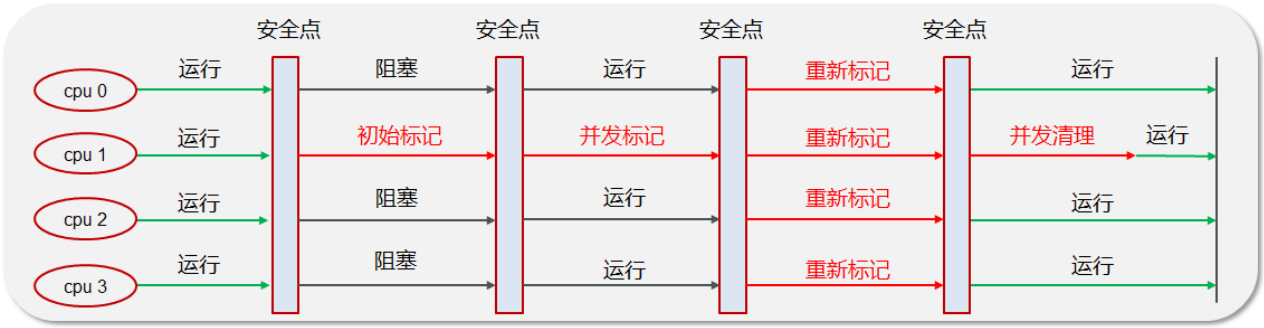
Parallel New和Parallel Old是一个并行垃圾回收器，**JDK8默认使用此垃圾回收器**

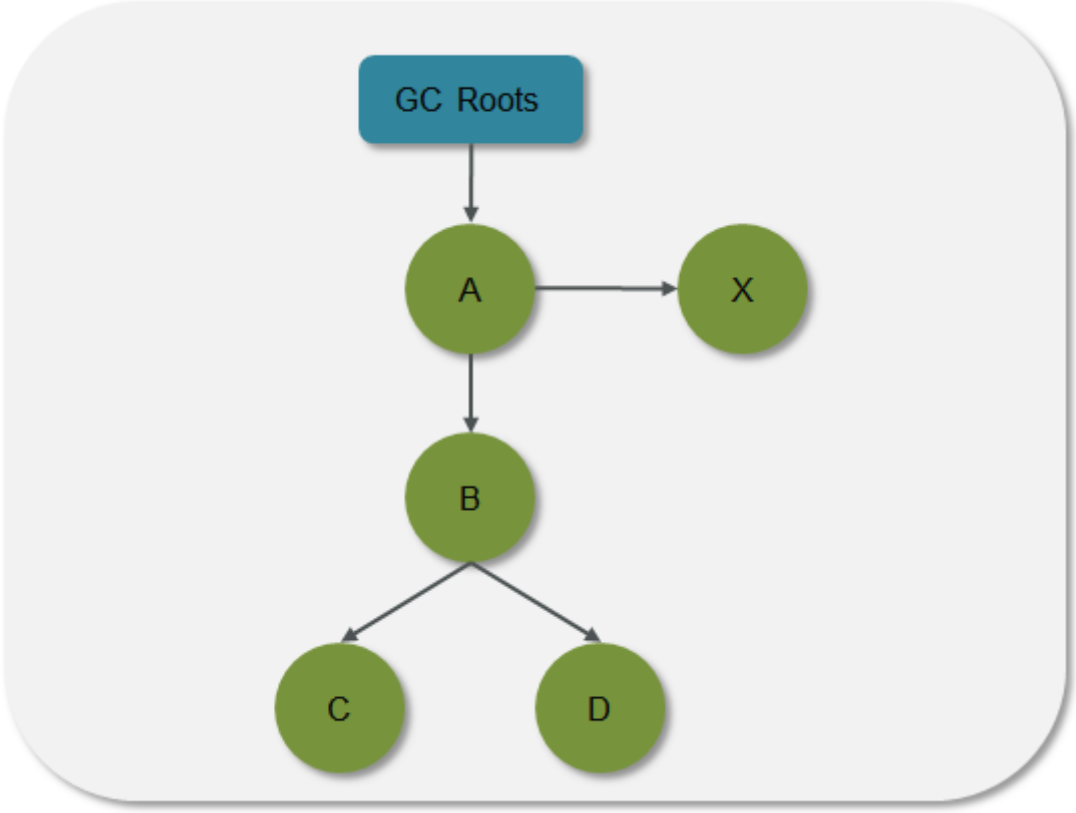
- Parallel New作用于新生代，采用复制算法
- Parallel Old作用于老年代，采用标记-整理算法

垃圾回收时，多个线程在工作，并且java应用中的所有线程都要暂停（STW），等待垃圾回收的完成。

3.5.2 CMS（并发）垃圾收集器

CMS全称 Concurrent Mark Sweep，是一款并发的、使用标记-清除算法的垃圾回收器，该回收器是针对老年代垃圾回收的，是一款以获取最短回收停顿时间为目标的收集器，停顿时间短，用户体验就好。其最大特点是在进行垃圾回收时，应用仍然能正常运行。



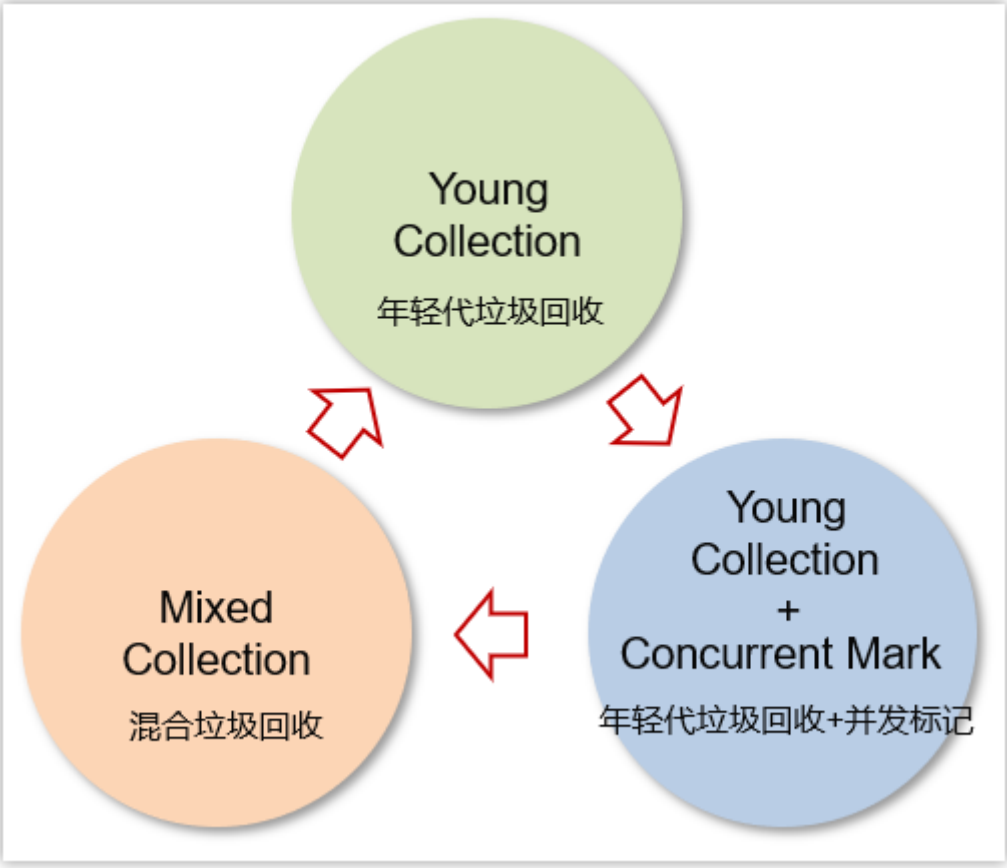


3.6 详细聊一下G1垃圾回收器

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆☆

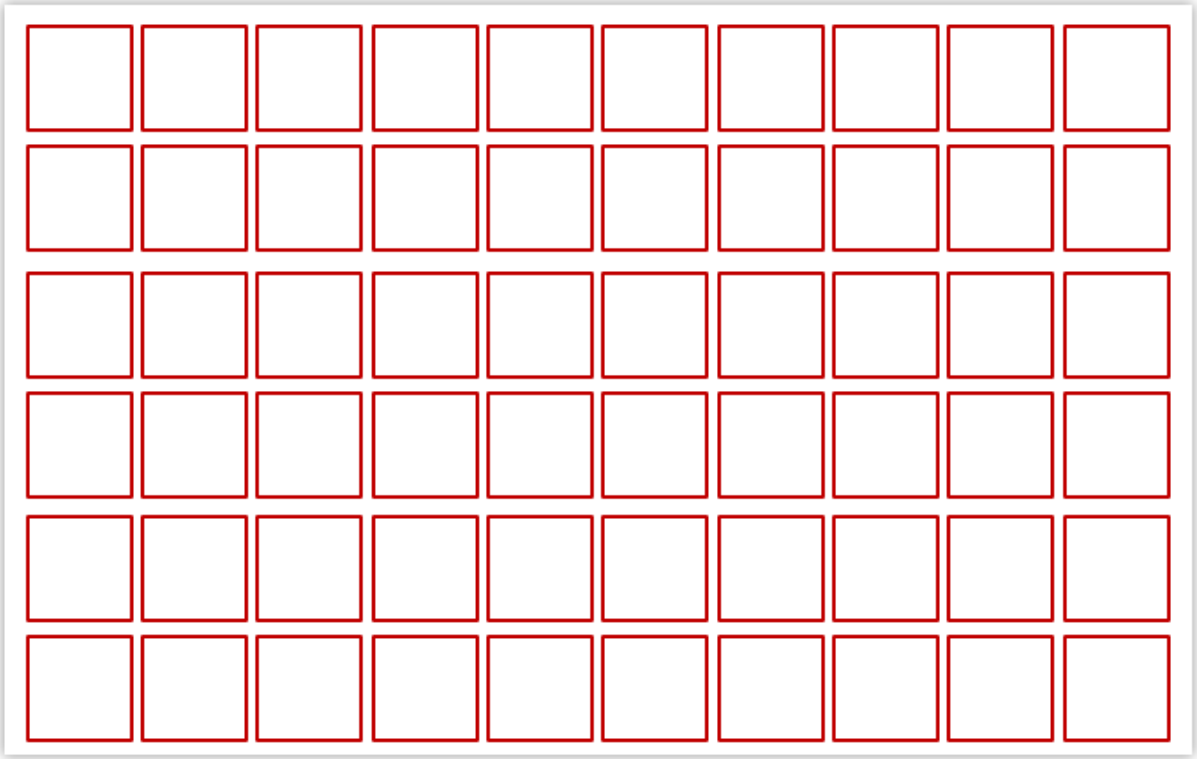
3.6.1 概述

- 应用于新生代和老年代，在**JDK9之后默认使用G1**
- 划分成多个区域，每个区域都可以充当 eden，survivor，old， humongous，其中 humongous 专为大对象准备
- 采用复制算法
- 响应时间与吞吐量兼顾
- 分成三个阶段：新生代回收、并发标记、混合收集
- 如果并发失败（即回收速度赶不上创建新对象速度），会触发 Full GC

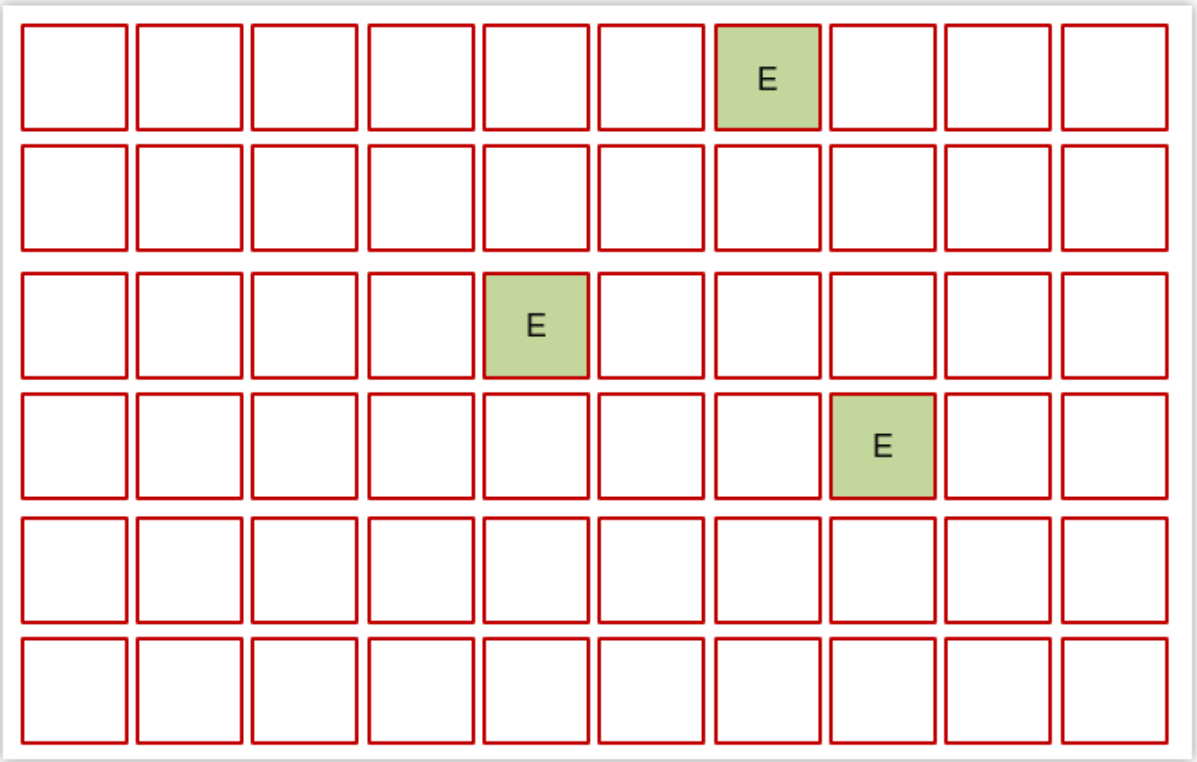


3.6.2 Young Collection(年轻代垃圾回收)

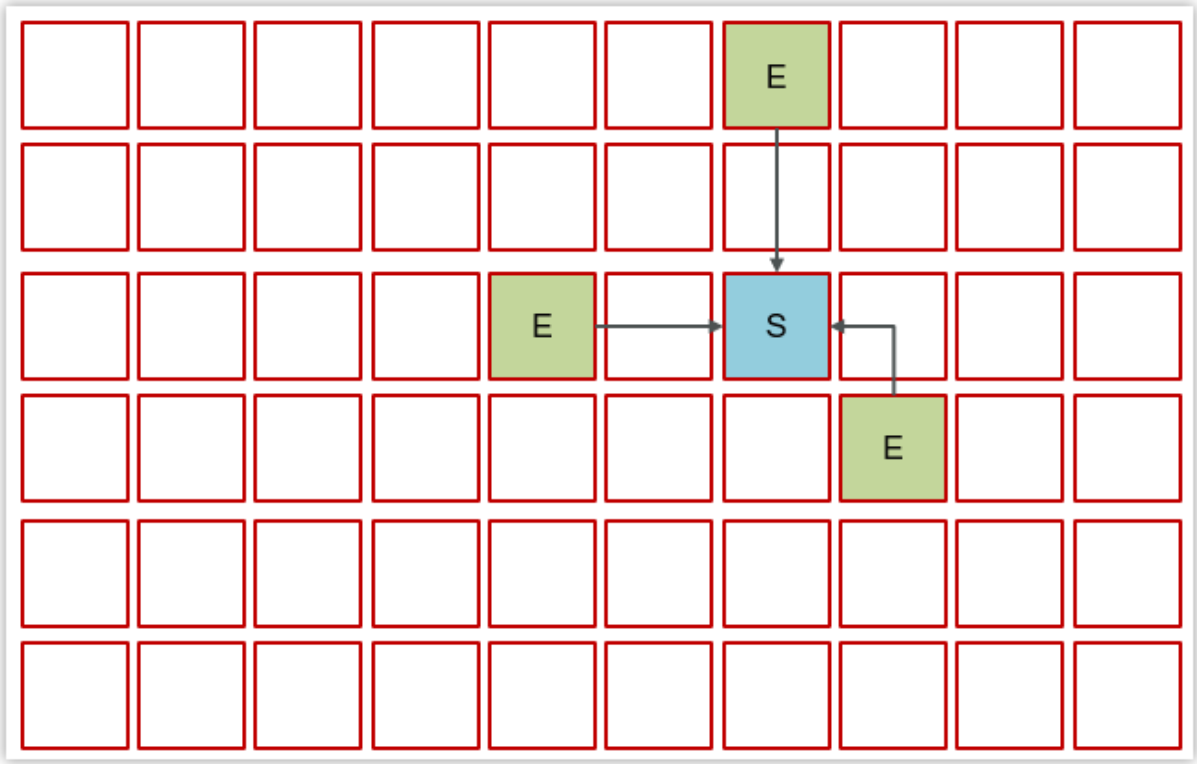
- 初始时，所有区域都处于空闲状态

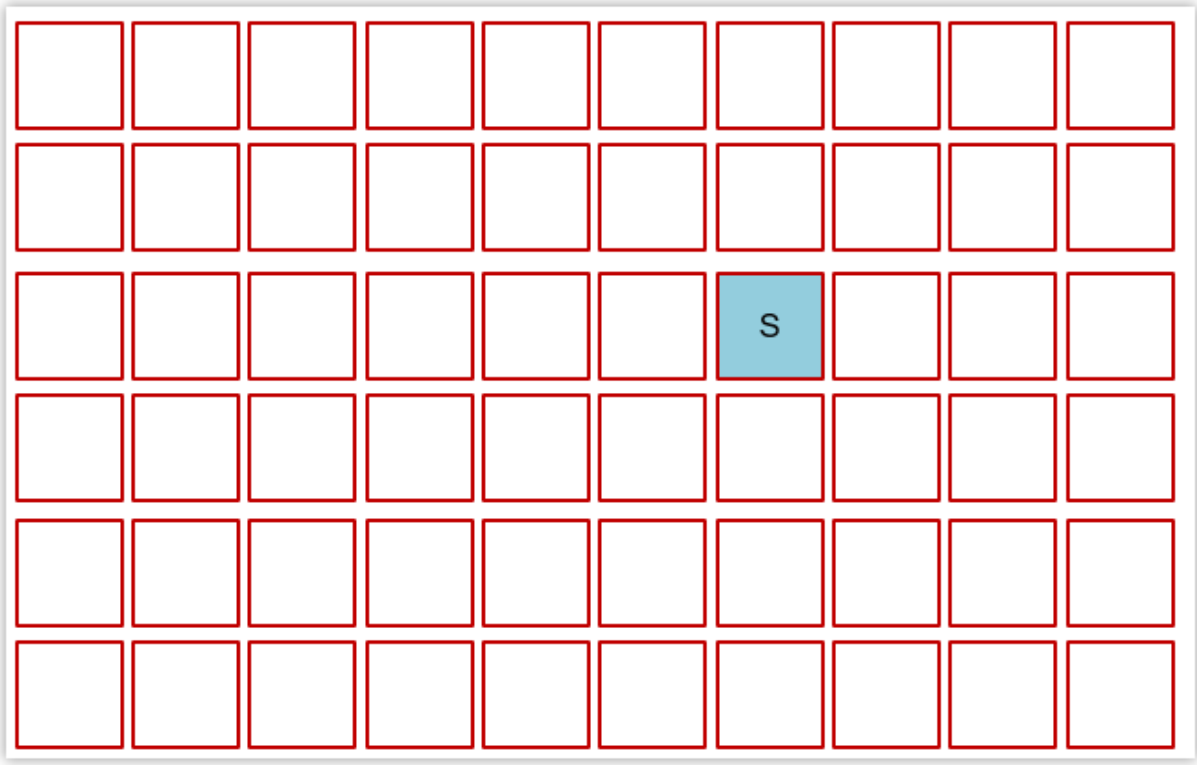


- 创建了一些对象，挑出一些空闲区域作为伊甸园区存储这些对象

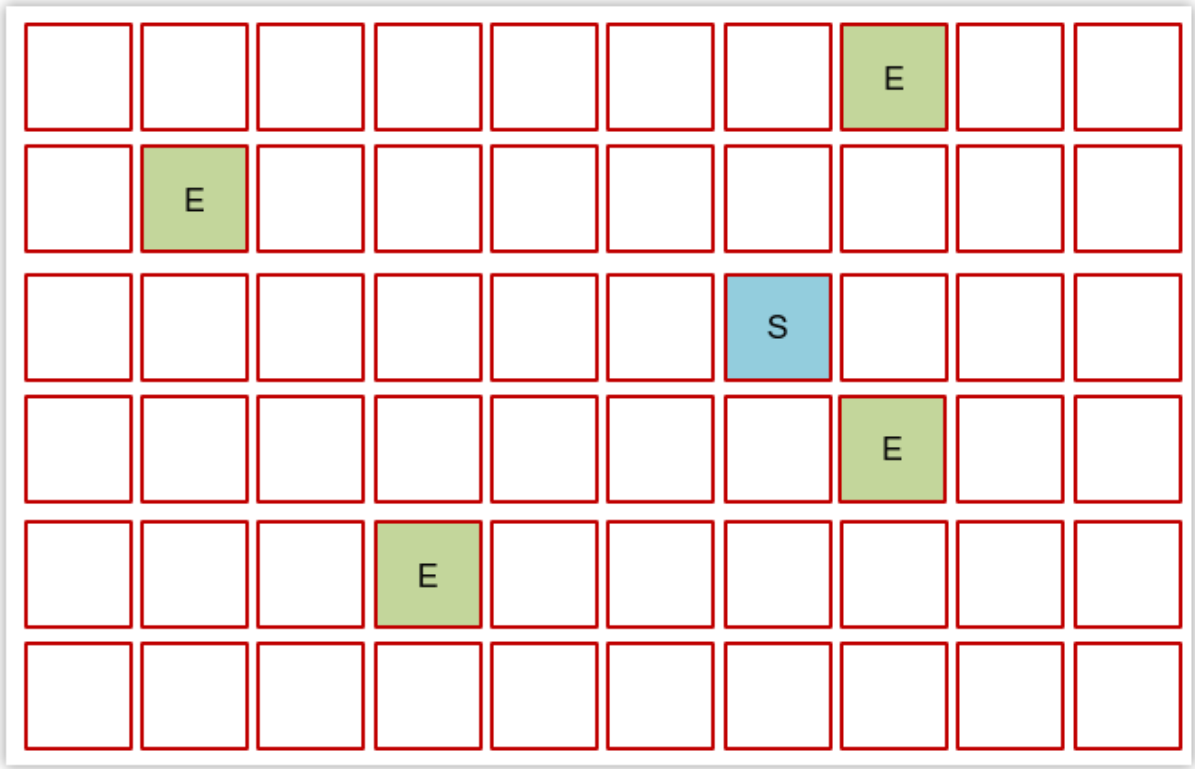


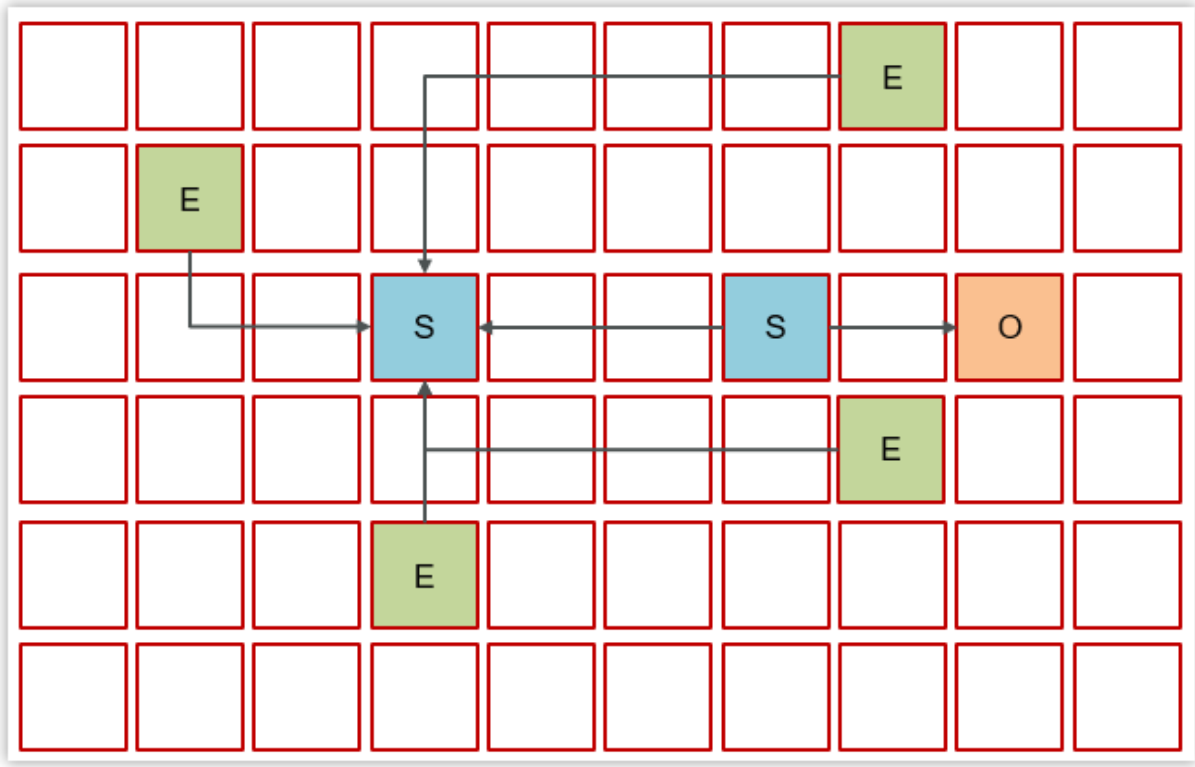
- 当伊甸园需要垃圾回收时，挑出一个空闲区域作为幸存区，用复制算法复制存活对象，需要暂停用户线程

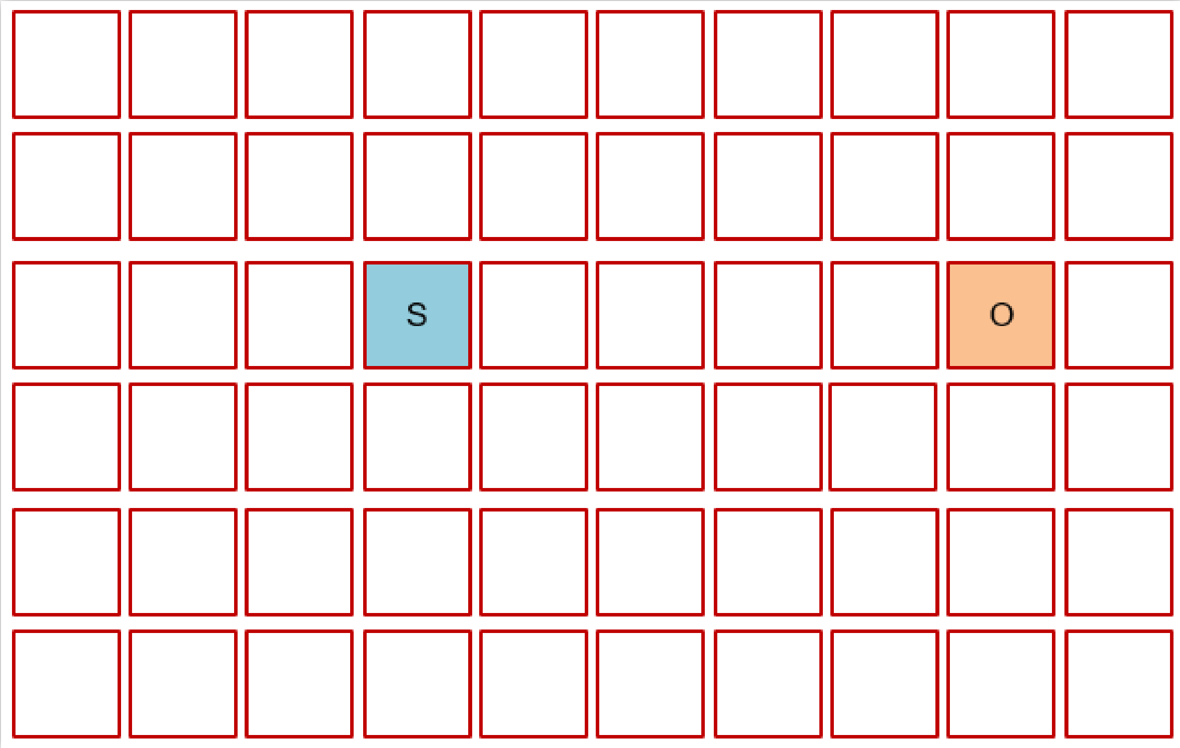




- 随着时间流逝，伊甸园的内存又有不足
- 将伊甸园以及之前幸存区中的存活对象，采用复制算法，复制到新的幸存区，其中较老对象晋升至老年代

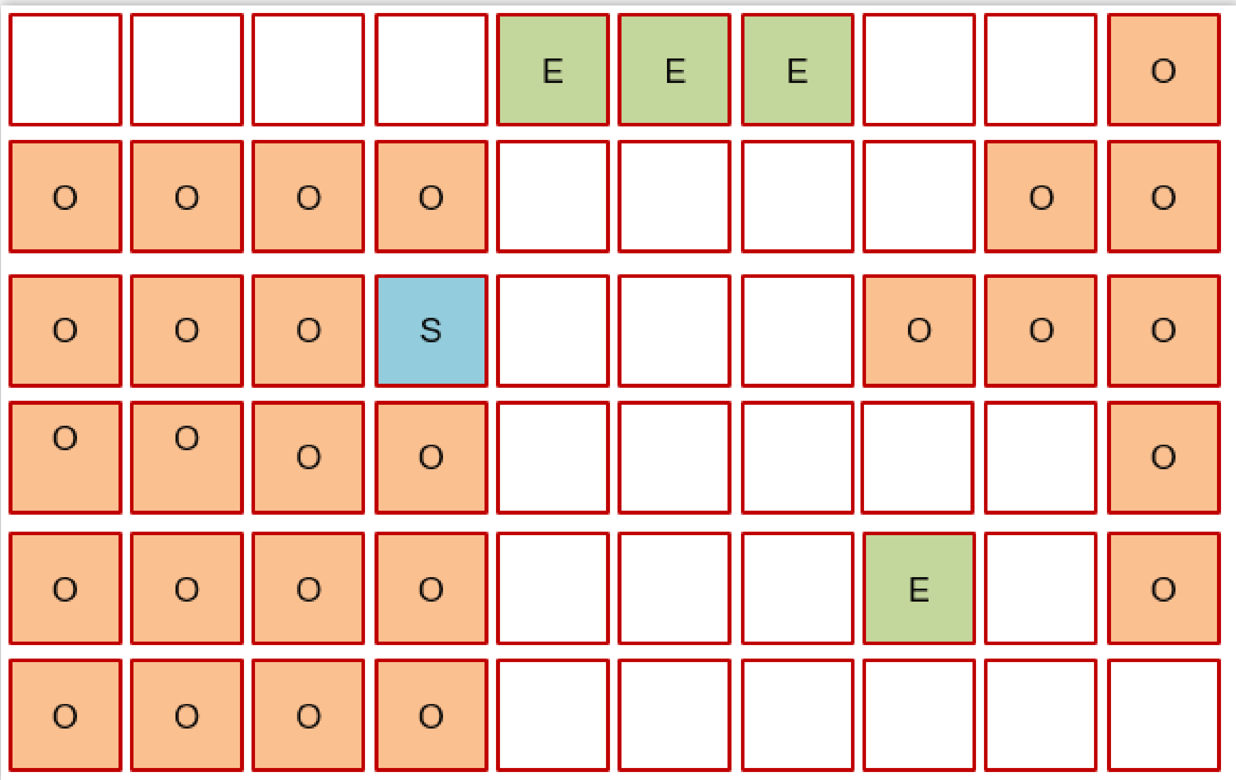

E
E
S
E
E


E
E
S
S
O
E
E

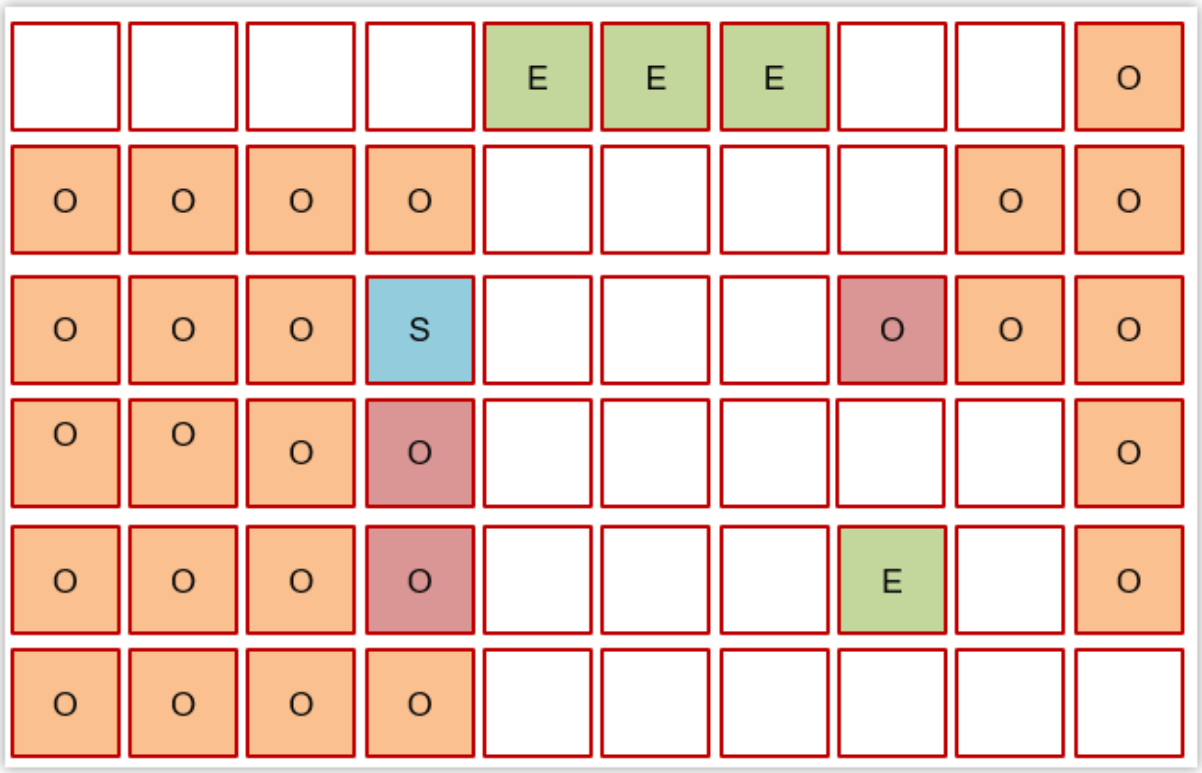


3.6.3 Young Collection + Concurrent Mark (年轻代垃圾回收+并发标记)

当老年代占用内存超过阈值(默认是45%)后，触发并发标记，这时无需暂停用户线程

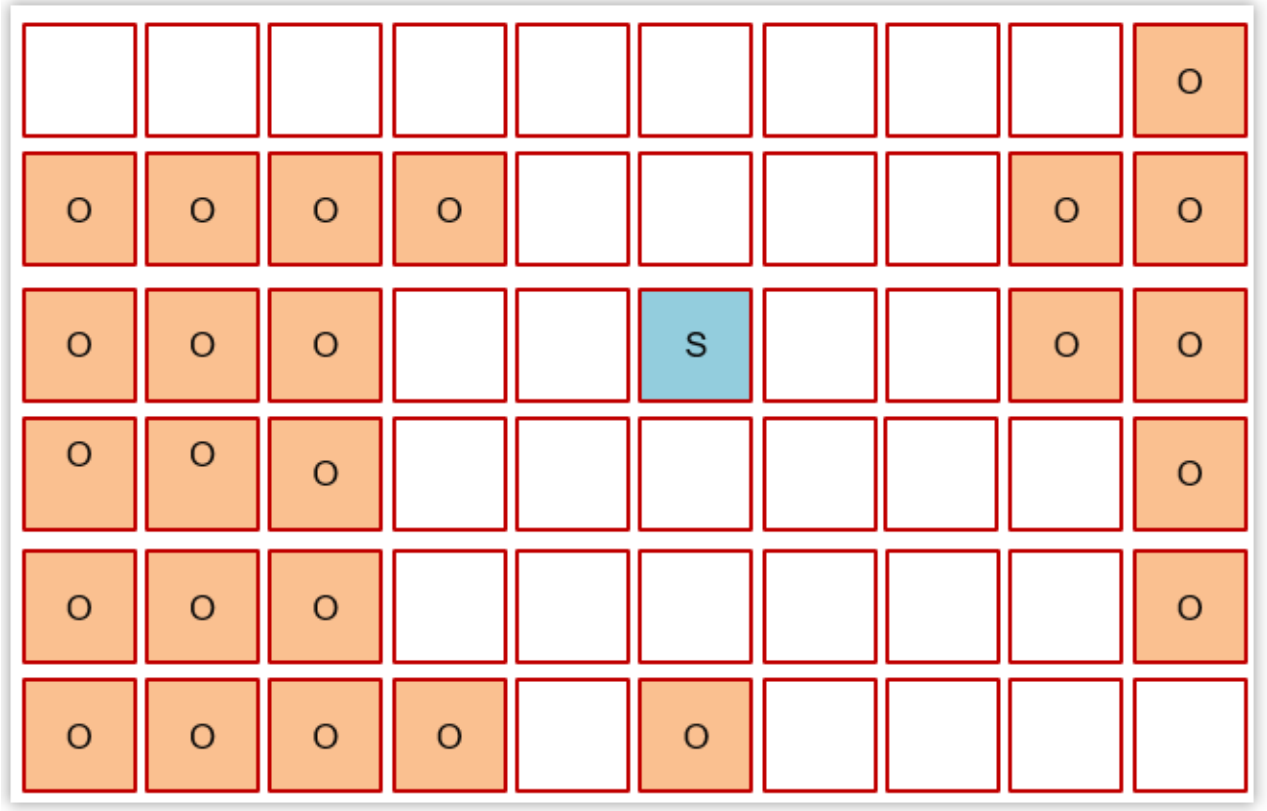


- 并发标记之后，会有重新标记阶段解决漏标问题，此时需要暂停用户线程。
- 这些都完成后就知道了老年代有哪些存活对象，随后进入混合收集阶段。此时不会对所有老年代区域进行回收，而是根据暂停时间目标优先回收价值高（存活对象少）的区域（这也是 Gabage First 名称的由来）。

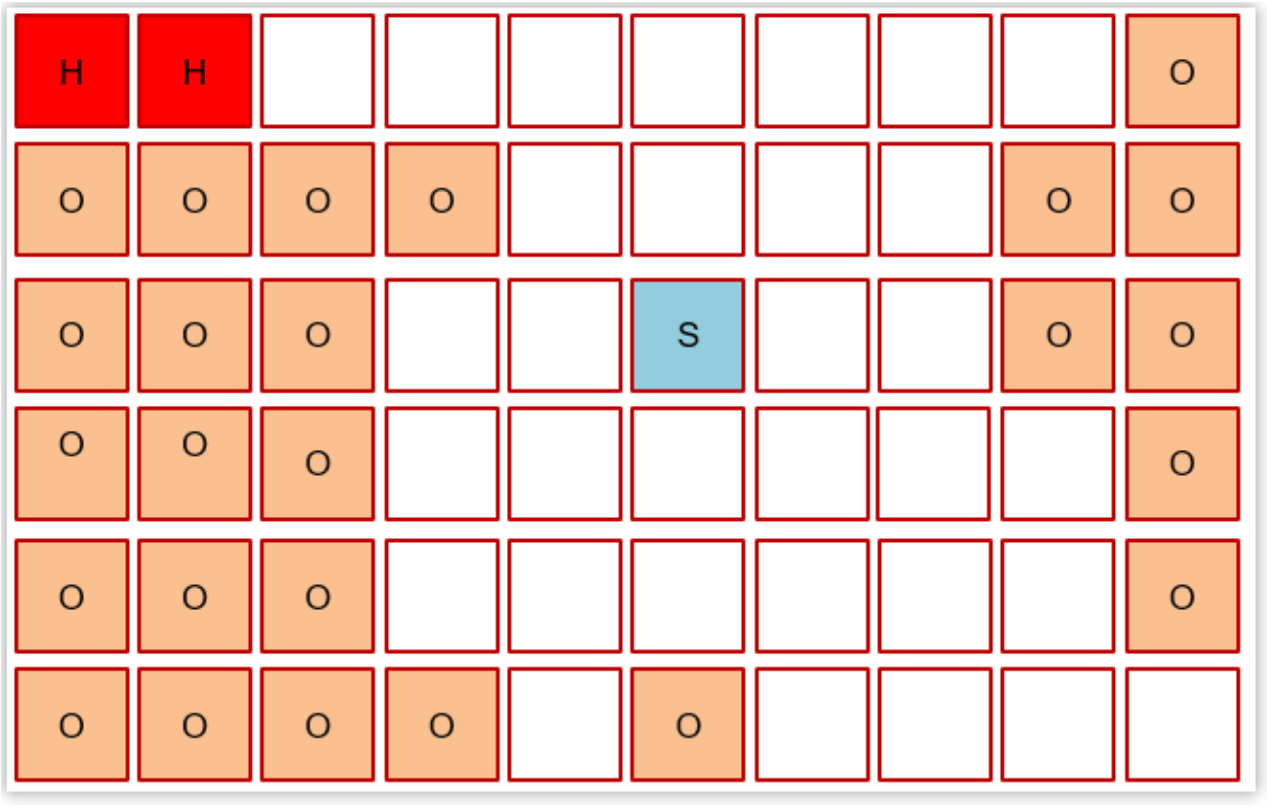


3.6.4 Mixed Collection (混合垃圾回收)

复制完成，内存得到释放。进入下一轮的新生代回收、并发标记、混合收集



其中H叫做巨型对象，如果对象非常大，会开辟一块连续的空间存储巨型对象



3.7 强引用、软引用、弱引用、虚引用的区别？

> 难易程度：☆☆☆☆

> 出现频率：☆☆☆

3.7.1 强引用

强引用：只有所有 GC Roots 对象都不通过【强引用】引用该对象，该对象才能被垃圾回收

```
User user = new User();
```

3.7.2 软引用

软引用：仅有软引用引用该对象时，在垃圾回收后，内存仍不足时会再次出发垃圾回收

```
User user = new User();
SoftReference softReference = new SoftReference(user);
```

3.7.3 弱引用

弱引用：仅有弱引用引用该对象时，在垃圾回收时，无论内存是否充足，都会回收弱引用对象

```
User user = new User();
WeakReference weakReference = new WeakReference(user);
```

> 延伸话题：ThreadLocal内存泄漏问题

ThreadLocal用的就是弱引用，看以下源码：

```
static class Entry extends WeakReference<ThreadLocal<?>> {
    Object value;

    Entry(ThreadLocal<?> k, Object v) {
         super(k);
         value = v; //强引用，不会被回收
     }
}
```

`Entry`的key是当前ThreadLocal，value值是我们要设置的数据。

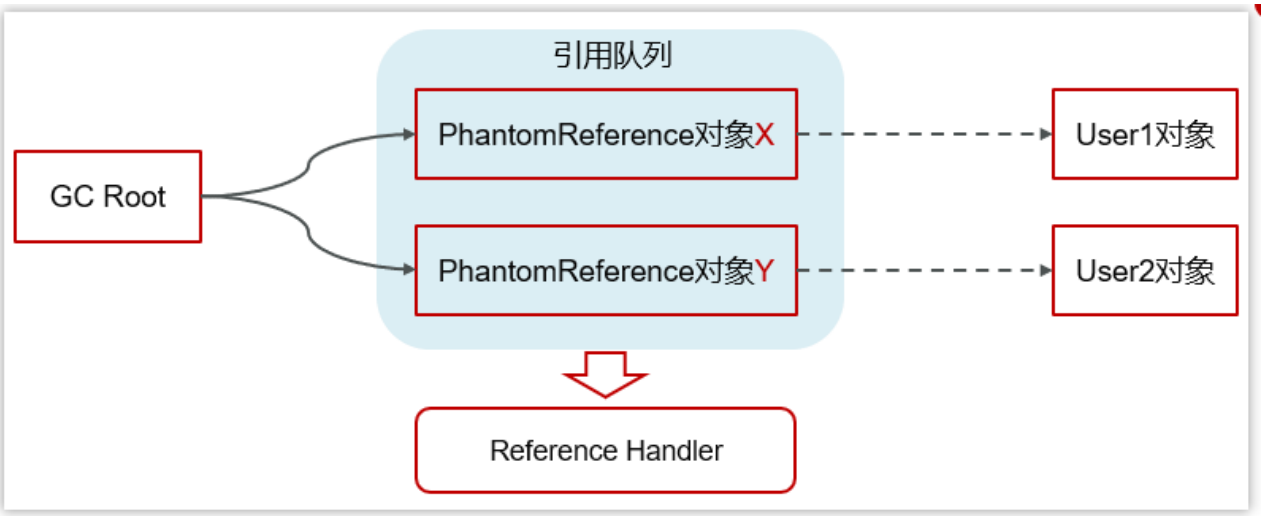
`WeakReference`表示的是弱引用，当JVM进行GC时，一旦发现了只具有弱引用的对象，不管当前内存空间是否足够，都会回收它的内存。但是`value`是强引用，它不会被回收掉。

> ThreadLocal使用建议：使用完毕后注意调用清理方法。

3.7.4 虚引用

虚引用：必须配合引用队列使用，被引用对象回收时，会将虚引用入队，由Reference Handler 线程调用虚引用相关方法释放直接内存

```
User user = new User();
ReferenceQueue referenceQueue = new ReferenceQueue();
PhantomReference phantomReference = new PhantomReference(user,queue);
```



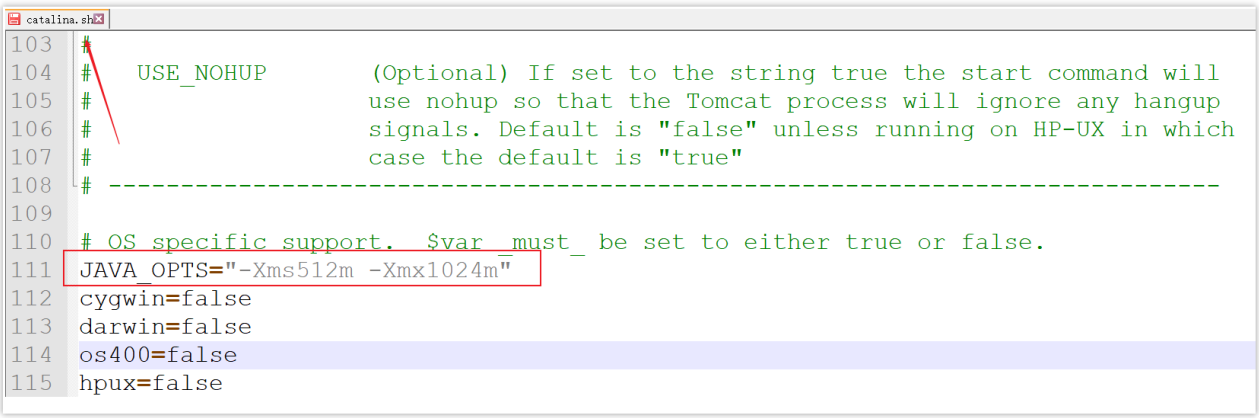
4.1 JVM 调优的参数可以在哪里设置参数值？

> 难易程度：☆☆
>
> 出现频率：☆☆☆

4.1.1 tomcat的设置vm参数

修改TOMCAT_HOME/bin/catalina.sh文件，如下图

```
JAVA_OPTS="-Xms512m -Xmx1024m"
```



4.1.2 springboot项目jar文件启动

通常在linux系统下直接加参数启动springboot项目

```
nohup java -Xms512m -Xmx1024m -jar xxxx.jar --
spring.profiles.active=prod &
```

> nohup：用于在系统后台不挂断地运行命令，退出终端不会影响程序的运行
>
> 参数 **&** ：让命令在后台执行，终端退出后命令仍旧执行。

4.2 用的 JVM 调优的参数都有哪些？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

对于JVM调优，主要就是调整年轻代、年老大、元空间的内存空间大小及使用的垃圾回收器类型。

https://www.oracle.com/java/technologies/javase/vmoptions-jsp.html

1）设置堆的初始大小和最大大小，为了防止垃圾收集器在初始大小、最大大小之间收缩堆而产生额外的时间，通常把最大、初始大小设置为相同的值。

```
-Xms：设置堆的初始化大小

-Xmx：设置堆的最大大小
```

2） 设置年轻代中Eden区和两个Survivor区的大小比例。该值如果不设置，则默认比例为8:1:1。Java官方通过增大Eden区的大小，来减少YGC发生的次数，但有时我们发现，虽然次数减少了，但Eden区满

的时候，由于占用的空间较大，导致释放缓慢，此时STW的时间较长，因此需要按照程序情况去调优。

```
-XXSurvivorRatio=3，表示年轻代中的分配比率：survivor:eden = 2:3
```

3）年轻代和老年代默认比例为1：2。可以通过调整二者空间大小比率来设置两者的大小。

```
-XX:newSize    设置年轻代的初始大小
-XX:MaxNewSize    设置年轻代的最大大小，    初始大小和最大大小两个值通常相同
```

4）线程堆栈的设置：**每个线程默认会开启1M的堆栈**，用于存放栈帧、调用参数、局部变量等，但一般256K就够用。通常减少每个线程的堆栈，可以产生更多的线程，但这实际上还受限于操作系统。

```
-Xss    对每个线程stack大小的调整,-Xss128k
```

5）一般来说，当survivor区不够大或者占用量达到50%，就会把一些对象放到老年区。通过设置合理的eden区，survivor区及使用率，可以将年轻对象保存在年轻代，从而避免full GC，使用-Xmn设置年轻代的大小

6）系统CPU持续飙高的话，首先先排查代码问题，如果代码没问题，则咨询运维或者云服务器供应商，通常服务器重启或者服务器迁移即可解决。

7）对于占用内存比较多的大对象，一般会选择在老年代分配内存。如果在年轻代给大对象分配内存，年轻代内存不够了，就要在eden区移动大量对象到老年代，然后这些移动的对象可能很快消亡，因此导致full GC。通过设置参数：-XX:PetenureSizeThreshold=1000000，单位为B，标明对象大小超过1M时，在老年代(tenured)分配内存空间。

8）一般情况下，年轻对象放在eden区，当第一次GC后，如果对象还存活，放到survivor区，此后，每GC一次，年龄增加1，当对象的年龄达到阈值，就被放到tenured老年区。这个阈值可以同构-XX:MaxTenuringThreshold设置。如果想让对象留在年轻代，可以设置比较大的阈值。

```
（1）-XX:+UseParallelGC:年轻代使用并行垃圾回收收集器。这是一个关注吞吐量的收集器，可以尽可能的减少垃圾回收时间。

（2）-XX:+UseParallelOldGC:设置老年代使用并行垃圾回收收集器。
```

9）尝试使用大的内存分页：使用大的内存分页增加CPU的内存寻址能力，从而系统的性能。

```
-XX:+LargePageSizeInBytes 设置内存页的大小
```

10）使用非占用的垃圾收集器。

```
-XX:+UseConcMarkSweepGC老年代使用CMS收集器降低停顿。
```

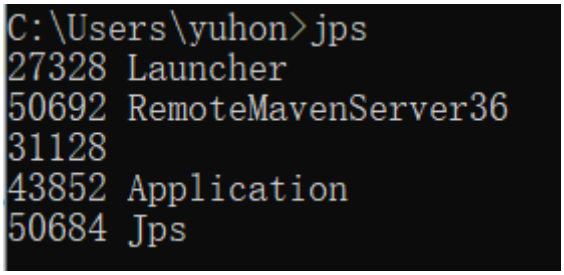
4.3 说一下 JVM 调优的工具？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆☆

4.3.1 命令工具

4.3.1.1 jps（Java Process Status）

输出JVM中运行的进程状态信息(现在一般使用jconsole)



4.3.1.2 jstack

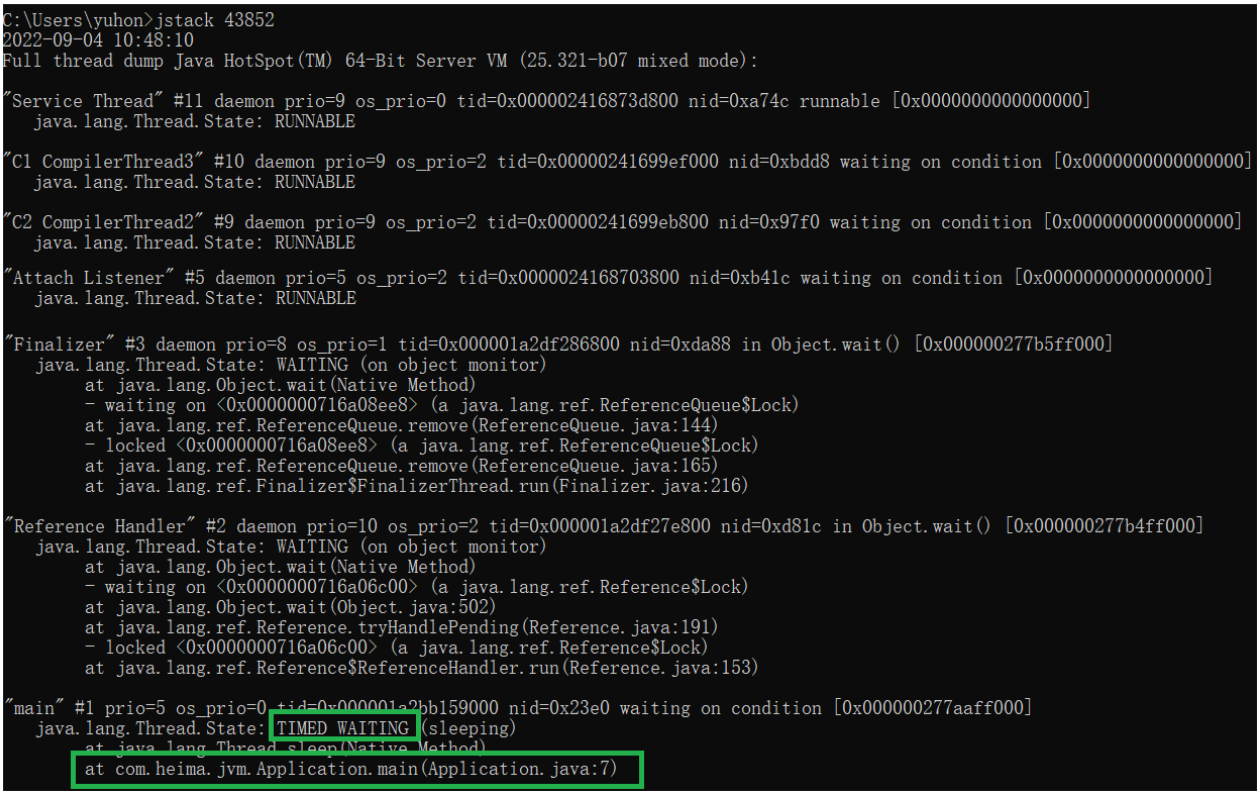
查看java进程内**线程的堆栈**信息。

```
jstack [option] <pid>
```

java案例

```
package com.heima.jvm;

public class Application {

    public static void main(String[] args) throws
InterruptedException {
        while (true){
            Thread.sleep(1000);
            System.out.println("哈哈哈");
        }
    }
}
```

使用jstack查看进行堆栈运行信息



4.3.1.3 jmap

用于生成堆转存快照

> jmap [options] pid 内存映像信息
>
> jmap -heap pid 显示Java堆的信息
>
> jmap -dump:format=b,file=heap.hprof pid
>
> format=b表示以hprof二进制格式转储Java堆的内存
> file=用于指定快照dump文件的文件名。

例：显示了某一个java运行的堆信息

```
C:\Users\yuhon>jmap -heap 53280
Attaching to process ID 53280, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.321-b07

using thread-local object allocation.
Parallel GC with 8 thread(s)    //并行的垃圾回收器
```

```
Heap Configuration:  //堆配置
   MinHeapFreeRatio         = 0   //空闲堆空间的最小百分比
   MaxHeapFreeRatio         = 100  //空闲堆空间的最大百分比
   MaxHeapSize              = 8524922880 (8130.0MB) //堆空间允许的最大值
   NewSize                  = 178257920 (170.0MB) //新生代堆空间的默认值
   MaxNewSize               = 2841640960 (2710.0MB) //新生代堆空间允许的最大值
   OldSize                  = 356515840 (340.0MB) //老年代堆空间的默认值
   NewRatio                 = 2 //新生代与老年代的堆空间比值，表示新生代：老年代=1：2
   SurvivorRatio            = 8 //两个Survivor区和Eden区的堆空间比值为8,表示S0:S1:Eden=1:1:8
   MetaspaceSize            = 21807104 (20.796875MB) //元空间的默认值
   CompressedClassSpaceSize = 1073741824 (1024.0MB) //压缩类使用空间大小
   MaxMetaspaceSize         = 17592186044415 MB //元空间允许的最大值
   G1HeapRegionSize         = 0 (0.0MB)//在使用 G1 垃圾回收算法时，JVM 会将 Heap 空间分隔为若干个 Region，该参数用来指定每个 Region 空间的大小。

Heap Usage:
PS Young Generation
Eden Space: //Eden使用情况
   capacity = 134217728 (128.0MB)
   used     = 10737496 (10.240074157714844MB)
   free     = 123480232 (117.75992584228516MB)
   8.000057935714722% used
From Space: //Survivor-From 使用情况
   capacity = 22020096 (21.0MB)
   used     = 0 (0.0MB)
   free     = 22020096 (21.0MB)
   0.0% used
To Space: //Survivor-To 使用情况
   capacity = 22020096 (21.0MB)
   used     = 0 (0.0MB)
   free     = 22020096 (21.0MB)
   0.0% used
PS Old Generation  //老年代 使用情况
```

```
   capacity = 356515840 (340.0MB)
   used     = 0 (0.0MB)
   free     = 356515840 (340.0MB)
   0.0% used

3185 interned Strings occupying 261264 bytes.
```

4.3.1.4 jhat

用于分析jmap生成的堆转存快照（一般不推荐使用，而是使用Ecplise Memory Analyzer）

4.3.1.5 jstat

是JVM统计监测工具。可以用来显示垃圾回收信息、类加载信息、新生代统计信息等。

常见参数：

①总结垃圾回收统计

```
jstat -gcutil pid
```

```
C:\Users\yuhon>jstat -gcutil 53280
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00   8.00   0.00  17.38  19.94       0    0.000      0    0.000    0.000
```

字段	含义
S0	幸存1区当前使用比例
S1	幸存2区当前使用比例
E	伊甸园区使用比例
O	老年代使用比例
M	元数据区使用比例
CCS	压缩使用比例
YGC	年轻代垃圾回收次数

字段	含义
YGCT	年轻代垃圾回收消耗时间
FGC	老年代垃圾回收次数
FGCT	老年代垃圾回收消耗时间
GCT	垃圾回收消耗总时间

②垃圾回收统计

```
jstat -gc pid
```

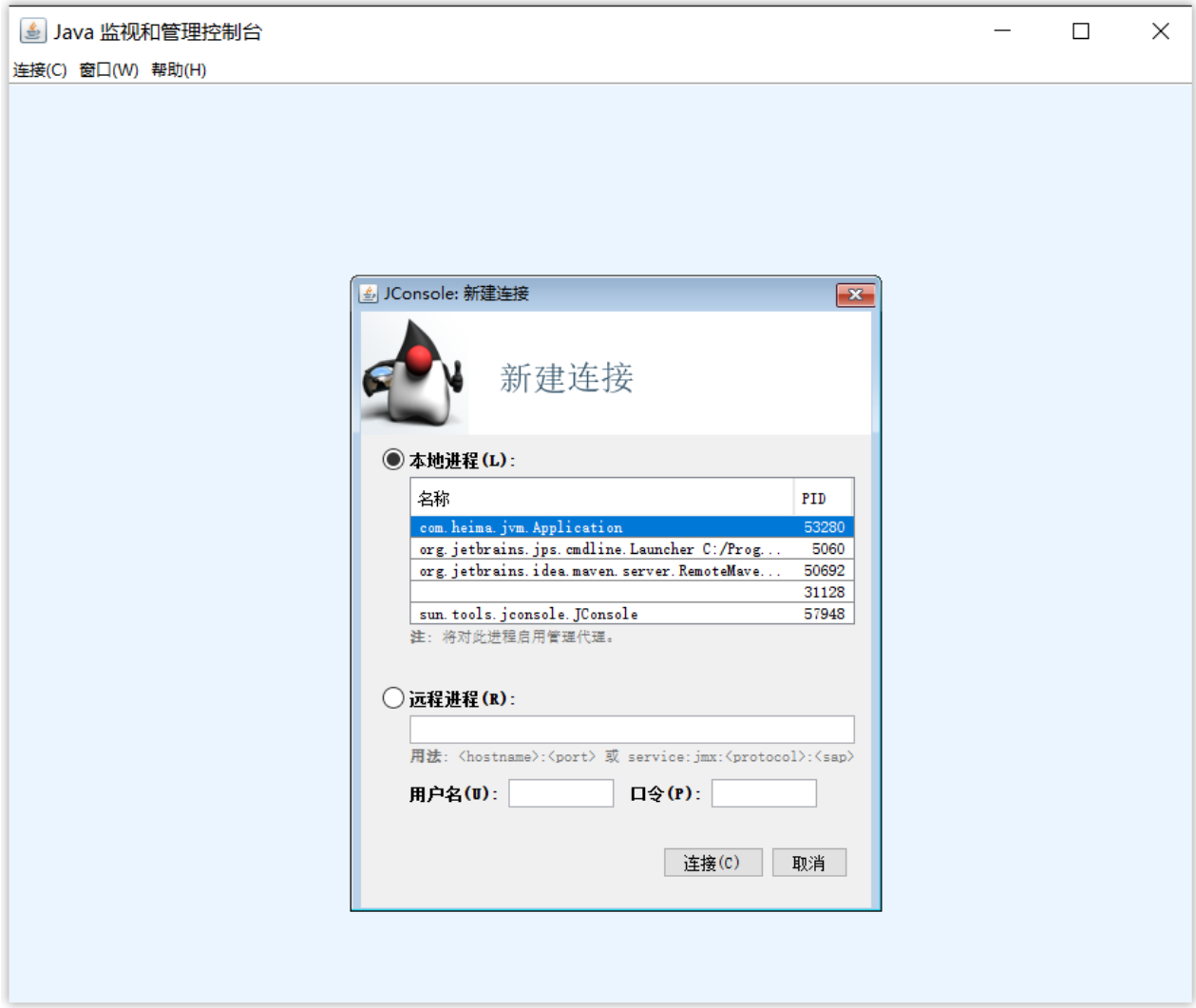
```
C:\Users\yuhon>jstat -gc 53280
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
21504.0 21504.0  0.0    0.0   131072.0 10485.8   348160.0     0.0     4480.0 778.5  384.0   76.6       0    0.000   0      0.000    0.000
```

4.3.2 可视化工具

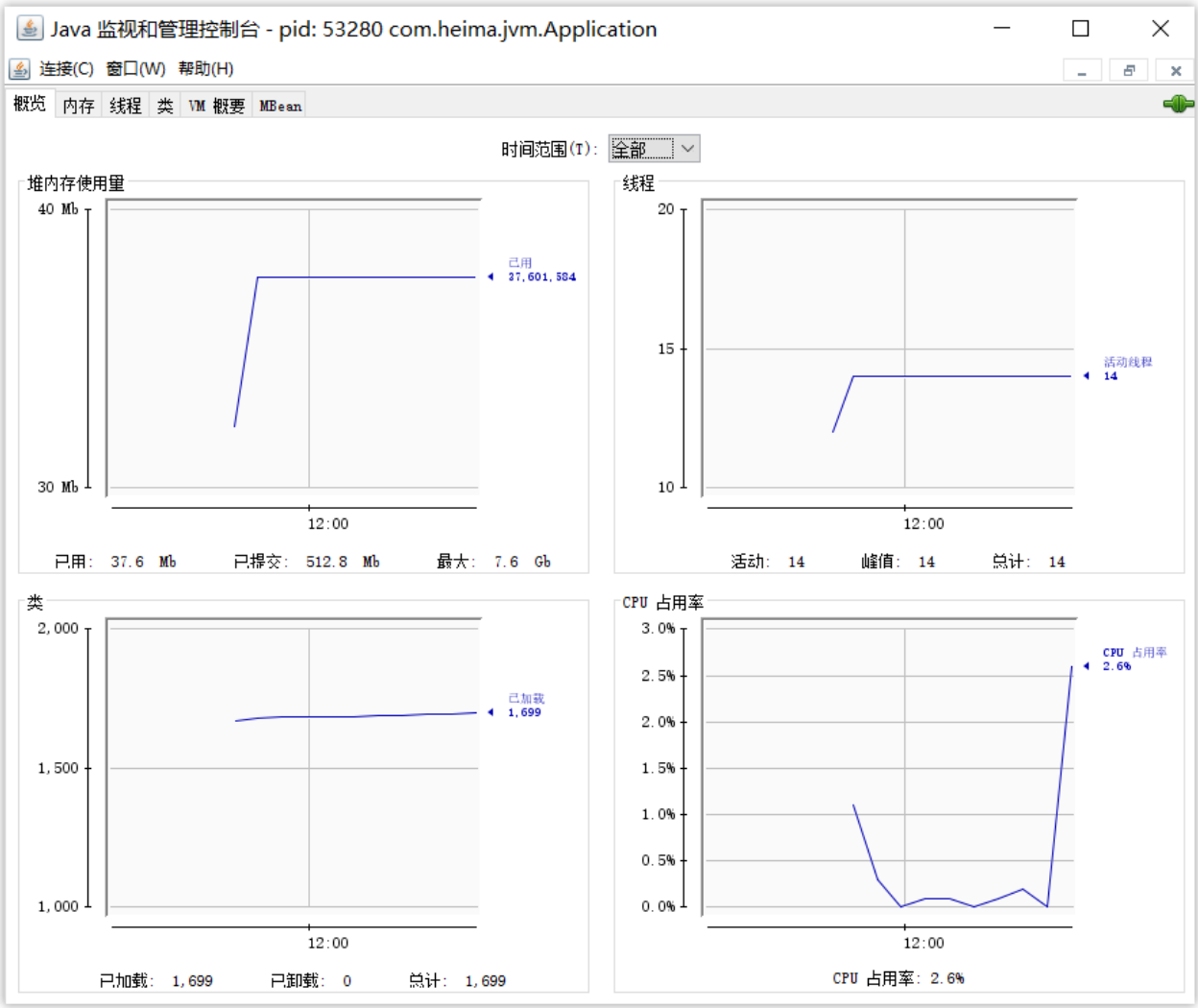
4.3.2.1 jconsole

用于对jvm的内存，线程，类 的监控，是一个基于 jmx 的 GUI 性能监控工具

打开方式：java 安装目录 bin目录下 直接启动 jconsole.exe 就行


Java 监视和管理控制台
连接(C) 窗口(W) 帮助(H)
JConsole: 新建连接
新建连接
本地进程(L):
名称
PID
com.heima.jvm.Application 53280
org.jetbrains.jps.cmdline.Launcher C:/Prog... 5060
org.jetbrains.idea.maven.server.RemoteMave... 50692
31128
sun.tools.jconsole.JConsole 57948
注：将对此进程启用管理代理。
远程进程(R):
用法：<hostname>:<port> 或 service:jmx:<protocol>:<sap>
用户名(U):
口令(P):
连接(C)
取消

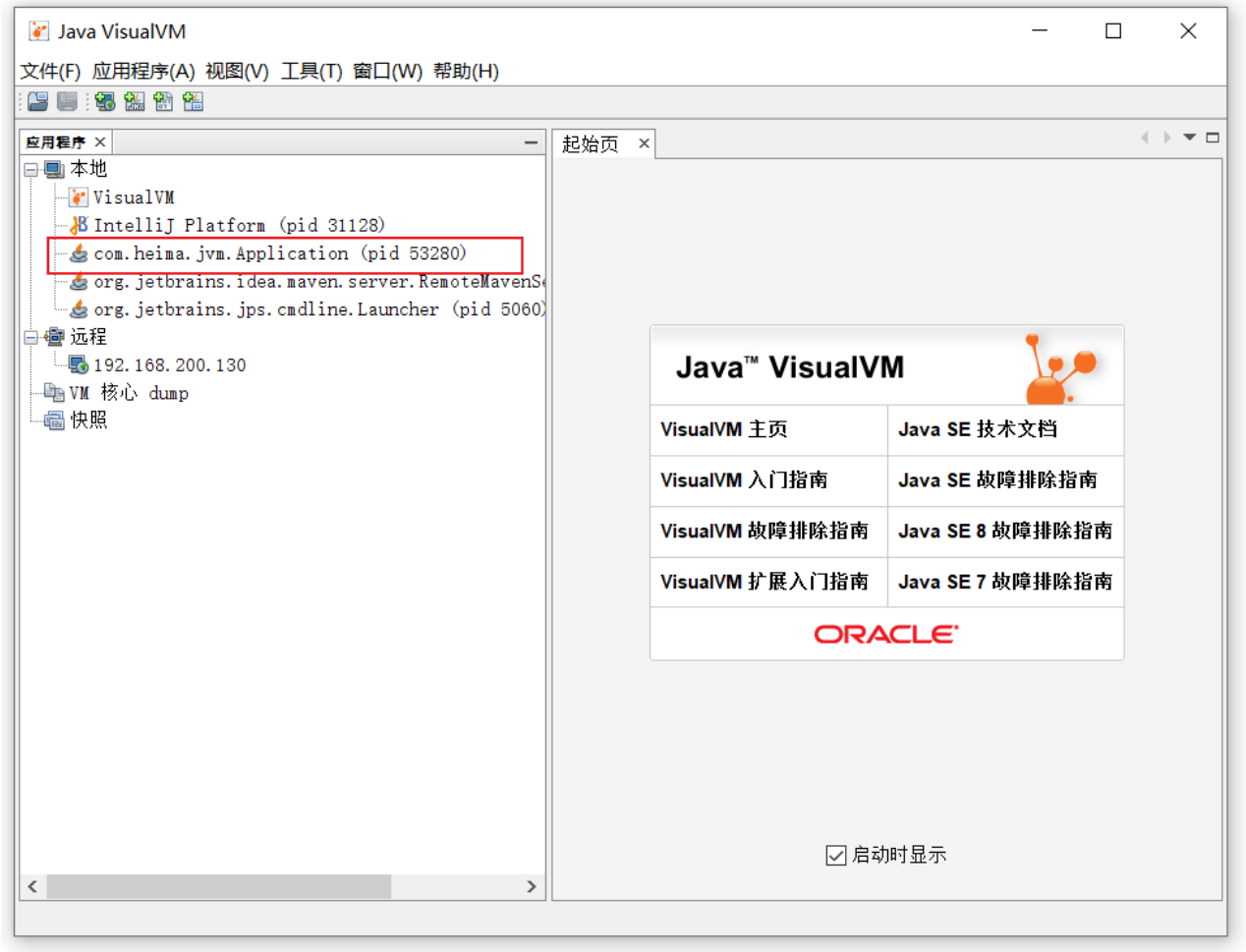
可以内存、线程、类等信息



4.3.2.2 VisualVM：故障处理工具

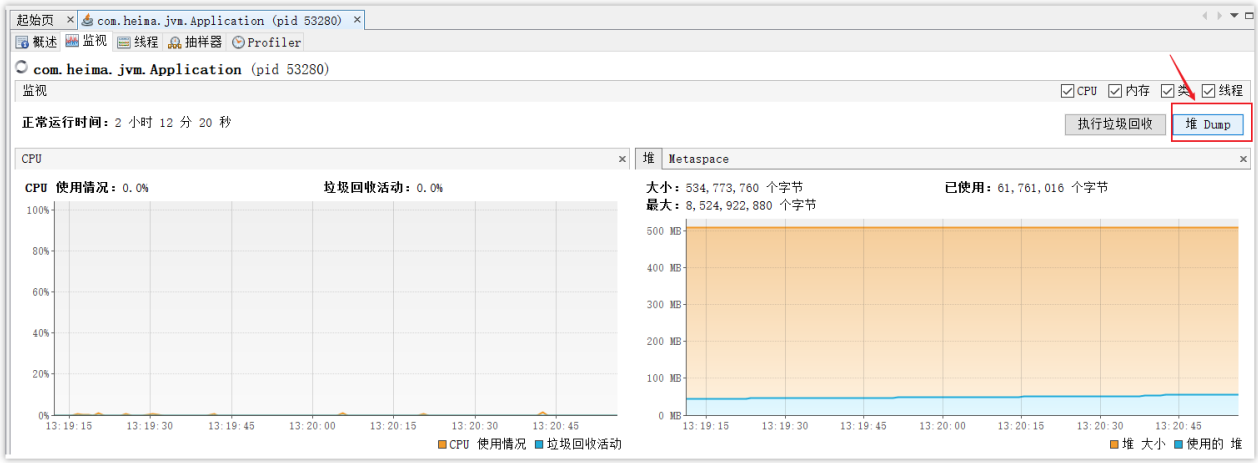
能够监控线程，内存情况，查看方法的CPU时间和内存中的对 象，已被GC的对象，反向查看分配的堆栈

打开方式：java 安装目录 bin目录下 直接启动 jvisualvm.exe就行

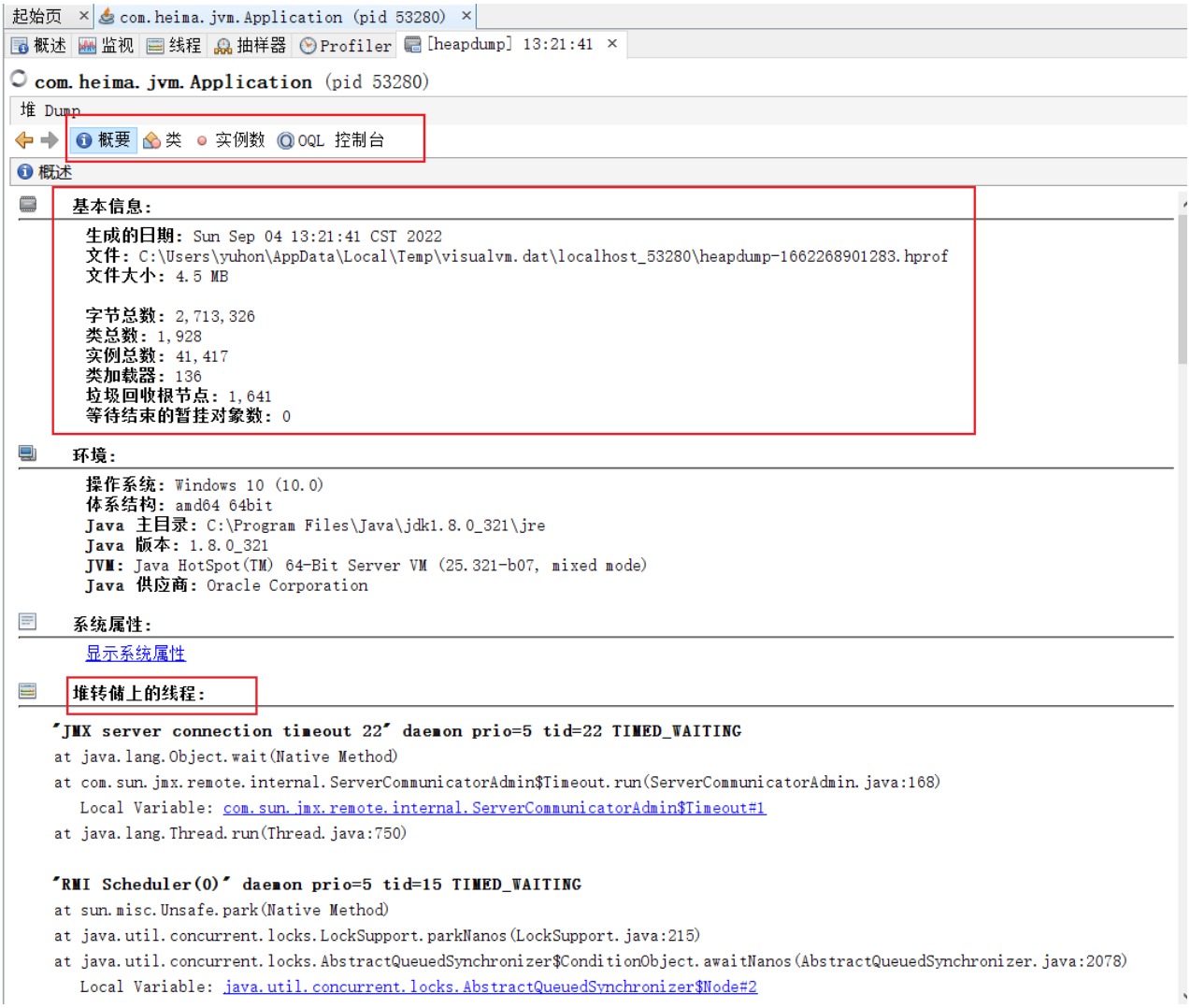


监控程序运行情况

查看运行中的dump



查看堆中的信息



4.4 java内存泄露的排查思路？

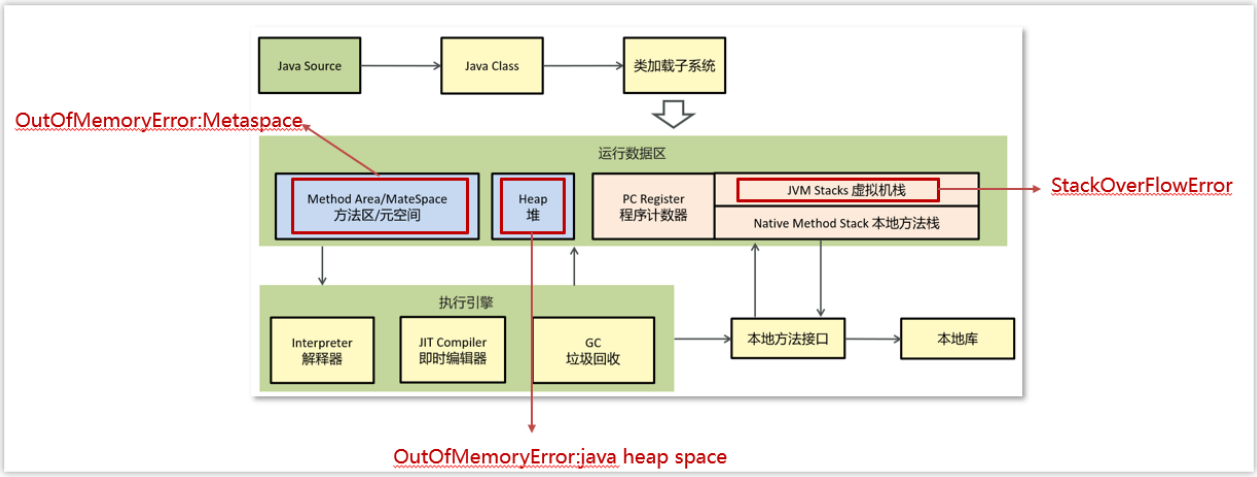
> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆☆

原因：

如果线程请求分配的栈容量超过java虚拟机栈允许的最大容量的时候，java虚拟机将抛出一个StackOverFlowError异常

如果java虚拟机栈可以动态拓展，并且扩展的动作已经尝试过，但是目前无法申请到足够的内存去完成拓展，或者在建立新线程的时候没有足够的内存去创建对应的虚拟机栈，那java虚拟机将会抛出一个OutOfMemoryError异常

如果一次加载的类太多，元空间内存不足，则会报OutOfMemoryError: Metaspace



1、通过jmap指定打印他的内存快照 dump

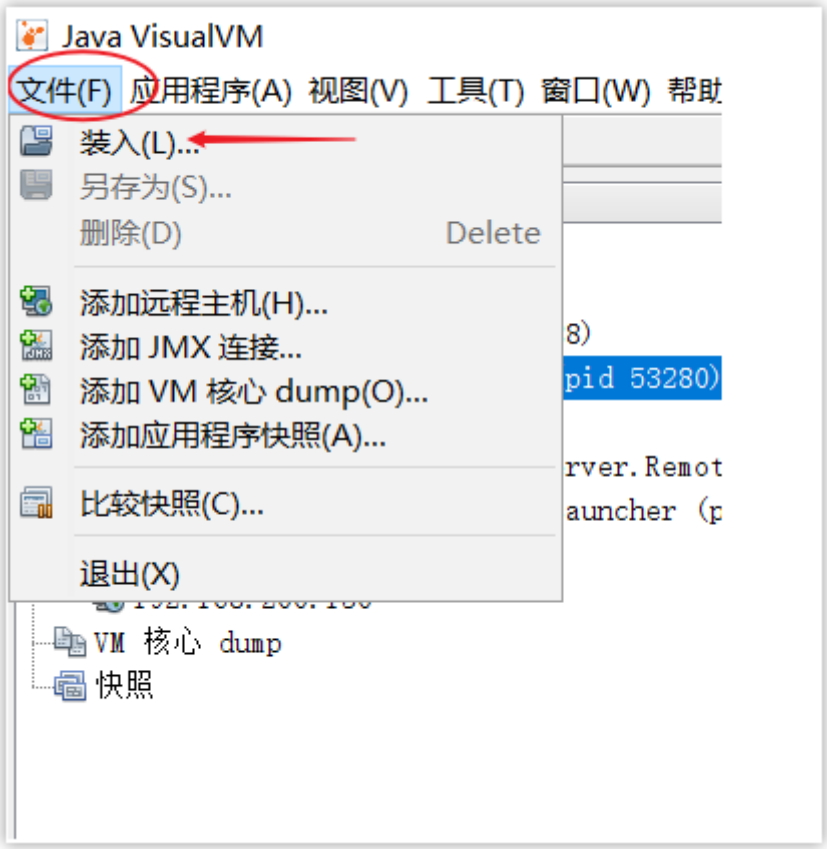
> 有的情况是内存溢出之后程序则会直接中断，而jmap只能打印在运行中的程序，所以建议通过参数的方式的生成dump文件，配置如下：
>
> -XX:+HeapDumpOnOutOfMemoryError
> -XX:HeapDumpPath=/home/app/dumps/　指定生成后文件的保存目录

2、通过工具， VisualVM（Ecplise MAT）去分析 dump文件

VisualVM可以加载离线的dump文件，如下图

文件-->装入--->选择dump文件即可查看堆快照信息

> 如果是linux系统中的程序，则需要把dump文件下载到本地（windows环境）下，打开VisualVM工具分析。VisualVM目前只支持在windows环境下运行可视化



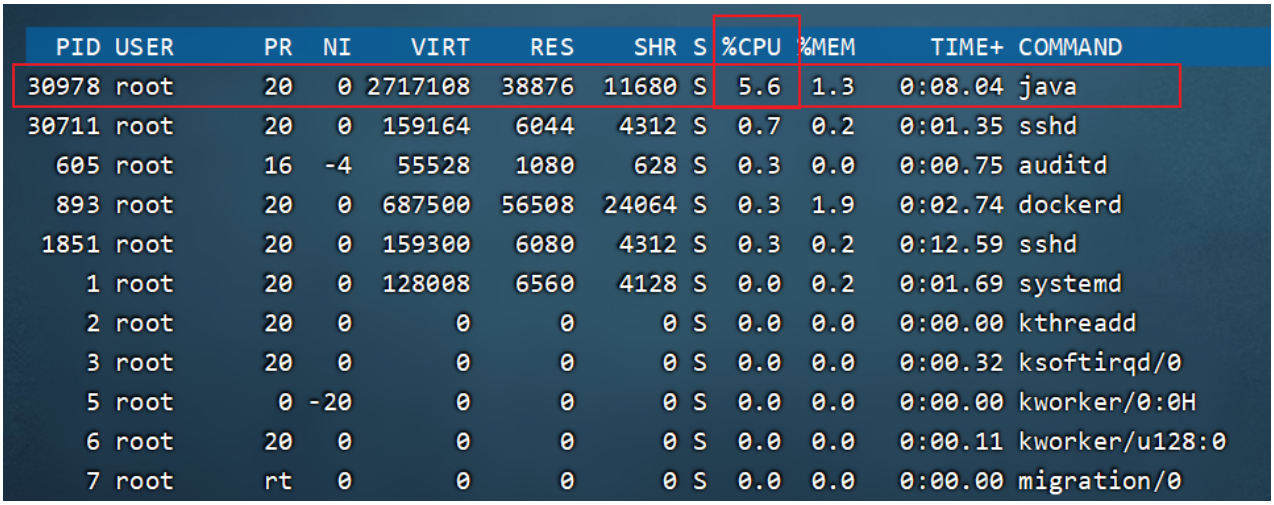
3、通过查看堆信息的情况，可以大概定位内存溢出是哪行代码出了问题

4、找到对应的代码，通过阅读上下文的情况，进行修复即可

4.5 CPU飙高排查方案与思路？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆☆

1.使用top命令查看占用cpu的情况



PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
30978	root	20	0	2717108	38876	11680	S	5.6	1.3	0:08.04	java
30711	root	20	0	159164	6044	4312	S	0.7	0.2	0:01.35	sshd
605	root	16	-4	55528	1080	628	S	0.3	0.0	0:00.75	auditd
893	root	20	0	687500	56508	24064	S	0.3	1.9	0:02.74	dockerd
1851	root	20	0	159300	6080	4312	S	0.3	0.2	0:12.59	sshd
1	root	20	0	128008	6560	4128	S	0.0	0.2	0:01.69	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:00.32	ksoftirqd/0
5	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:0H
6	root	20	0	0	0	0	S	0.0	0.0	0:00.11	kworker/u128:0
7	root	rt	0	0	0	0	S	0.0	0.0	0:00.00	migration/0

2.通过top命令查看后，可以查看是哪一个进程占用cpu较高，上图所示的进程为：30978

3.查看当前线程中的进程信息

```
ps H -eo pid,tid,%cpu | grep 40940
```

> pid 进行id
>
> tid 进程中的线程id
>
> % cpu使用率

```
[root@myhbase ~]# ps H -eo pid,tid,%cpu | grep 30978
30978 30978  0.0
30978 30979  3.5
30978 30980  0.0
30978 30981  0.0
30978 30982  0.0
30978 30983  0.0
30978 30984  0.0
30978 30985  0.0
30978 30986  0.0
30978 30987  0.0
```

4.通过上图分析，在进程30978中的线程30979占用cpu较高

注意：上述的线程id是一个十进制，我们需要把这个线程id转换为16进制才行，因为通常在日志中展示的都是16进制的线程id名称

转换方式：

在linux中执行命令

```
printf "%x\n" 30979
```

```
[root@myhbase ~]# printf "%x\n" 30979
7903
[root@myhbase ~]#
```

5.可以根据线程 id 找到有问题的线程，进一步定位到问题代码的源码行号

执行命令

```
jstack 30978    此处是进程id
```

```
"main" #1 prio=5 os_prio=0 tid=0x00007f805c009000 nid=0x7903 runnable [0x00007f8065972000]
   java.lang.Thread.State: RUNNABLE
        at java.io.FileOutputStream.writeBytes(Native Method)
        at java.io.FileOutputStream.write(FileOutputStream.java:326)
        at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:82)
        at java.io.BufferedOutputStream.flush(BufferedOutputStream.java:140)
        - locked <0x00000000e26bbec8> (a java.io.BufferedOutputStream)
        at java.io.PrintStream.write(PrintStream.java:482)
        - locked <0x00000000e26b42d0> (a java.io.PrintStream)
        at sun.nio.cs.StreamEncoder.writeBytes(StreamEncoder.java:221)
        at sun.nio.cs.StreamEncoder.implFlushBuffer(StreamEncoder.java:291)
        at sun.nio.cs.StreamEncoder.flushBuffer(StreamEncoder.java:104)
        - locked <0x00000000e26b4288> (a java.io.OutputStreamWriter)
        at java.io.OutputStreamWriter.flushBuffer(OutputStreamWriter.java:185)
        at java.io.PrintStream.newLine(PrintStream.java:546)
        - eliminated <0x00000000e26b42d0> (a java.io.PrintStream)
        at java.io.PrintStream.println(PrintStream.java:807)
        - locked <0x00000000e26b42d0> (a java.io.PrintStream)
        at MainTest.main(MainTest.java:5)
```

转换后的线程id

有可能出问题的代码行号

5.面试现场

5.1 JVM组成

面试官：JVM由那些部分组成，运行流程是什么？

候选人：

嗯，好的~~

在JVM中共有四大部分，分别是ClassLoader（类加载器）、Runtime Data Area（运行时数据区，内存分区）、Execution Engine（执行引擎）、Native Method Library（本地库接口）

它们的运行流程是：

第一，类加载器（ClassLoader）把Java代码转换为字节码

第二，运行时数据区（Runtime Data Area）把字节码加载到内存中，而字节码文件只是JVM的一套指令集规范，并不能直接交给底层系统去执行，而是有执行引擎运行

第三，执行引擎（Execution Engine）将字节码翻译为底层系统指令，再交由CPU执行去执行，此时需要调用其他语言的本地库接口（Native Method Library）来实现整个程序的功能。

面试官：好的，你能详细说一下 JVM 运行时数据区吗？

候选人：

嗯，好~

运行时数据区包含了堆、方法区、栈、本地方法栈、程序计数器这几部分，每个功能作用不一样。

- 堆解决的是对象实例存储的问题，垃圾回收器管理的主要区域。
- 方法区可以认为是堆的一部分，用于存储已被虚拟机加载的信息，常量、静态变量、即时编译器编译后的代码。
- 栈解决的是程序运行的问题，栈里面存的是栈帧，栈帧里面存的是局部变量表、操作数栈、动态链接、方法出口等信息。
- 本地方法栈与栈功能相同，本地方法栈执行的是本地方法，一个Java调用非Java代码的接口。
- 程序计数器（PC寄存器）程序计数器中存放的是当前线程所执行的字节码的行数。JVM工作时就是通过改变这个计数器的值来选取下一个需要执行的字节码指

令。

面试官：好的，你再详细介绍一下程序计数器的作用？

候选人:

嗯，是这样~~

java虚拟机对于多线程是通过线程轮流切换并且分配线程执行时间。在任何的一个时间点上，一个处理器只会处理执行一个线程，如果当前被执行的这个线程它所分配的执行时间用完了【挂起】。处理器会切换到另外的一个线程上来进行执行。并且这个线程的执行时间用完了，接着处理器就会又来执行被挂起的这个线程。这时候程序计数器就起到了关键作用，程序计数器在来回切换的线程中记录他上一次执行的行号，然后接着继续向下执行。

面试官：你能给我详细的介绍Java堆吗?

候选人:

好的~

Java中的堆术语线程共享的区域。主要用来保存**对象实例，数组**等，当堆中没有内存空间可分配给实例，也无法再扩展时，则抛出OutOfMemoryError异常。

在JAVA8中堆内会存在年轻代、老年代

1）Young区被划分为三部分，Eden区和两个大小严格相同的Survivor区，其中，Survivor区间中，某一时刻只有其中一个是被使用的，另外一个留做垃圾收集时复制对象用。在Eden区变满的时候， GC就会将存活的对象移到空闲的Survivor区间中，根据JVM的策略，在经过几次垃圾收集后，任然存活于Survivor的对象将被移动到Tenured区间。

2）Tenured区主要保存生命周期长的对象，一般是一些老的对象，当一些对象在Young复制转移一定的次数以后，对象就会被转移到Tenured区。

面试官：能不能解释一下方法区？

候选人:

好的~

与虚拟机栈类似。本地方法栈是为虚拟机**执行本地方法时提供服务的**。不需要进行GC。本地方法一般是由其他语言编写。

面试官：你听过直接内存吗？

候选人:

嗯~~

它又叫做**堆外内存，线程共享的区域**，在 Java 8 之前有个**永久代**的概念，实际上指的是 HotSpot 虚拟机上的永久代，它用永久代实现了 JVM 规范定义的方法区功能，**主要存储类的信息，常量，静态变量**，即时编译器编译后代码等，这部分由于是在堆中实现的，受 GC 的管理，不过由于永久代有 -XX:MaxPermSize 的上限，所以如果大量动态生成类（将类信息放入永久代），很容易造成 OOM，有人说可以把永久代设置得足够大，但很难确定一个合适的大小，受类数量，常量数量的多少影响很大。

所以在 Java 8 中就把方法区的实现移到了本地内存中的元空间中，这样方法区就不受 JVM 的控制了,也就不会进行 GC，也因此提升了性能。

面试官：什么是虚拟机栈

候选人:

虚拟机栈是描述的是方法执行时的内存模型,是线程私有的，生命周期与线程相同,每个方法被执行的同时会创建**栈桢**。保存执行方法时的**局部变量、动态连接信息、方法返回地址信息**等等。方法开始执行的时候会进栈，方法执行完会出栈【相当于清空了数据】，所以这块区域**不需要进行 GC**。

面试官：能说一下堆栈的区别是什么吗？

候选人:

嗯，好的，有这几个区别

第一，栈内存一般会用来存储局部变量和方法调用，但堆内存是用来存储Java对象和数组的的。堆会GC垃圾回收，而栈不会。

第二、栈内存是线程私有的，而堆内存是线程共有的。

第三、两者异常错误不同，但如果栈内存或者堆内存不足都会抛出异常。

栈空间不足：java.lang.StackOverFlowError。

堆空间不足：java.lang.OutOfMemoryError。

5.2 类加载器

面试官：什么是类加载器，类加载器有哪些?

候选人:

嗯，是这样的

JVM只会运行二进制文件，而类加载器（ClassLoader）的主要作用就是将**字节码文件加载到JVM中**，从而让Java程序能够启动起来。

常见的类加载器有4个

第一个是启动类加载器(BootStrap ClassLoader)：其是由C++编写实现。用于加载JAVA_HOME/jre/lib目录下的类库。

第二个是扩展类加载器(ExtClassLoader)：该类是ClassLoader的子类，主要加载JAVA_HOME/jre/lib/ext目录中的类库。

第三个是应用类加载器(AppClassLoader)：该类是ClassLoader的子类，主要用于加载classPath下的类，也就是加载开发者自己编写的Java类。

第四个是自定义类加载器：开发者自定义类继承ClassLoader，实现自定义类加载规则。

面试官：说一下类装载的执行过程？

候选人:

嗯，这个过程还是挺多的。

类从加载到虚拟机中开始，直到卸载为止，它的整个生命周期包括了：加载、验证、准备、解析、初始化、使用和卸载这7个阶段。其中，验证、准备和解析这三个部分统称为连接（linking）

1.加载：查找和导入class文件

2.验证：保证加载类的准确性

3.准备：为类变量分配内存并设置类变量初始值

4.解析：把类中的符号引用转换为直接引用

5.初始化：对类的静态变量，静态代码块执行初始化操作

6.使用：JVM 开始从入口方法开始执行用户的程序代码

7.卸载：当用户程序代码执行完毕后，JVM 便开始销毁创建的 Class 对象，最后负责运行的 JVM 也退出内存

面试官：什么是双亲委派模型？

候选人:

嗯，它是是这样的。

如果一个类加载器收到了类加载的请求，它首先不会自己尝试加载这个类，而是把这请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传说到顶层的启动类加载器中，只有当父类加载器返回自己无法完成这个加载请求（它的搜索返回中没有找到所需的类）时，子类加载器才会尝试自己去加载

面试官：JVM为什么采用双亲委派机制

候选人:

主要有两个原因。

第一、通过双亲委派机制可以避免某一个类被重复加载，当父类已经加载后则无需重复加载，保证唯一性。

第二、为了安全，保证类库API不会被修改

5.3 垃圾回收

面试官：简述Java垃圾回收机制？（GC是什么？为什么要GC）

候选人:

嗯，是这样~~

为了让程序员更专注于代码的实现，而不用过多的考虑内存释放的问题，所以，在Java语言中，有了自动的垃圾回收机制，也就是我们熟悉的GC(Garbage Collection)。

有了垃圾回收机制后，程序员只需要关心内存的申请即可，内存的释放由系统自动识别完成。

在进行垃圾回收时，不同的对象引用类型，GC会采用不同的回收时机

面试官：强引用、软引用、弱引用、虚引用的区别？

候选人:

嗯嗯~

强引用最为普通的引用方式，表示一个对象处于**有用且必须**的状态，如果一个对象具有强引用，则GC并不会回收它。即便堆中内存不足了，宁可出现OOM，也不会对其进行回收

软引用表示一个对象处于**有用且非必须**状态，如果一个对象处于软引用，在内存空间足够的情况下，GC机制并不会回收它，而在内存空间不足时，则会在OOM异常出现之间对其进行回收。但值得注意的是，因为GC线程优先级较低，软引用并不会立即被回收。

弱引用表示一个对象处于**可能有用且非必须**的状态。在GC线程扫描内存区域时，一旦发现弱引用，就会回收到弱引用相关联的对象。对于弱引用的回收，无关内存区域是否足够，一旦发现则会被回收。同样的，因为GC线程优先级较低，所以弱引用也并不是会被立刻回收。

虚引用表示一个对象处于**无用**的状态。在任何时候都有可能被垃圾回收。虚引用的使用必须和引用队列Reference Queue联合使用

面试官：对象什么时候可以被垃圾器回收

候选人:

思考一会~~

如果一个或多个对象没有任何的引用指向它了，那么这个对象现在就是垃圾，如果定位了垃圾，则有可能会被垃圾回收器回收。

如果要定位什么是垃圾，有两种方式来确定，第一个是引用计数法，第二个是可达性分析算法

通常都使用可达性分析算法来确定是不是垃圾

面试官： JVM 垃圾回收算法有哪些？

候选人:

我记得一共有四种，分别是标记清除算法、复制算法、标记整理算法、分代回收

面试官： 你能详细聊一下分代回收吗？

候选人:

关于分代回收是这样的

在java8时，堆被分为了两份：新生代和老年代，它们默认空间占用比例是1:2

对于新生代，内部又被分为了三个区域。Eden区，S0区，S1区默认空间占用比例是8:1:1

具体的工作机制是有些情况：

1）当创建一个对象的时候，那么这个对象会被分配在新生代的Eden区。当Eden区要满了时候，触发YoungGC。

2）当进行YoungGC后，此时在Eden区存活的对象被移动到S0区，并且**当前对象的年龄会加1**，清空Eden区。

3）当再一次触发YoungGC的时候，会把Eden区中存活下来的对象和S0中的对象，移动到S1区中，这些对象的年龄会加1，清空Eden区和S0区。

4）当再一次触发YoungGC的时候，会把Eden区中存活下来的对象和S1中的对象，移动到S0区中，这些对象的年龄会加1，清空Eden区和S1区。

5）对象的年龄达到了某一个限定的值（**默认15岁** ），那么这个对象就会进入到老年代中。

当然也有特殊情况，如果进入Eden区的是一个大对象，在触发YoungGC的时候，会直接存放到老年代

当老年代满了之后，**触发FullGC**。**FullGC同时回收新生代和老年代**，当前只会存在一个FullGC的线程进行执行，其他的线程全部会被挂起。 我们在程序中要尽量避免FullGC的出现。

面试官：讲一下新生代、老年代、永久代的区别？

候选人:

嗯！是这样的，简单说就是

新生代主要用来存放新生的对象。

老年代主要存放应用中生命周期长的内存对象。

永久代指的是永久保存区域。主要存放Class和Meta（元数据）的信息。在Java8中，永久代已经被移除，取而代之的是一个称之为“元数据区”（**元空间**）的区域。元空间和永久代类似，不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存的限制。

面试官：说一下 JVM 有哪些垃圾回收器？

候选人:

在jvm中，实现了多种垃圾收集器，包括：串行垃圾收集器、并行垃圾收集器（JDK8默认）、CMS（并发）垃圾收集器、G1垃圾收集器（JDK9默认）

面试官：Minor GC、Major GC、Full GC是什么

候选人:

嗯，其实它们指的是不同代之间的垃圾回收

Minor GC 发生在新生代的垃圾回收，暂停时间短

Major GC 老年代区域的垃圾回收，老年代空间不足时，会先尝试触发Minor GC。Minor GC之后空间还不足，则会触发Major GC，Major GC速度比较慢，暂停时间长

Full GC 新生代 + 老年代完整垃圾回收，暂停时间长，**应尽力避免**

5.4 JVM实践（调优）

面试官：JVM 调优的参数可以在哪里设置参数值？

候选人:

我们当时的项目是springboot项目，可以在项目启动的时候，java -jar中加入参数就行了

面试官：用的 JVM 调优的参数都有哪些？

候选人:

嗯，这些参数是比较多的

我记得当时我们设置过堆的大小，像-Xms和-Xmx

还有就是可以设置年轻代中Eden区和两个Survivor区的大小比例

还有就是可以设置使用哪种垃圾回收器等等。具体的指令还真记不太清楚。

面试官：嗯，好的，你们平时调试 JVM都用了哪些工具呢？

候选人:

嗯，我们一般都是使用jdk自带的一些工具，比如

jps 输出JVM中运行的进程状态信息

jstack查看java进程内**线程的堆栈**信息。

jmap 用于生成堆转存快照

jstat用于JVM统计监测工具

还有一些可视化工具，像jconsole和VisualVM等

面试官：假如项目中产生了java内存泄露，你说一下你的排查思路？

候选人:

嗯，这个我在之前项目排查过

第一呢可以通过jmap指定打印他的内存快照 dump文件，不过有的情况打印不了，我们会设置vm参数让程序自动生成dump文件

第二，可以通过工具去分析 dump文件，jdk自带的VisualVM就可以分析

第三，通过查看堆信息的情况，可以大概定位内存溢出是哪行代码出了问题

第四，找到对应的代码，通过阅读上下文的情况，进行修复即可

面试官：好的，那现在再来说一种情况，就是说服务器CPU持续飙高，你的排查方案与思路？

候选人:

嗯，我思考一下~~

可以这么做~~

第一可以使用使用top命令查看占用cpu的情况

第二通过top命令查看后，可以查看是哪一个进程占用cpu较高，记录这个进程id

第三可以通过ps 查看当前进程中的线程信息，看看哪个线程的cpu占用较高

第四可以jstack命令打印进行的id，找到这个线程，就可以进一步定位问题代码的行号