

# JVM相关面试题



黑马程序员  
[www.itheima.com](http://www.itheima.com)

传智教育旗下  
高端IT教育品牌

## 为什么要学习它

应对面试

中高级程序员必备

深入理解Java



升级

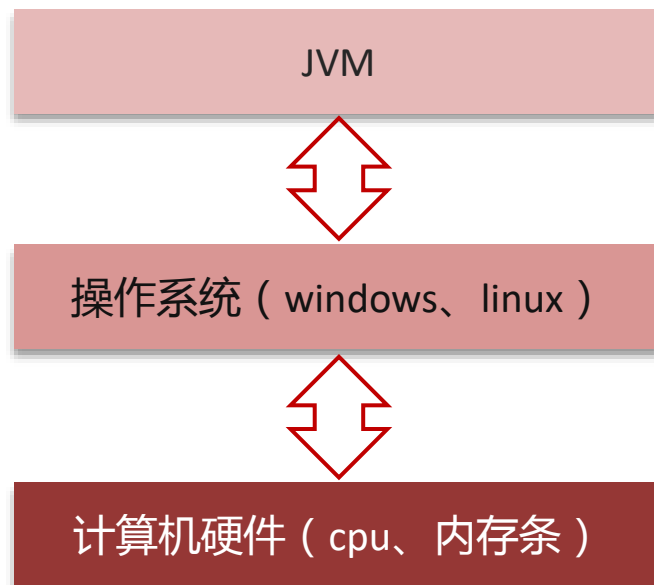


## JVM是什么

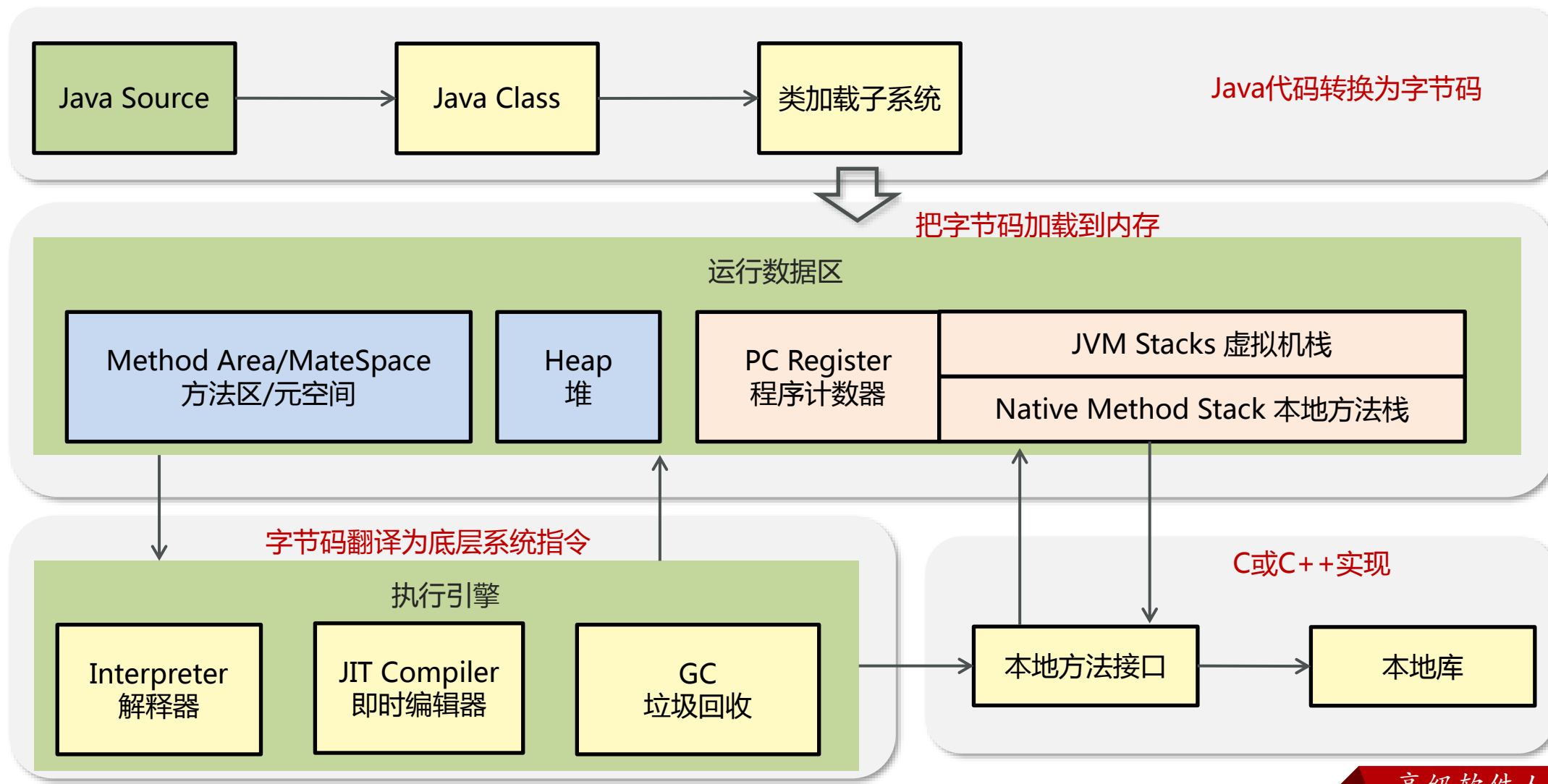
Java Virtual Machine Java程序的运行环境（java二进制字节码的运行环境）

好处：

- 一次编写，到处运行
- 自动内存管理，垃圾回收机制

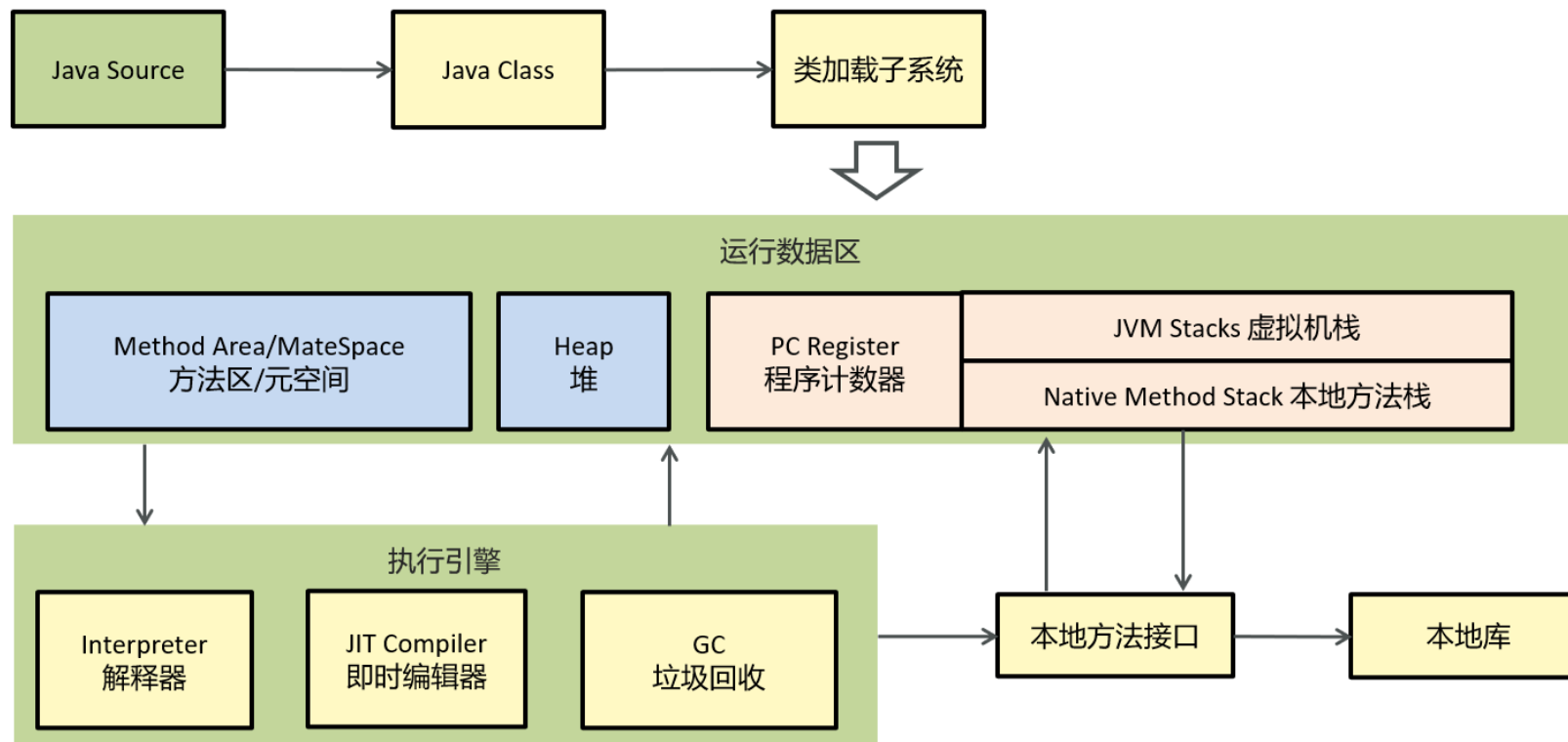


## JVM由哪些部分组成，运行流程是什么？



## 学习什么

- JVM组成
- 类加载器
- 垃圾回收
- JVM实践



## JVM组成

什么是程序计数器  
你能给我详细的介绍下堆吗？  
能不能介绍一下方法区  
你听过直接内存吗  
什么是虚拟机栈  
垃圾回收是否涉及栈内存？  
栈内存分配越大越好吗？  
方法内的局部变量是否线程安全？  
什么情况下会导致栈内存溢出？  
堆栈的区别是什么

## 类加载器

什么是类加载器，类加载器有哪些  
什么是双亲委派模型？  
JVM为什么采用双亲委派机制？  
说一下类装载的执行过程

## 垃圾回收

强引用、软引用、弱引用、虚对象  
什么时候可以被垃圾器回收  
JVM 垃圾回收算法有哪些？  
说一下JVM中的分代回收  
说一下JVM有哪些垃圾回收器？  
详细聊一下G1垃圾回收器

## JVM实践

JVM 调优的参数可以在哪里设置  
用的 JVM 调优的参数都有哪些？  
说一下 JVM 调优的工具？  
Java内存泄露的排查思路？  
CPU飙高排查方案与思路？

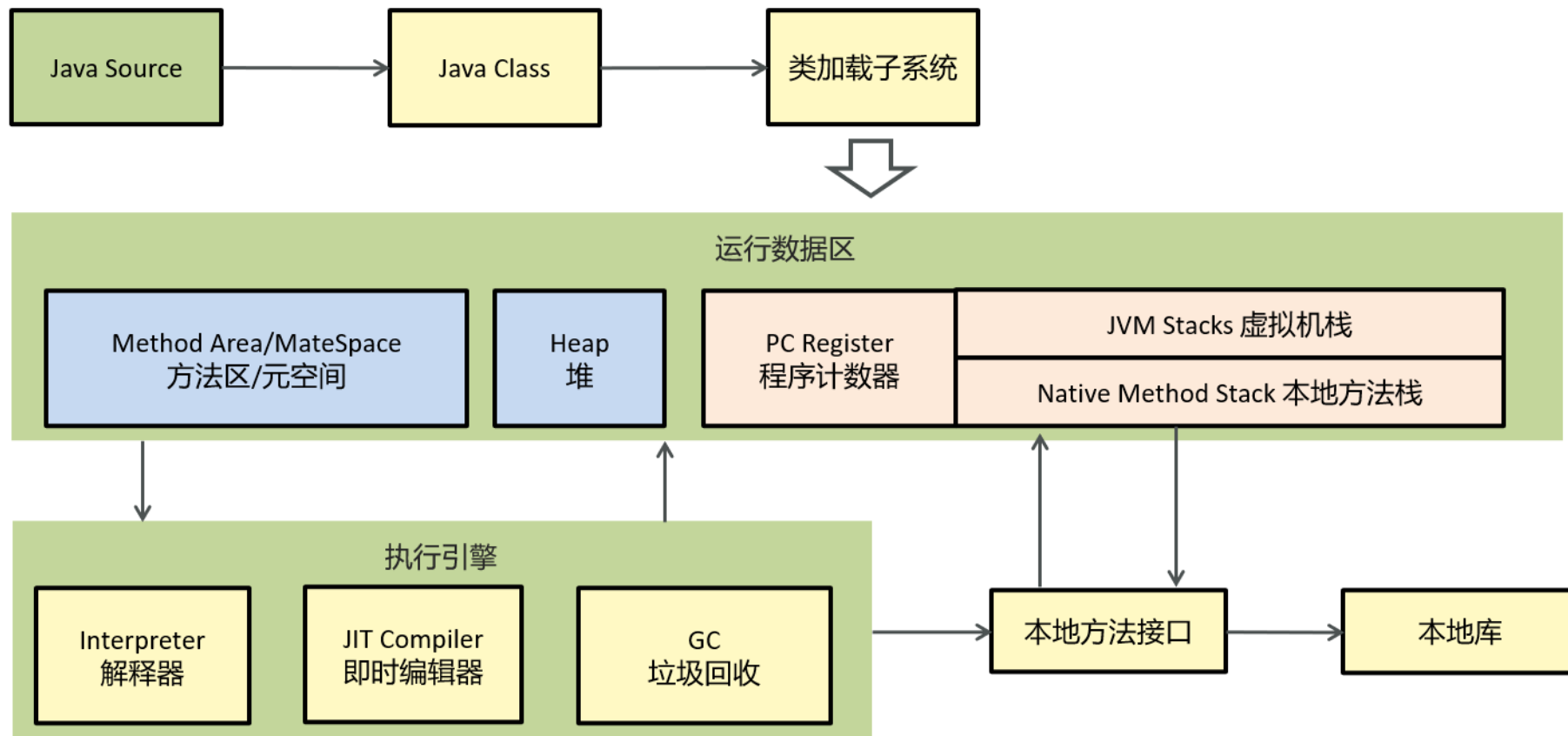
# 什么是程序计数器？

难易程度： ★★★★★

出现频率： ★★★★★

## 什么是程序计数器？

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。






## 什么是程序计数器？


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

线程1



```
stack=2, locals=1, args_size=0
 0: new          #2
 3: dup
 4: invokespecial #3
 7: astore_0
 8: aload_0
 9: iconst_1
10: invokevirtual #13
13: pop
14: aload_0
15: iconst_2
16: invokevirtual #13
19: pop
20: aload_0
21: iconst_3
22: invokevirtual #13
25: pop
```

线程2



```
stack=2, locals=1, args_size=0
 0: new          #2
 3: dup
 4: invokespecial #3
 7: astore_0
 8: aload_0
 9: iconst_1
10: invokevirtual #13
13: pop
14: aload_0
15: iconst_2
16: invokevirtual #13
19: pop
20: aload_0
21: iconst_3
22: invokevirtual #13
25: pop
```



# 总结

## 什么是程序计数器？

线程私有的，每个线程一份，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。

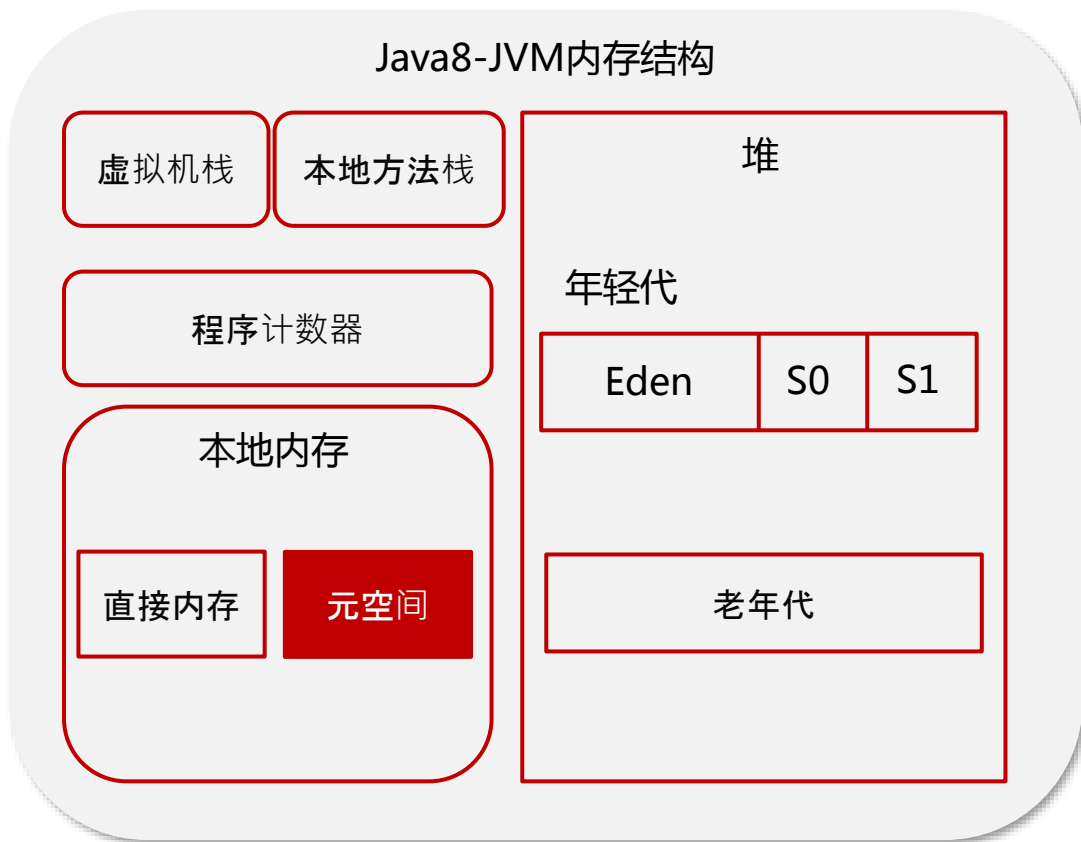
# 你能给我详细的介绍Java堆吗?

难易程度： ★★★★★

出现频率： ★★★★★

## 你能给我详细的介绍Java堆吗?

**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。



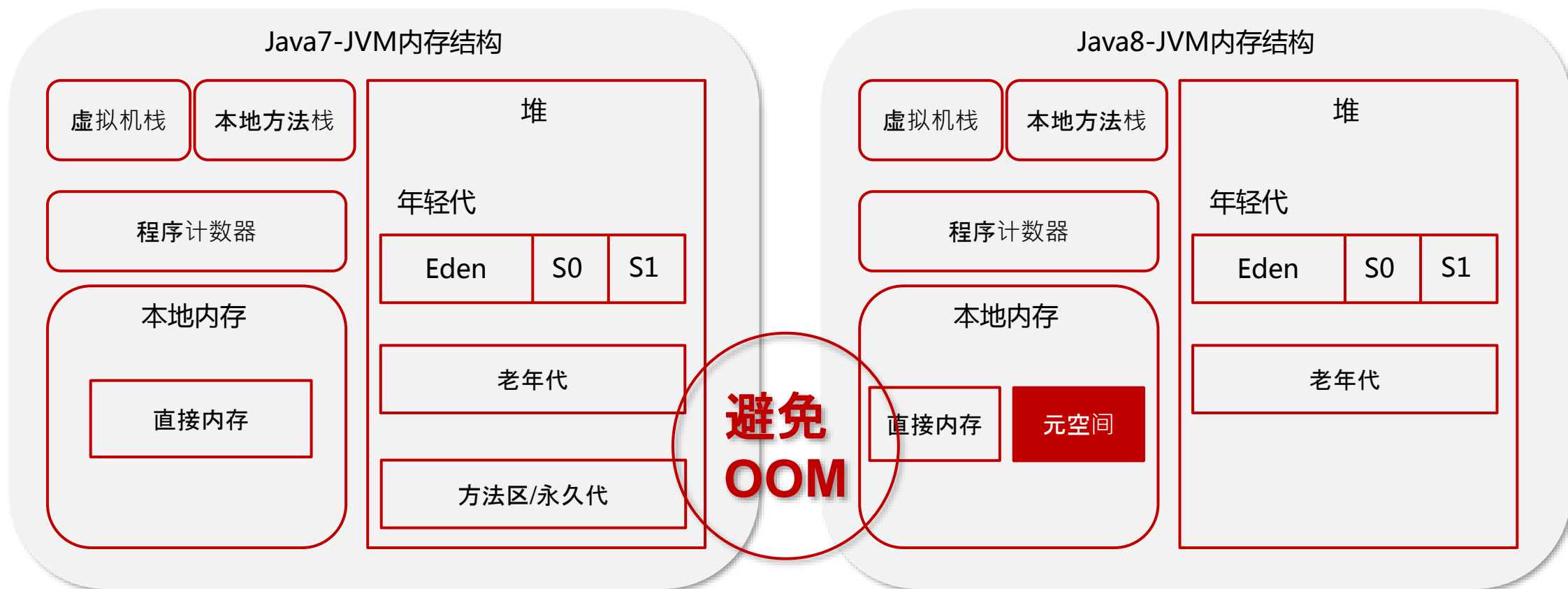
**年轻代**被划分为三部分，Eden区和两个大小严格相同的Survivor区，根据JVM的策略，在经过几次垃圾收集后，任然存活于Survivor的对象将被移动到老年代区间。

**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

## 你能给我详细的介绍Java堆吗？

**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。



# 总结

你能给我详细的介绍Java堆吗?

- **线程共享的区域**：主要用来保存**对象实例**，**数组**等，内存不够则抛出OutOfMemoryError异常。
- **组成**：**年轻代**+**老年代**
  - **年轻代**被划分为三部分，Eden区和两个大小严格相同的Survivor区
  - **老年代**主要保存生命周期长的对象，一般是一些老的对象
- **Jdk1.7和1.8的区别**
  - 1.7中有有一个永久代，存储的是类信息、静态变量、常量、编译后的代码
  - 1.8移除了永久代，把数据存储到了本地内存的元空间中，防止内存溢出

# 什么是虚拟机栈

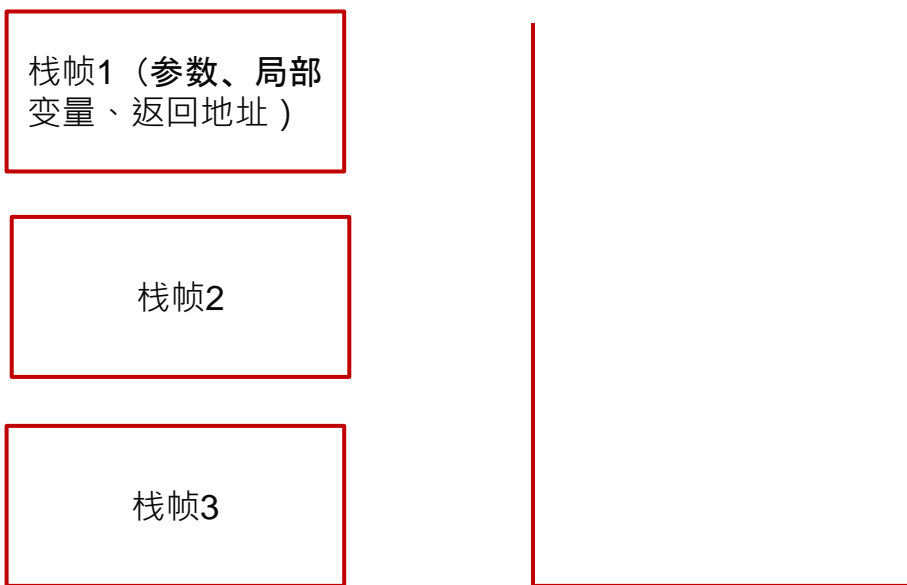
难易程度： ★★★★★

出现频率： ★★★★★

## 什么是虚拟机栈

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





## 什么是虚拟机栈

### 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

栈帧1（参数、局部  
变量、返回地址）

栈帧2

栈帧3

## 什么是虚拟机栈

### 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

```
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

```
public static void m4(){  
    m4();  
}
```

java.lang.**StackOverflowError**



# 总结

## 1.什么是虚拟机栈

- 每个线程运行时所需要的内存，称为虚拟机栈
- 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

## 2.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

## 3.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

## 4.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

## 5.什么情况下会导致栈内存溢出？

- 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



# 总结

## 6.堆栈的区别是什么？

- 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。堆会GC垃圾回收，而栈不会。
- 栈内存是线程私有的，而堆内存是线程共有的。
- 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`。

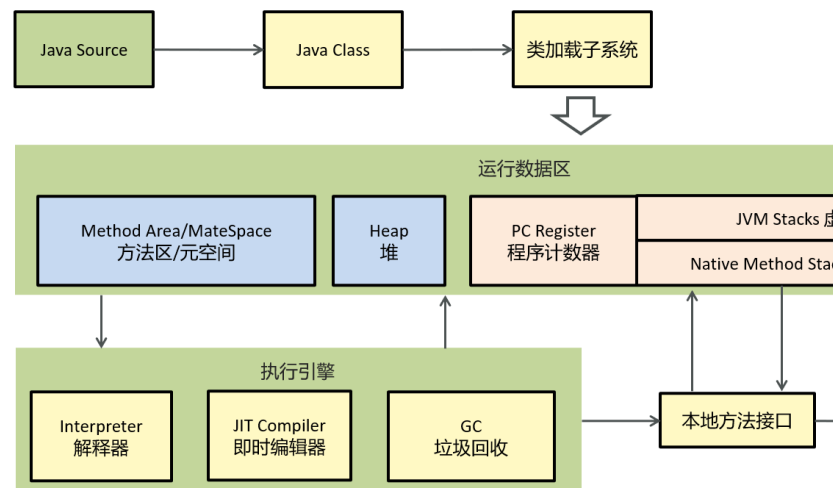
堆空间不足：`java.lang.OutOfMemoryError`。

# 能不能解释一下方法区？

难易程度：

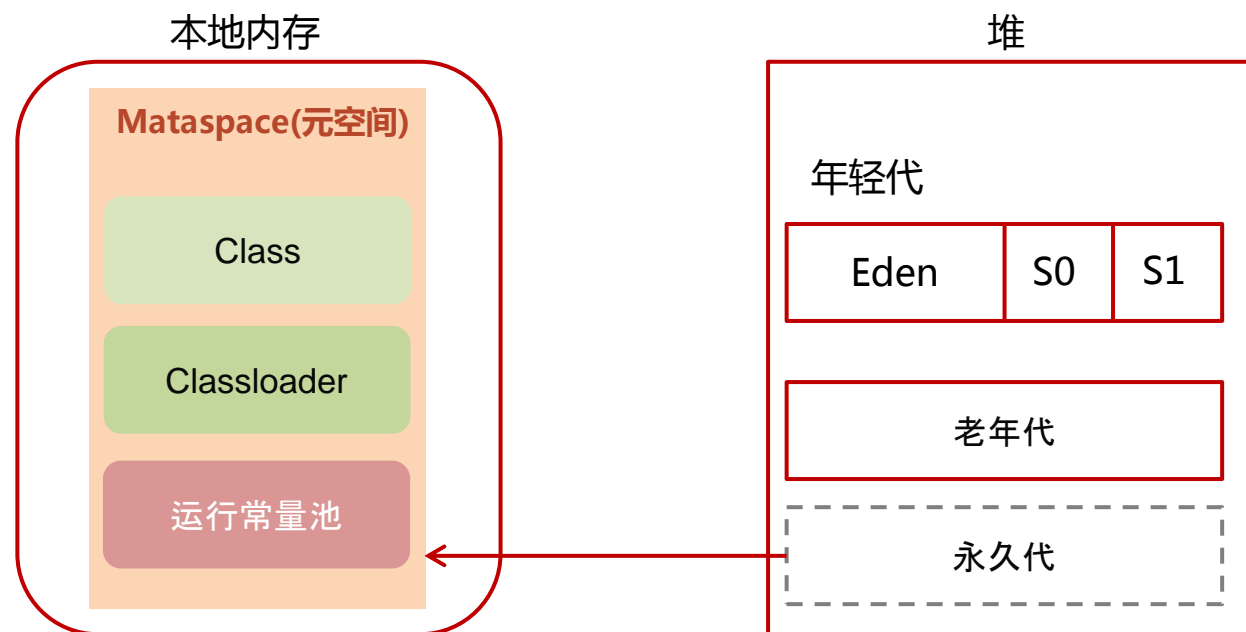


出现频率：



## 能不能解释一下方法区？

- 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



## 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

```
javap -v Application.class
```

查看字节码结构（类的基本信息、常量池、方法定义）

```
public static void main(java.lang.String[]);  
descriptor: ([Ljava/lang/String;)V  
flags: ACC_PUBLIC, ACC_STATIC  
Code:  
  stack=2, locals=1, args_size=1  
    0: getstatic    #2  
    3: ldc          #3  
    5: invokevirtual #4  
    8: return
```

方法的机器指令

查表翻译

```
Constant pool:  
  #1 = Methodref      #6.#20  
  #2 = Fieldref        #21.#22  
  #3 = String           #23  
  #4 = Methodref      #24.#25  
  #5 = Class            #26  
  #6 = Class            #27  
  #7 = Utf8             <init>  
  #8 = Utf8             ()V  
  #9 = Utf8             Code
```

常量池



## 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

```
public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
     0: getstatic    #2
     3: ldc          #3
     5: invokevirtual #4
     8: return
```

方法的机器指令

查表翻译

Constant pool:

#1	= Methodref	#6.#20
#2	= Fieldref	#21.#22
#3	= String	#23
#4	= Methodref	#24.#25
#5	= Class	#26
#6	= Class	#27
#7	= Utf8	<init>
#8	= Utf8	()V
#9	= Utf8	Code

符号引用

常量池



# 总结

## 1.能不能解释一下方法区？

- 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

## 2.介绍一下运行时常量池

- 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
- 当类被加载，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

# 你听过直接内存吗？

难易程度： ★★★★★

出现频率： ★★★★★

## 你听过直接内存吗？

直接内存：并不属于JVM中的内存结构，不由JVM进行管理。是虚拟机的系统内存，常见于 NIO 操作时，用于数据缓冲区，它分配回收成本较高，但读写性能高

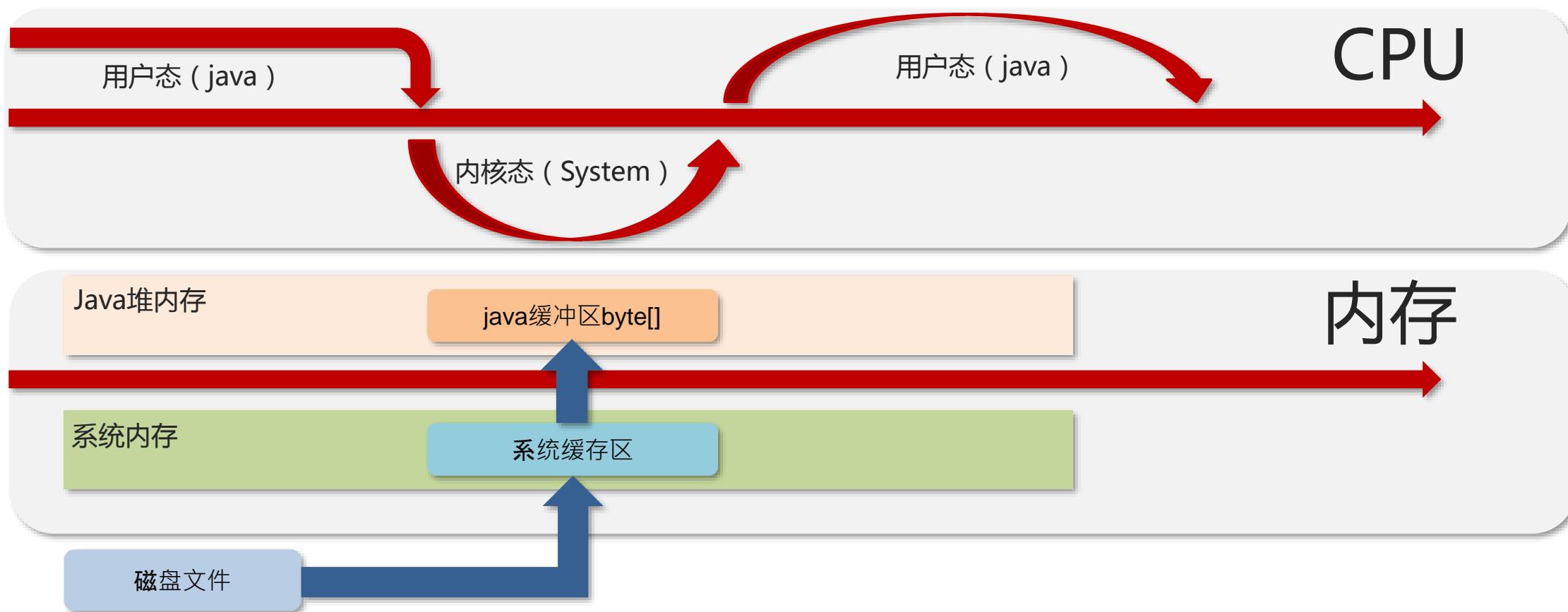
举例

Java代码完成文件拷贝



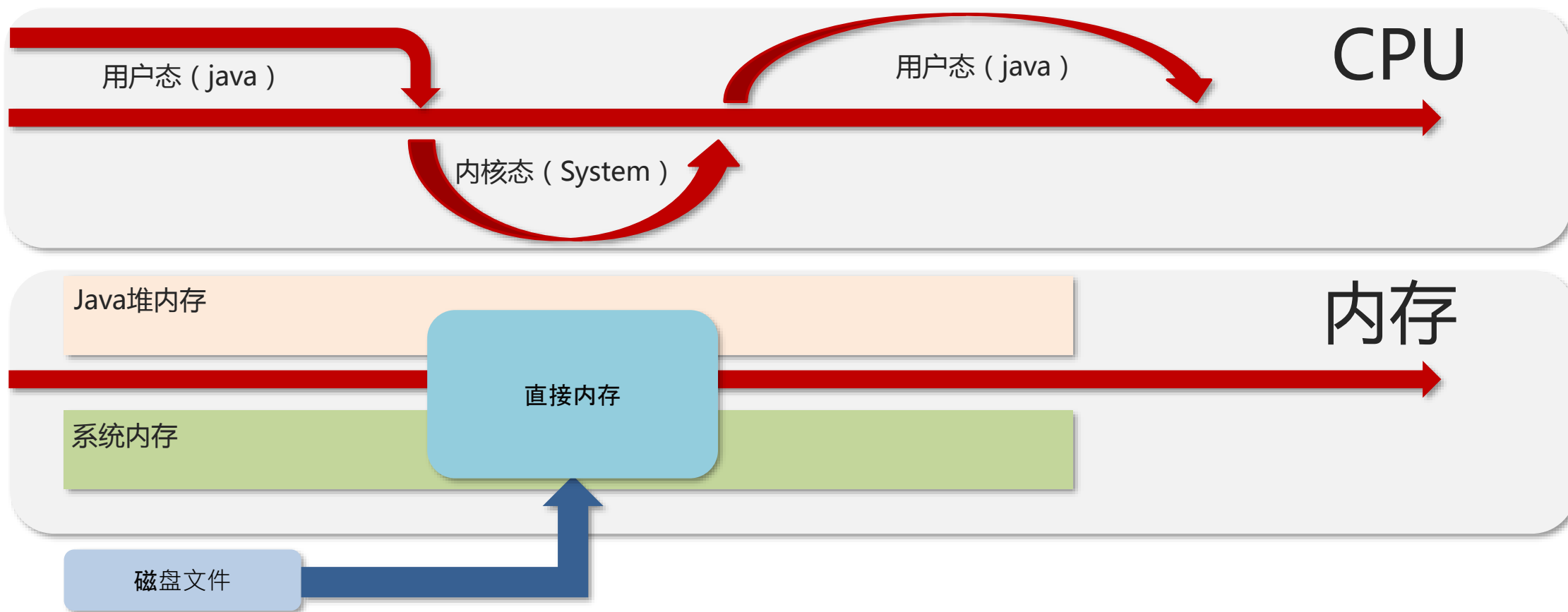
## 你听过直接内存吗？

常规IO的数据拷贝流程



## 你听过直接内存吗？

### NIO数据拷贝流程





# 总结

## 你听过直接内存吗？

- 并不属于JVM中的内存结构，不由JVM进行管理。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

## JVM组成

完成

什么是程序计数器

你能给我详细的介绍下堆吗？

能不能介绍一下方法区

你听过直接内存吗

什么是虚拟机栈

垃圾回收是否涉及栈内存？

栈内存分配越大越好吗？

方法内的局部变量是否线程安全？

什么情况下会导致栈内存溢出？

堆栈的区别是什么

## 类加载器

什么是类加载器，类加载器有哪些

什么是双亲委派模型？

JVM为什么采用双亲委派机制？

说一下类装载的执行过程

## 垃圾回收

强引用、软引用、弱引用、虚对象

什么时候可以被垃圾器回收

JVM 垃圾回收算法有哪些？

说一下JVM中的分代回收

说一下JVM有哪些垃圾回收器？

详细聊一下G1垃圾回收器

## JVM实践

JVM 调优的参数可以在哪里设置

用的 JVM 调优的参数都有哪些？

说一下 JVM 调优的工具？

Java内存泄露的排查思路？

CPU飙高排查方案与思路？



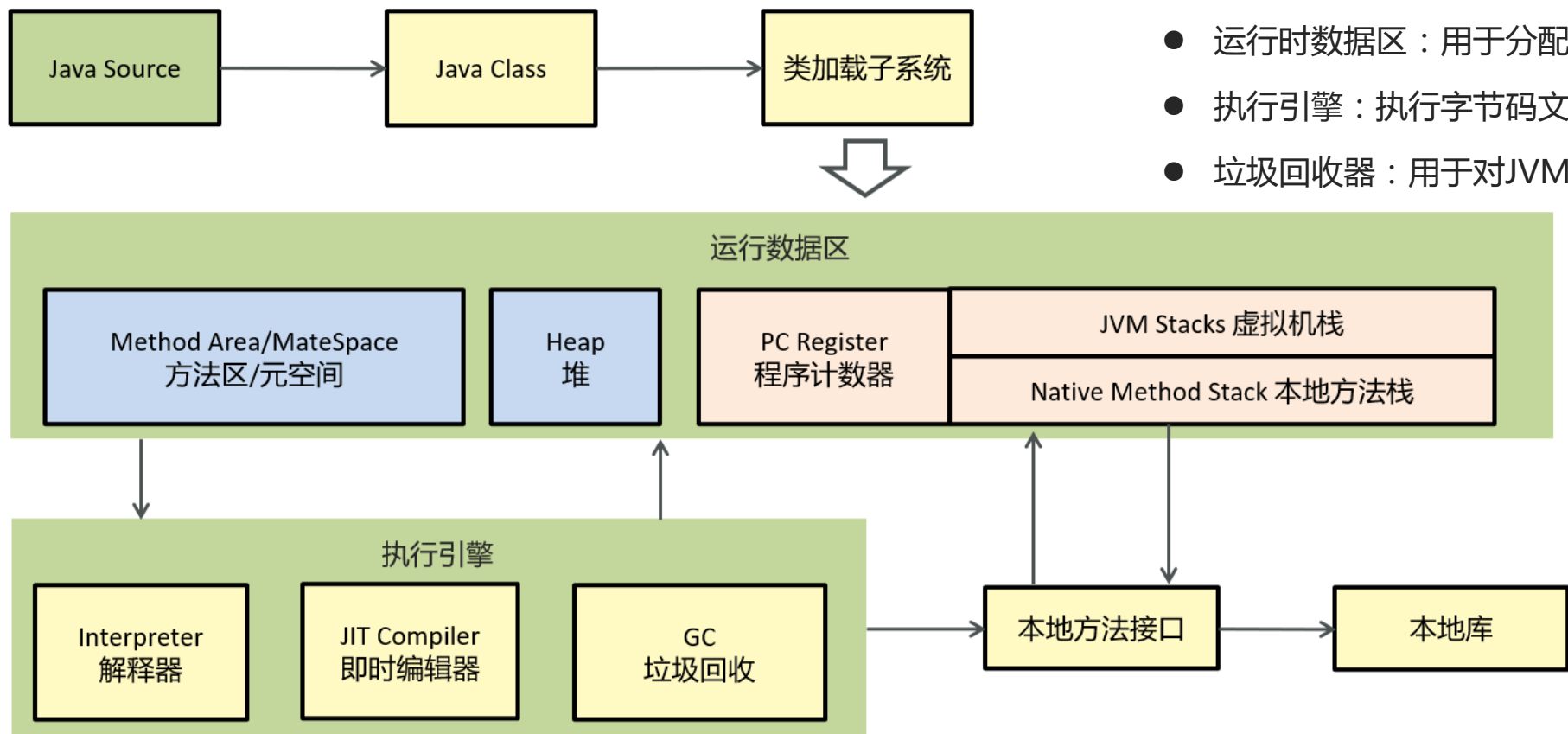
# 什么是类加载器，类加载器有哪些

难易程度： ★★★★★

出现频率： ★★★★★

## 什么是类加载器，类加载器有哪些

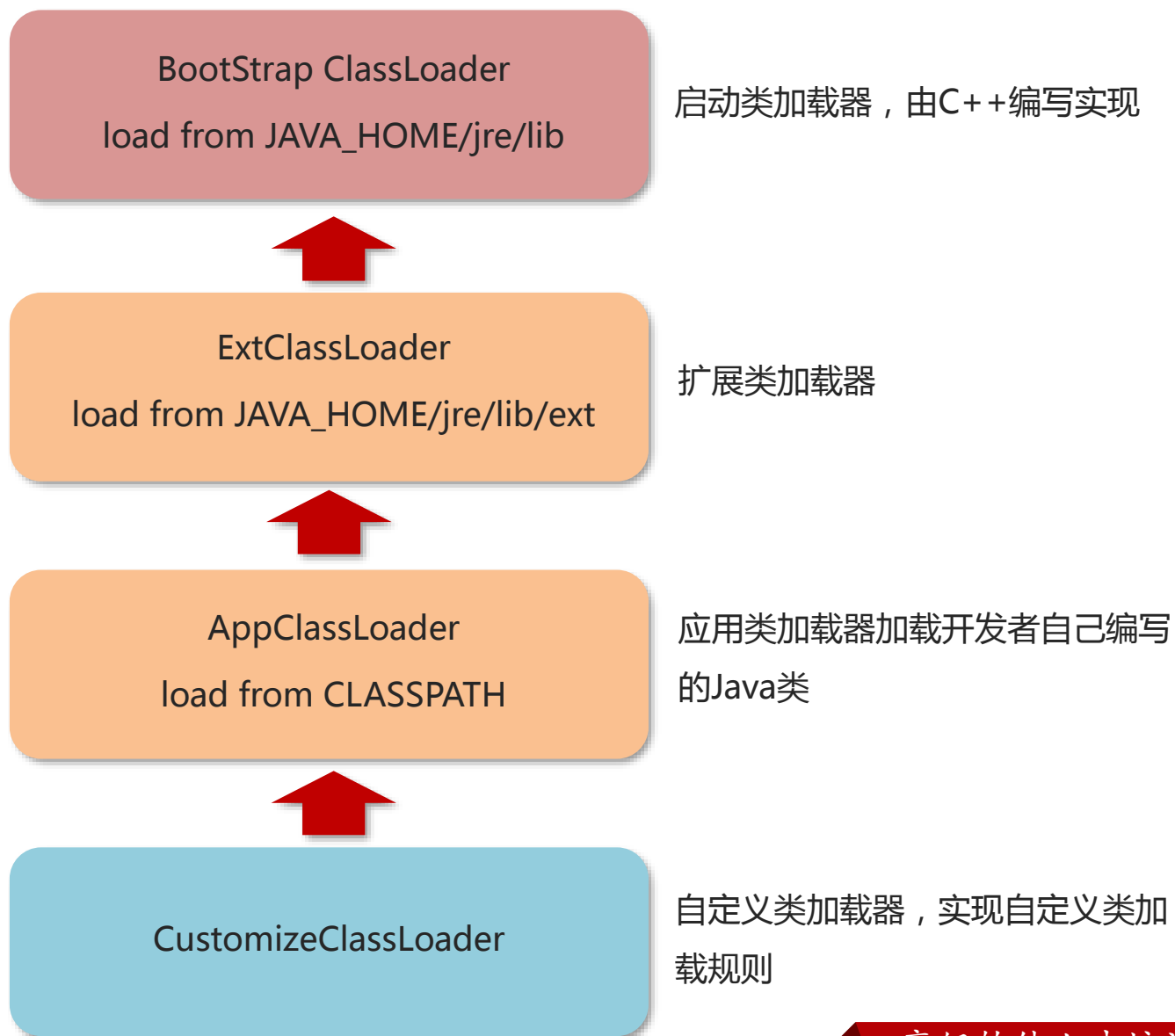
- 类加载器：用于装载字节码文件(.class文件)
- 运行时数据区：用于分配存储空间
- 执行引擎：执行字节码文件或本地方法
- 垃圾回收器：用于对JVM中的垃圾内容进行回收



## 什么是类加载器，类加载器有哪些

### 类加载器

JVM只会运行二进制文件，类加载器的作用就是将**字节码文件加载到JVM中**，从而让Java程序能够启动起来。





# 总结

## 1. 什么是类加载器

JVM只会运行二进制文件，类加载器的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。

## 2. 类加载器有哪些

- 启动类加载器(BootStrap ClassLoader):加载JAVA\_HOME/jre/lib目录下的库
- 扩展类加载器(ExtClassLoader):主要加载JAVA\_HOME/jre/lib/ext目录中的类
- 应用类加载器(AppClassLoader):用于加载classPath下的类
- 自定义类加载器(CustomizeClassLoader):自定义类继承ClassLoader，实现自定义类加载规则。

# 什么是双亲委派模型？

难易程度： ★★★★★

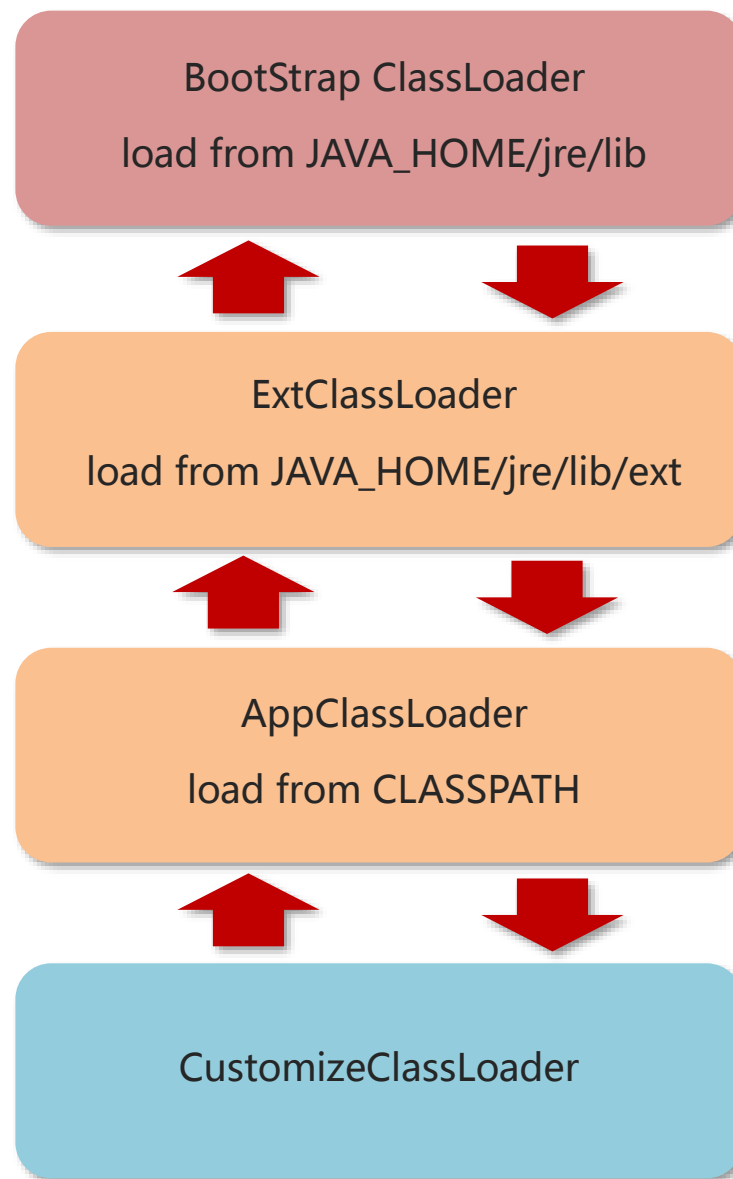
出现频率： ★★★★★

## 什么是双亲委派模型？

加载某一个类，先委托上一级的加载器进行加载，如果上级加载器也有上级，则会继续向上委托，如果该类委托上级没有被加载，子加载器尝试加载该类

Student类

String类



## JVM为什么采用双亲委派机制？

- (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

```
package java.lang;  
  
public class String {  
  
    public static void main(String[] args) {  
        System.out.println("demo info");  
    }  
}
```

由于是双亲委派的机制，java.lang.String的在启动类加载器得到加载，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。

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

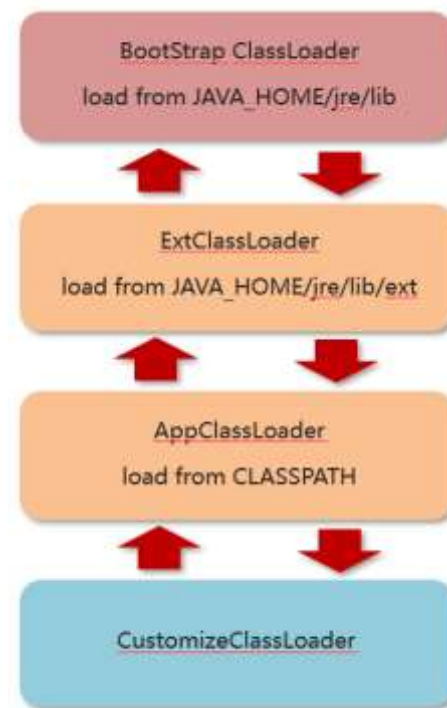
# 总结

## 1. 什么是双亲委派模型？

加载某一个类，先委托上一级的加载器进行加载，如果上级加载器也有上级，则会继续向上委托，如果该类委托上级没有被加载，子加载器尝试加载该类

## 2. JVM为什么采用双亲委派机制？

- 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。
- 为了安全，保证类库API不会被修改





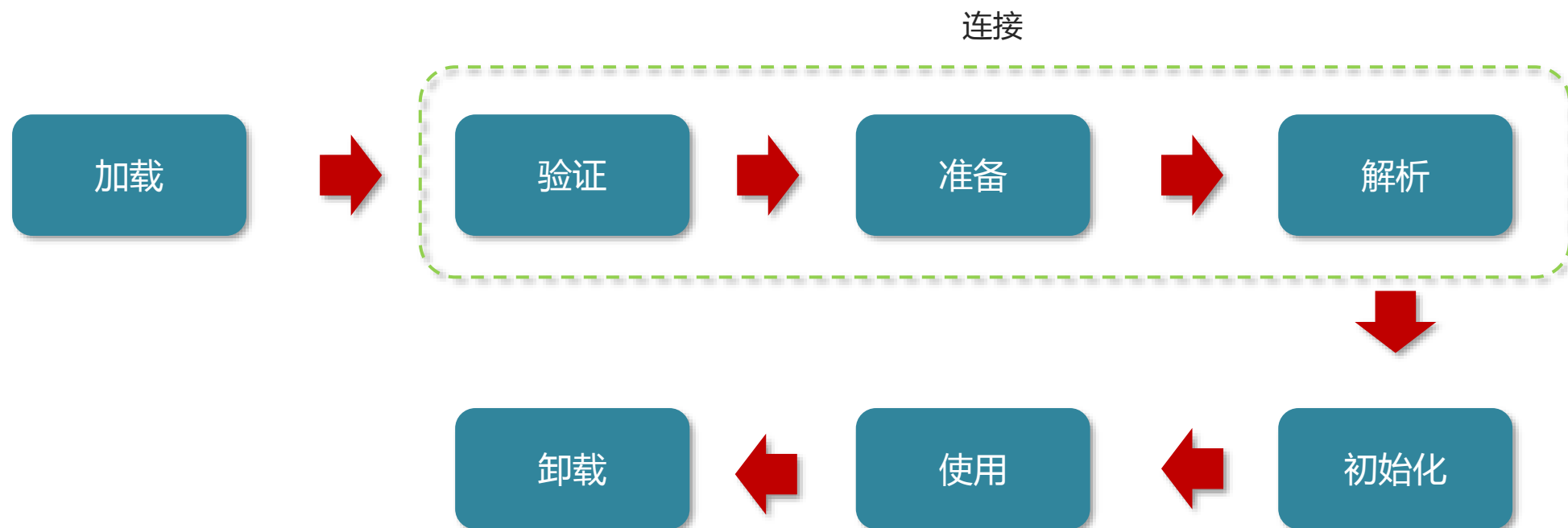
# 说一下类装载的执行过程？

难易程度： ★★★★★

出现频率： ★★★★★

## 说一下类装载的执行过程？

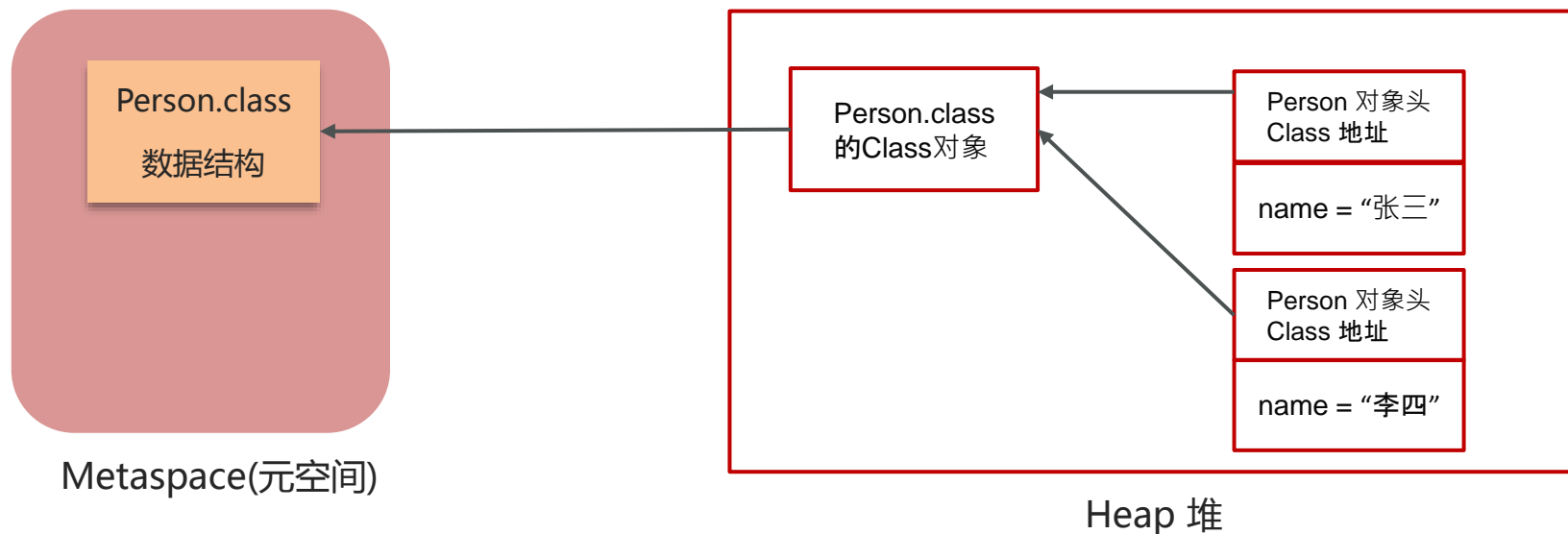
类从加载到虚拟机中开始，直到卸载为止，它的整个生命周期包括了：加载、验证、准备、解析、初始化、使用 and 卸载这7个阶段。其中，验证、准备和解析这三个部分统称为连接（linking）



## 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



## 验证



### 验证类是否符合 JVM 规范，安全性检查

(1)文件格式验证

(2)元数据验证

(3)字节码验证

(4)符号引用验证

格式检查，如：文件格式是否错误、语法是否错误、字节码是否合规

Class文件在其常量池会通过字符串记录自己将要使用的其他类或者方法，检查它们是否存在

Constant pool:

#1 = Methodref	#6.#20
#2 = Fieldref	#21.#22
#3 = String	#23
#4 = Methodref	#24.#25
#5 = Class	#26
#6 = Class	#27
#7 = Utf8	<init>
#8 = Utf8	()V
#9 = Utf8	Code

## 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();  
  
}
```

## 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。

```
public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
     0: getstatic    #2
     3: ldc          #3
     5: invokevirtual #4
     8: return
```

方法的机器指令

```
#24 = Class
#25 = NameAndType
#26 = Utf8
#27 = Utf8
#28 = Utf8
#29 = Utf8
#30 = Utf8
#31 = Utf8
#32 = Utf8
#33 = Utf8
```

Constant pool:

```
#1 = Methodref      #6.#20
#2 = Fieldref       #21.#22
#31                // java/io/PrintStream
#32:#33            // println:(Ljava/lang/String;)V
com/heimajvm/Application
java/lang/Object
java/lang/System
out
Ljava/io/PrintStream;
java/io/PrintStream
println
(Ljava/lang/String;)V
```

## 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。

## 初始化



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





## 总结

说一下类装载的执行过程？

- 加载:查找和导入class文件
- 验证:保证加载类的准确性
- 准备:为类变量分配内存并设置类变量初始值
- 解析:把类中的符号引用转换为直接引用
- 初始化:对类的静态变量，静态代码块执行初始化操作
- 使用:JVM 开始从入口方法开始执行用户的程序代码
- 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。

## JVM组成

完成

什么是程序计数器

你能给我详细的介绍下堆吗？

能不能介绍一下方法区

你听过直接内存吗

什么是虚拟机栈

垃圾回收是否涉及栈内存？

栈内存分配越大越好吗？

方法内的局部变量是否线程安全？

什么情况下会导致栈内存溢出？

堆栈的区别是什么

## 类加载器

完成

什么是类加载器，类加载器有哪些

什么是双亲委派模型？

JVM为什么采用双亲委派机制？

说一下类装载的执行过程

## 垃圾回收

对象什么时候可以被垃圾器回收

JVM 垃圾回收算法有哪些？

说一下JVM中的分代回收

说一下JVM有哪些垃圾回收器？

详细聊一下G1垃圾回收器

强引用、软引用、弱引用、虚对象

## JVM实践

JVM 调优的参数可以在哪里设置

用的 JVM 调优的参数都有哪些？

说一下 JVM 调优的工具？

Java内存泄露的排查思路？

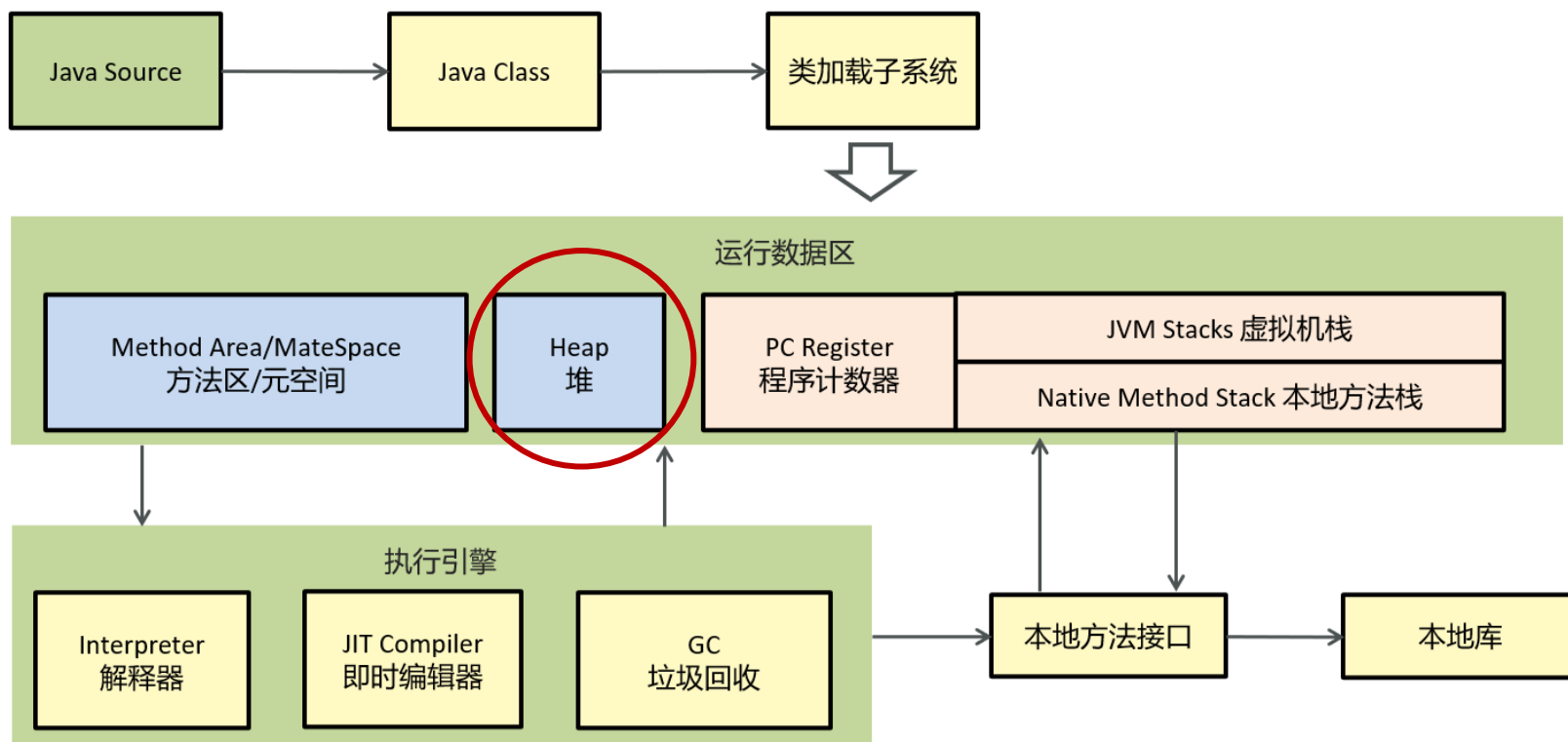
CPU飙高排查方案与思路？

# 对象什么时候可以被垃圾器回收

难易程度：★★★★☆

出现频率：★★★★☆

## 对象什么时候可以被垃圾器回收



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

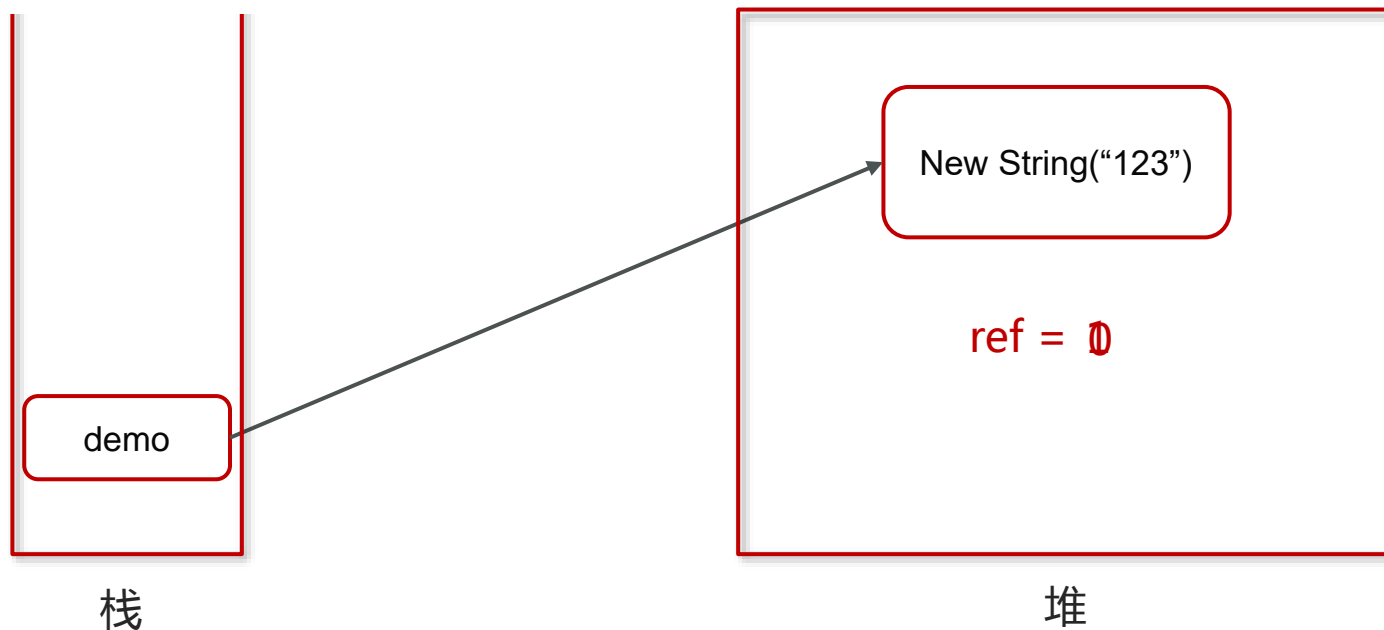
## 引用计数法

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;
```



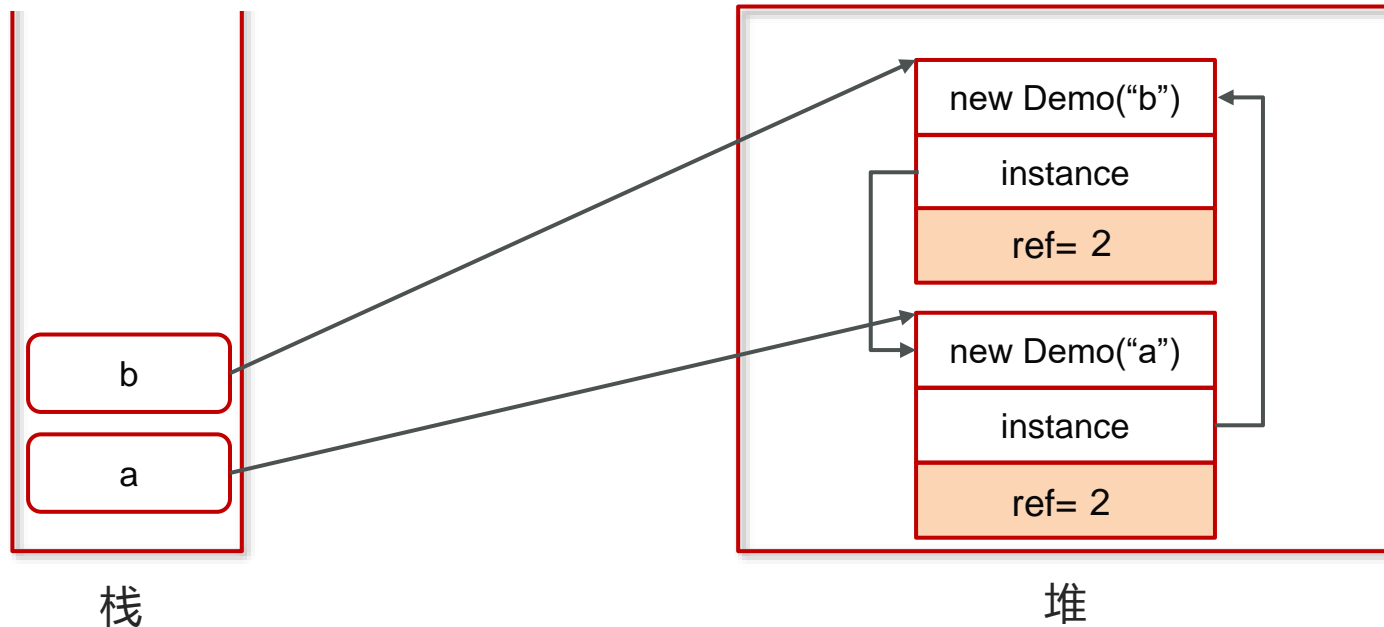
## 引用计数法

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;
```



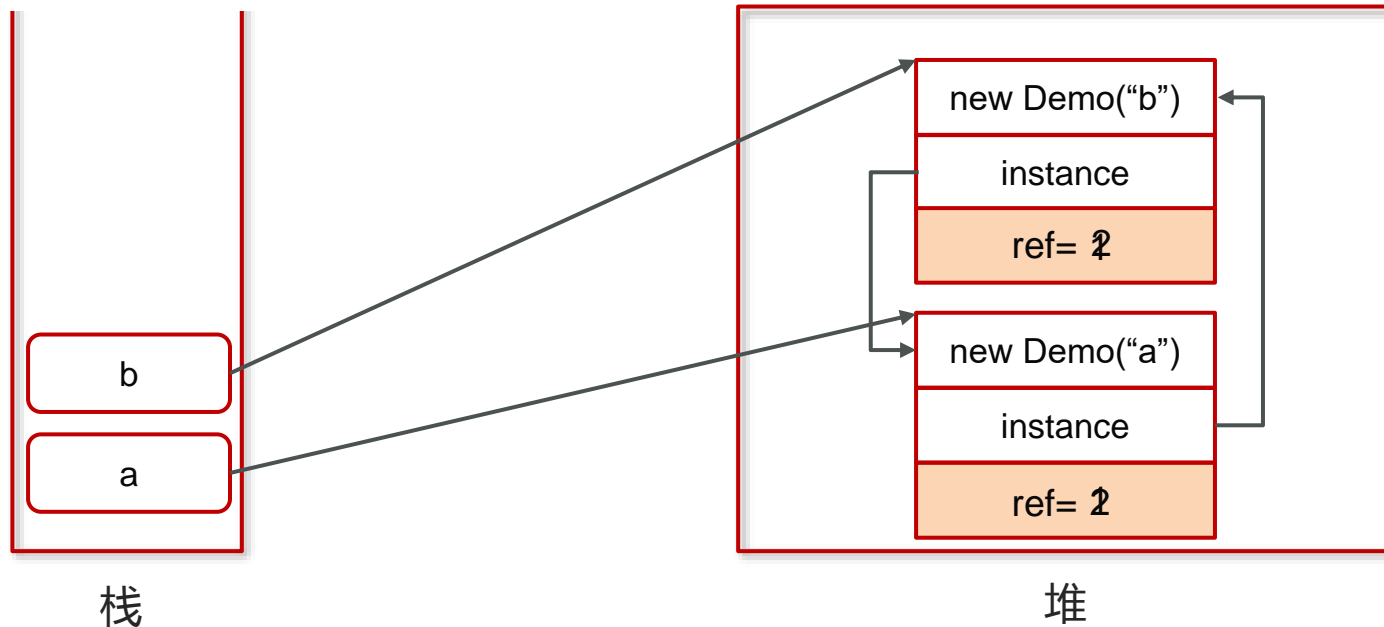
## 引用计数法

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;
```

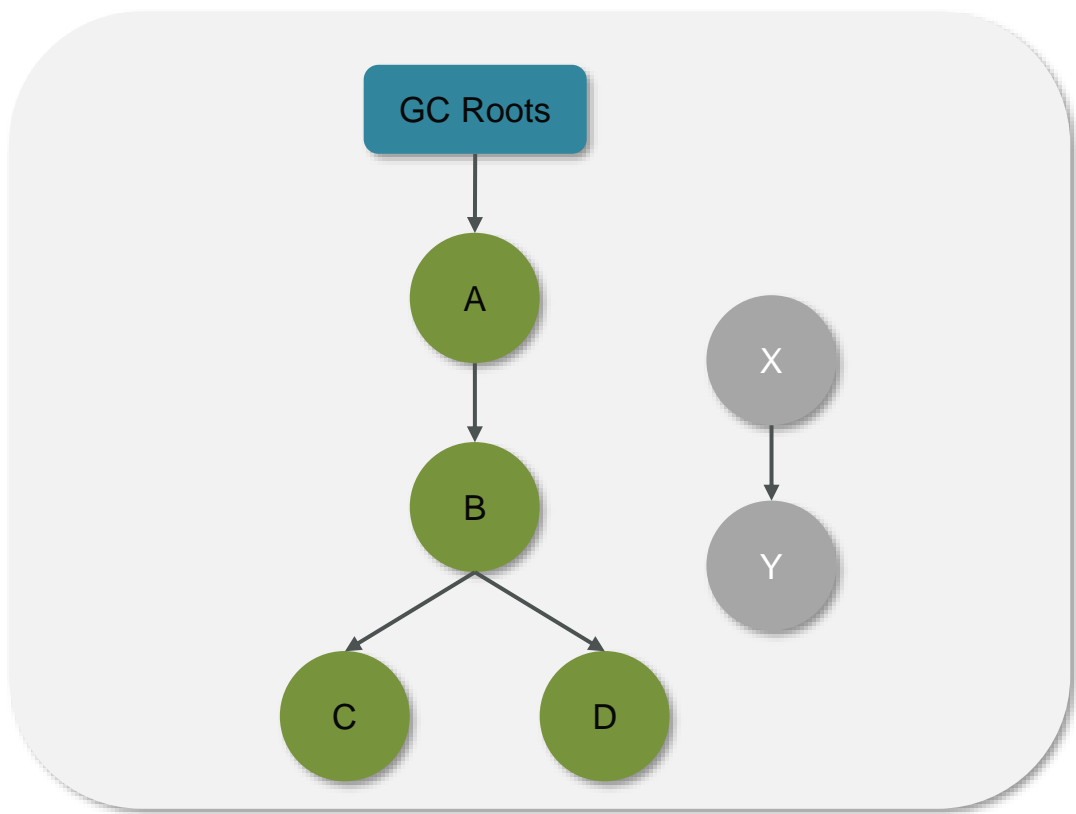
```
a = null;  
b = null;
```



循环引用，会引发内存泄露

## 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。



**X,Y这两个节点是可回收的**

- Java 虚拟机中的垃圾回收器采用可达性分析来探索所有存活的对象
- 扫描堆中的对象，看是否能够沿着 GC Root 对象为起点的引用链找到该对象，找不到，表示可以回收

**哪些对象可以作为 GC Root ?**



## 哪些对象可以作为 GC Root ?

- 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

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

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

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



# 总结

## 对象什么时候可以被垃圾器回收

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，则有可能被垃圾回收器回收。

定位垃圾的方式有两种

- 引用计数法
- 可达性分析算法

# JVM 垃圾回收算法有哪些？

难易程度： ★★★★★

出现频率： ★★★★★

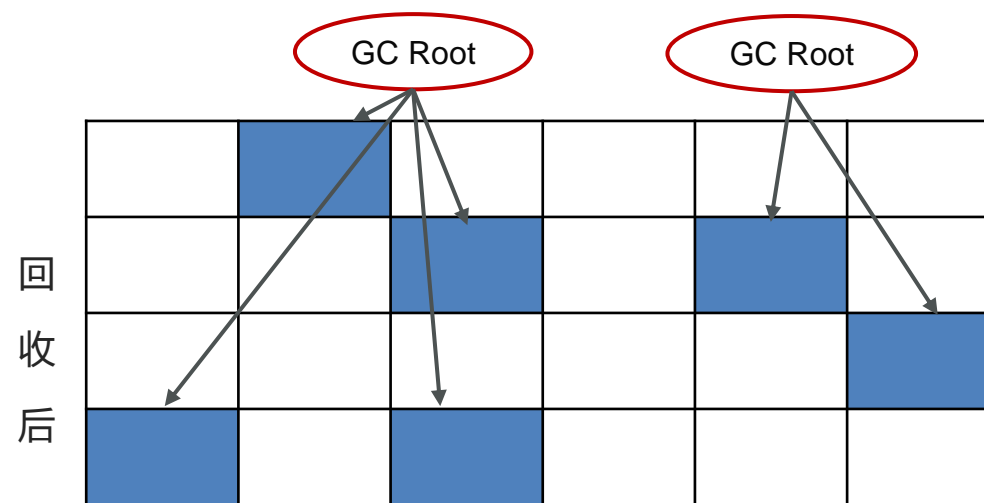
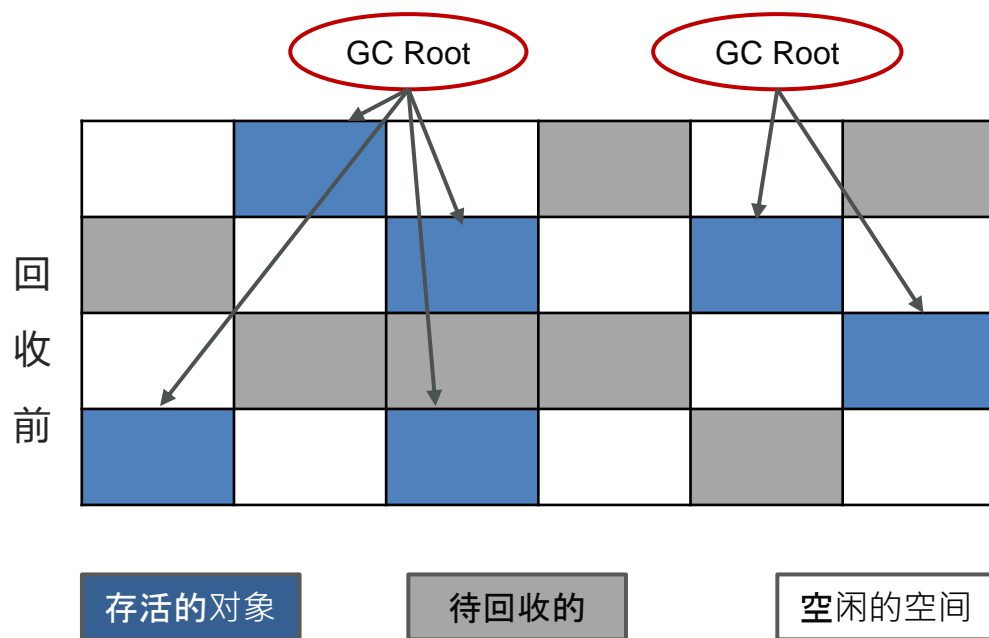
## JVM 垃圾回收算法有哪些？

- 标记清除算法
- 复制算法
- 标记整理算法

## 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**。

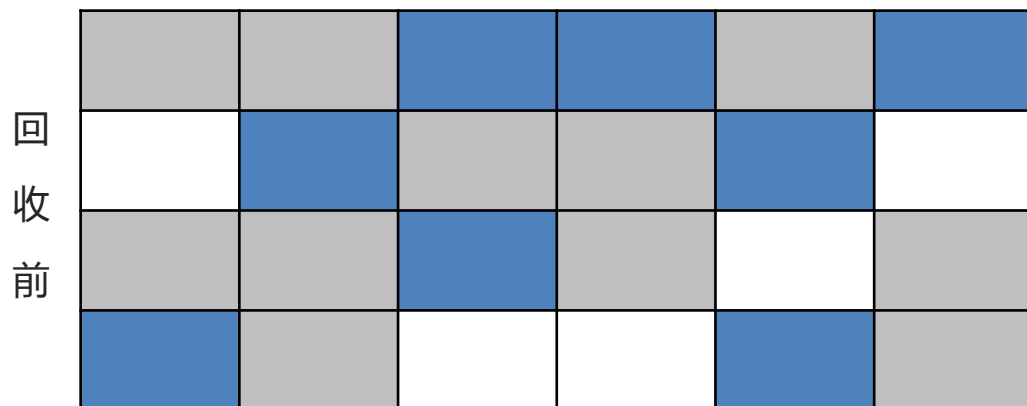
- 1.根据可达性分析算法得出的垃圾进行标记
- 2.对这些标记为可回收的内容进行垃圾回收



优点：标记和清除速度较快

缺点：碎片化较为严重，内存不连贯的

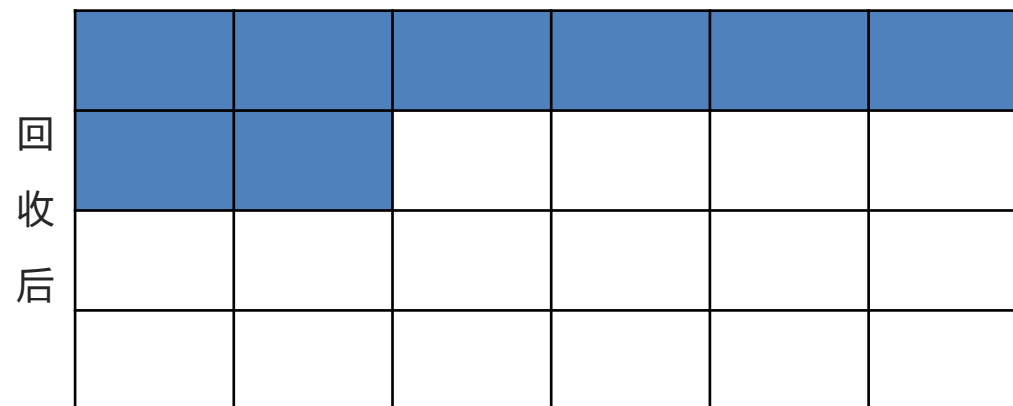
## 标记整理算法



存活的对象

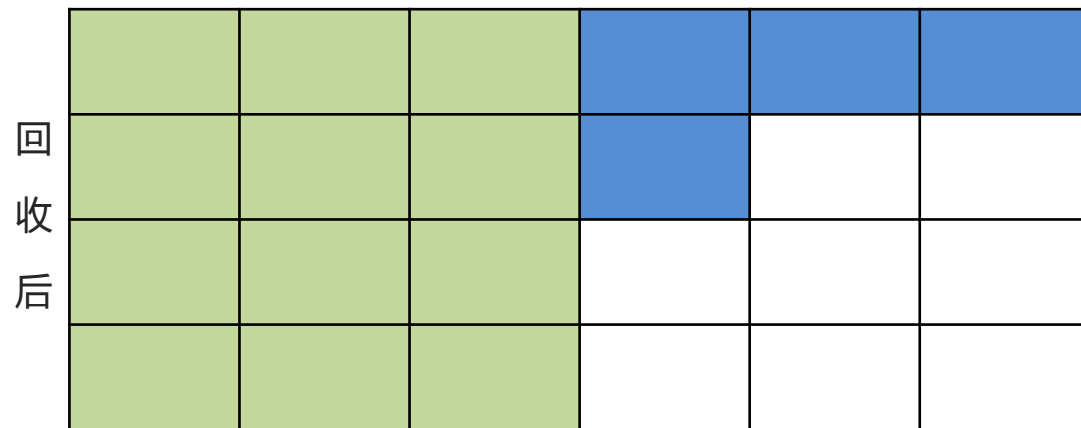
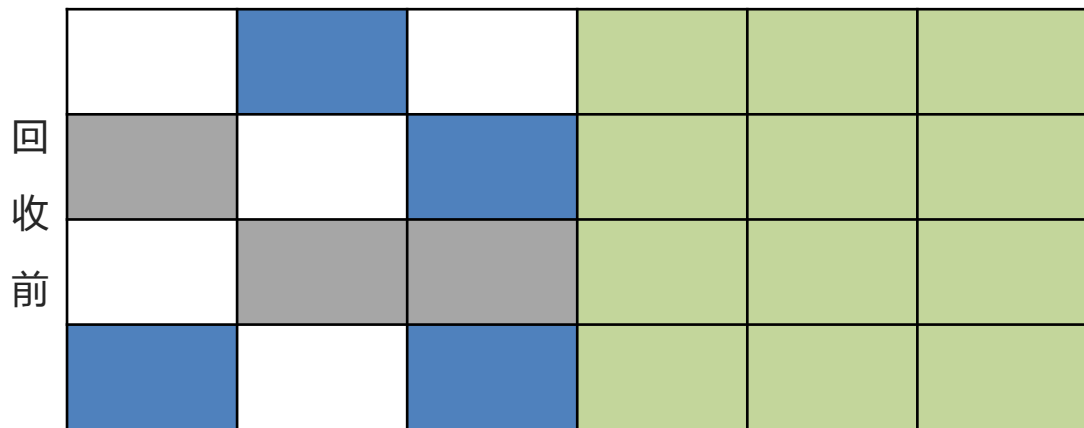
待回收的

空闲的空间



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。

## 复制算法



存活的对象

待回收的

另一块内存

空闲的空间

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

# 总结

## JVM 垃圾回收算法有哪些？

- **标记清除算法**：垃圾回收分为2个阶段，分别是标记和清除,效率高,有磁盘碎片，内存不连续
- **标记整理算法**：标记清除算法一样，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，无碎片，对象需要移动，效率低
- **复制算法**：将原有的内存空间一分为二，每次只用其中的一块,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;无碎片，内存使用率低



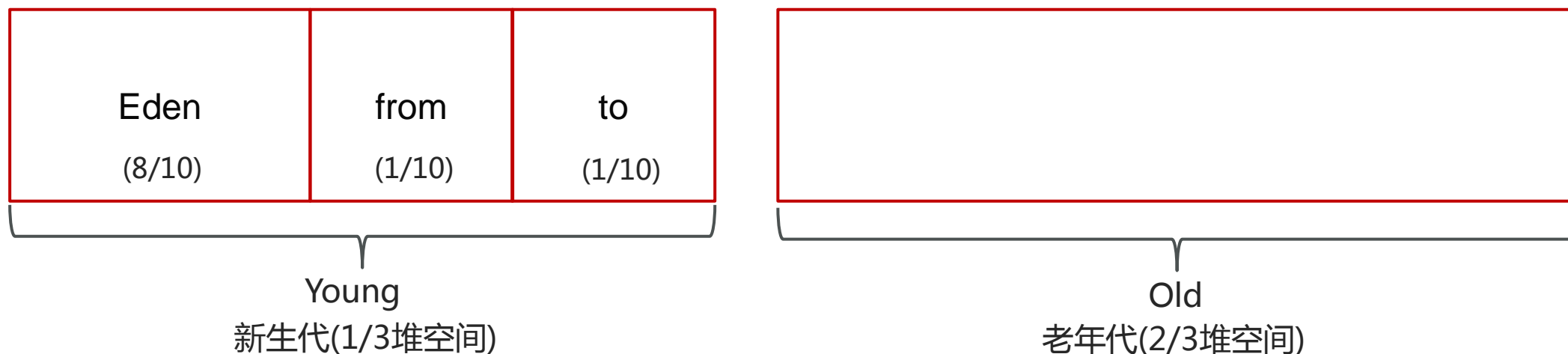
# 说一下JVM中的分代回收

难易程度： ★★★★★

出现频率： ★★★★★

## 分代收集算法

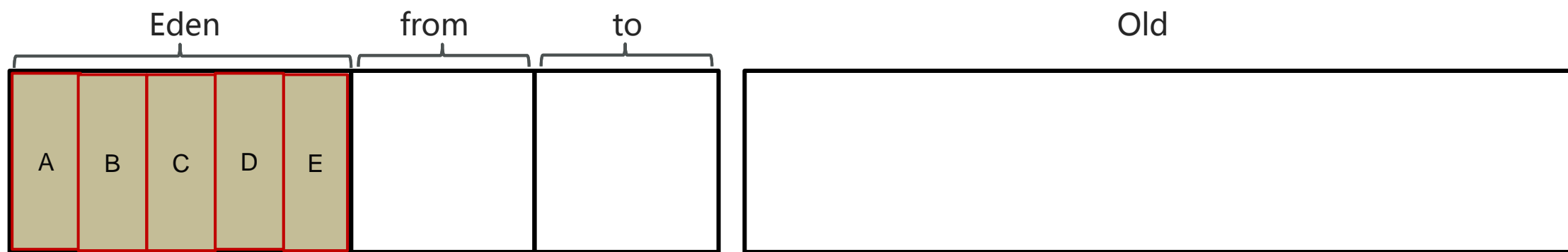
在java8时，堆被分为了两份：**新生代和老年代【1：2】**



对于新生代，内部又被分为了三个区域。

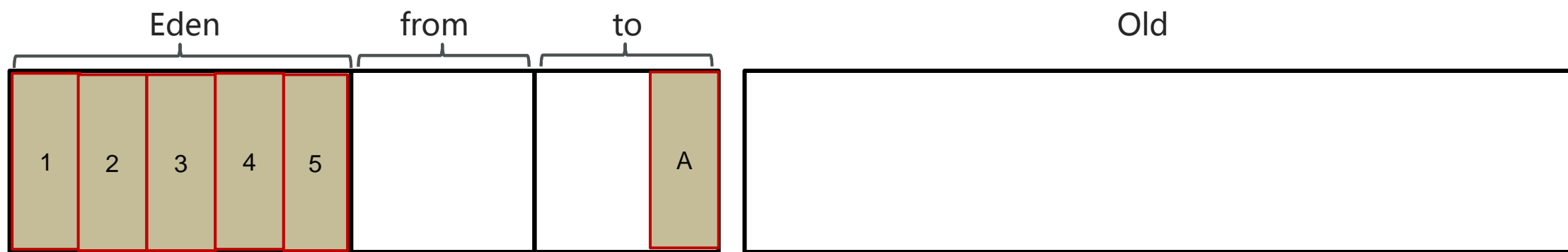
- 伊甸园区Eden，新生的对象都分配到这里
- 幸存者区survivor(分成from和to)
- Eden区，from区，to区【8：1：1】

## 分代收集算法-工作机制



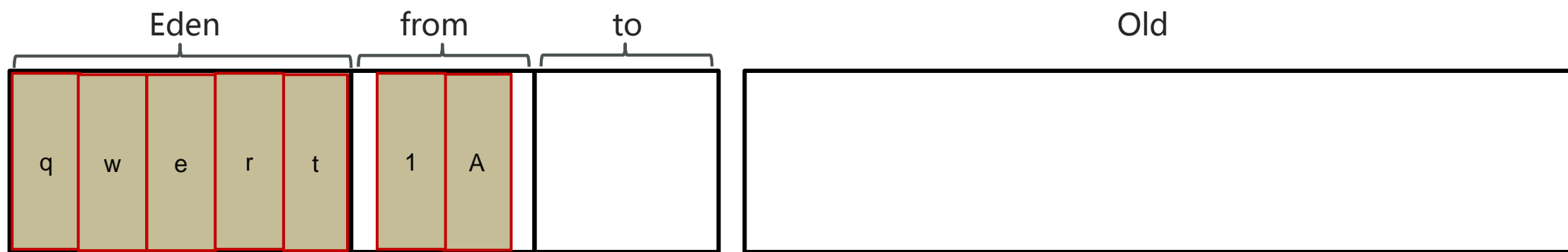
- 新创建的对象，都会先分配到eden区
- 当伊甸园内存不足，标记伊甸园与 from（现阶段没有）的存活对象
- 将存活对象采用复制算法复制到 to 中，复制完毕后，伊甸园和 from 内存都得到释放

## 分代收集算法-工作机制



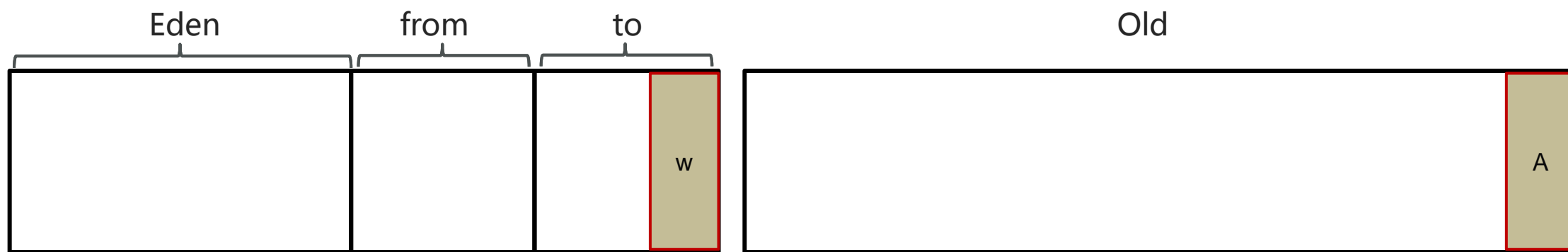
- 新创建的对象，都会先分配到eden区
- 当伊甸园内存不足，标记伊甸园与 from（现阶段没有）的存活对象
- 将存活对象采用复制算法复制到 to 中，复制完毕后，伊甸园和 from 内存都得到释放
- 经过一段时间后伊甸园的内存又出现不足，标记eden区域to区存活的对象，将存活的对象复制到from区

## 分代收集算法-工作机制



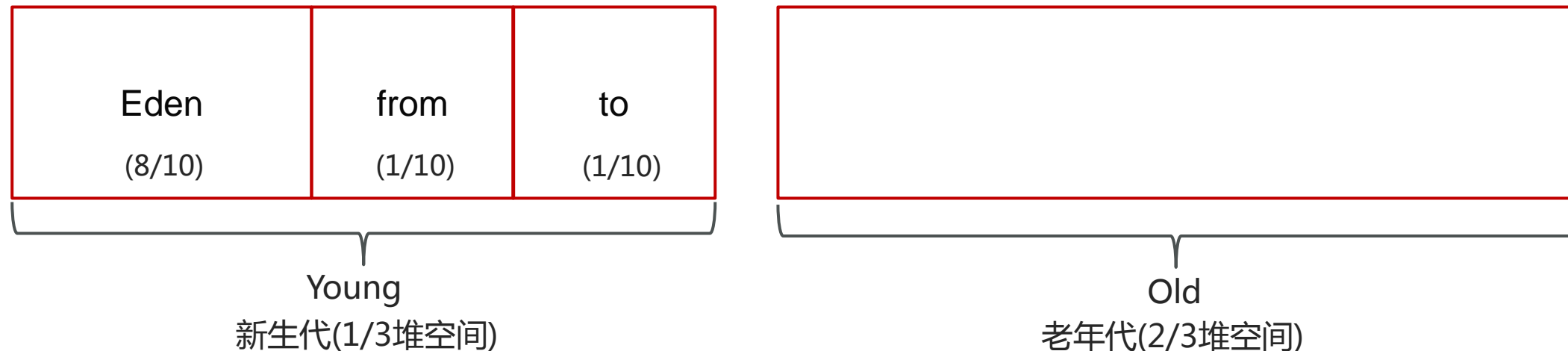
- 新创建的对象，都会先分配到eden区
- 当伊甸园内存不足，标记伊甸园与 from（现阶段没有）的存活对象
- 将存活对象采用复制算法复制到 to 中，复制完毕后，伊甸园和 from 内存都得到释放
- 经过一段时间后伊甸园的内存又出现不足，标记eden区域to区存活的对象，将存活的对象复制到from区
- 当幸存区对象熬过几次回收（最多15次），晋升到老年代（幸存区内存不足或大对象会导致提前晋升）

## 分代收集算法-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



## 总结

### 说一下JVM中的分代回收

#### 一、堆的区域划分

1. 堆被分为了两份：新生代和老年代【1：2】
2. 对于新生代，内部又被分为了三个区域。Eden区，幸存者区survivor(分成from和to)【8：1：1】

#### 二、对象回收分代回收策略

1. 新创建的对象，都会先分配到eden区
2. 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
3. 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
4. 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，将其复制到from区
5. 当幸存者对象熬过几次回收（最多15次），晋升到老年代（幸存者内存不足或大对象会提前晋升）

### 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



# 说一下JVM有哪些垃圾回收器？

难易程度：★★★★☆

出现频率：★★★★☆

## 说一下 JVM 有哪些垃圾回收器？

在jvm中，实现了多种垃圾收集器，包括：

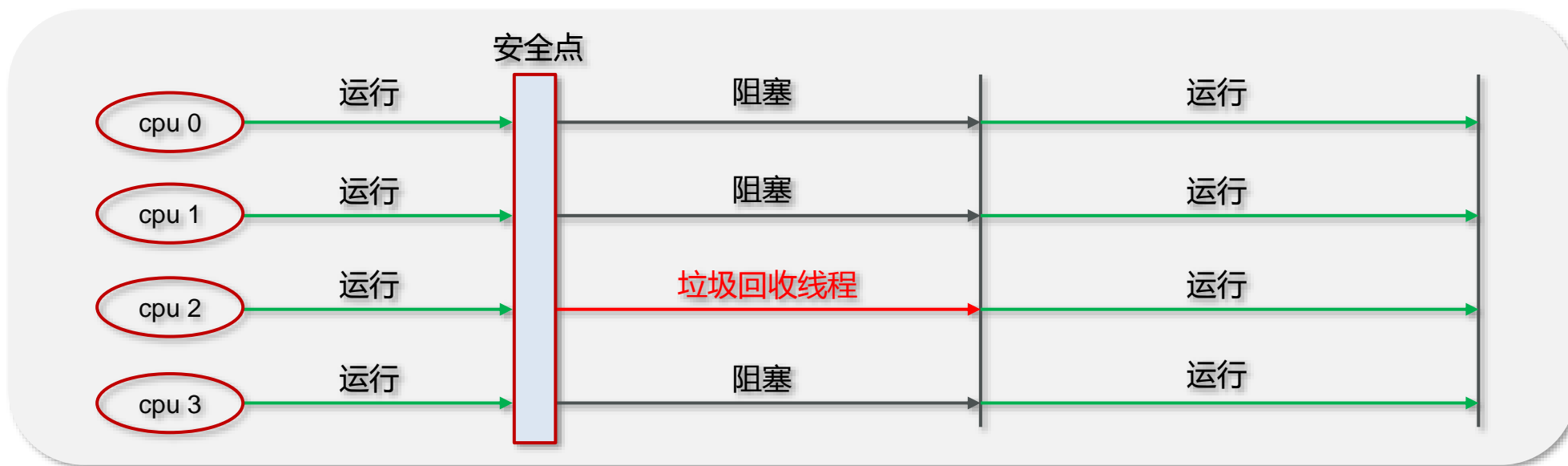
- 串行垃圾收集器
- 并行垃圾收集器
- CMS（并发）垃圾收集器
- G1垃圾收集器

## 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。

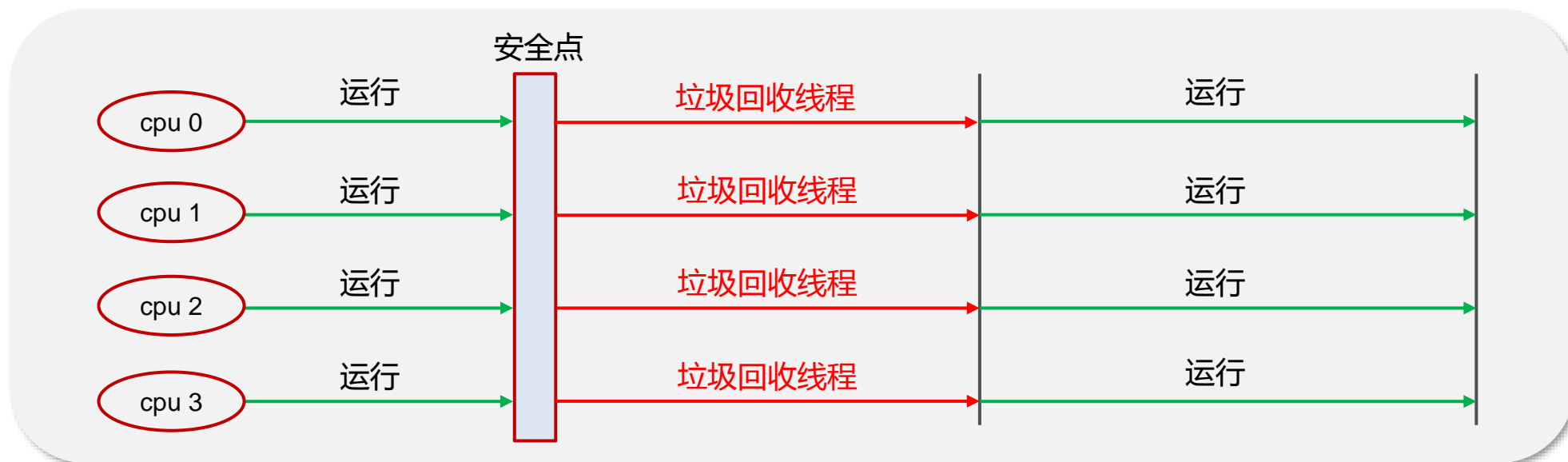


## 并行垃圾收集器

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**

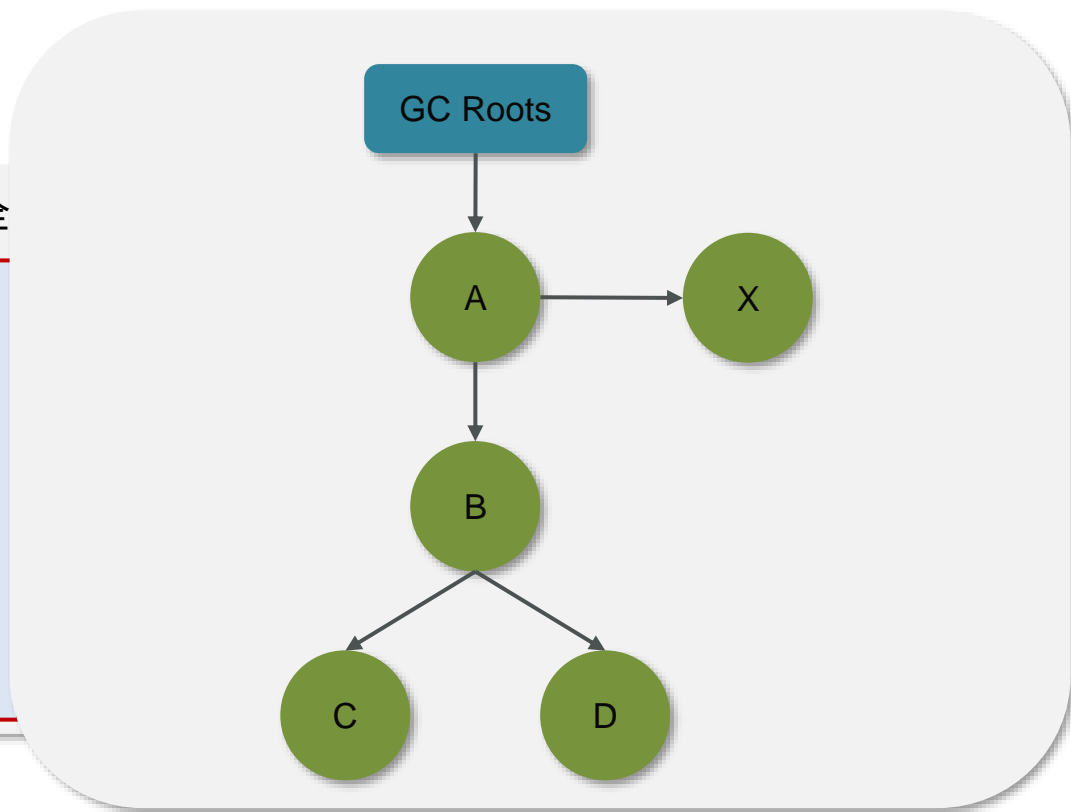
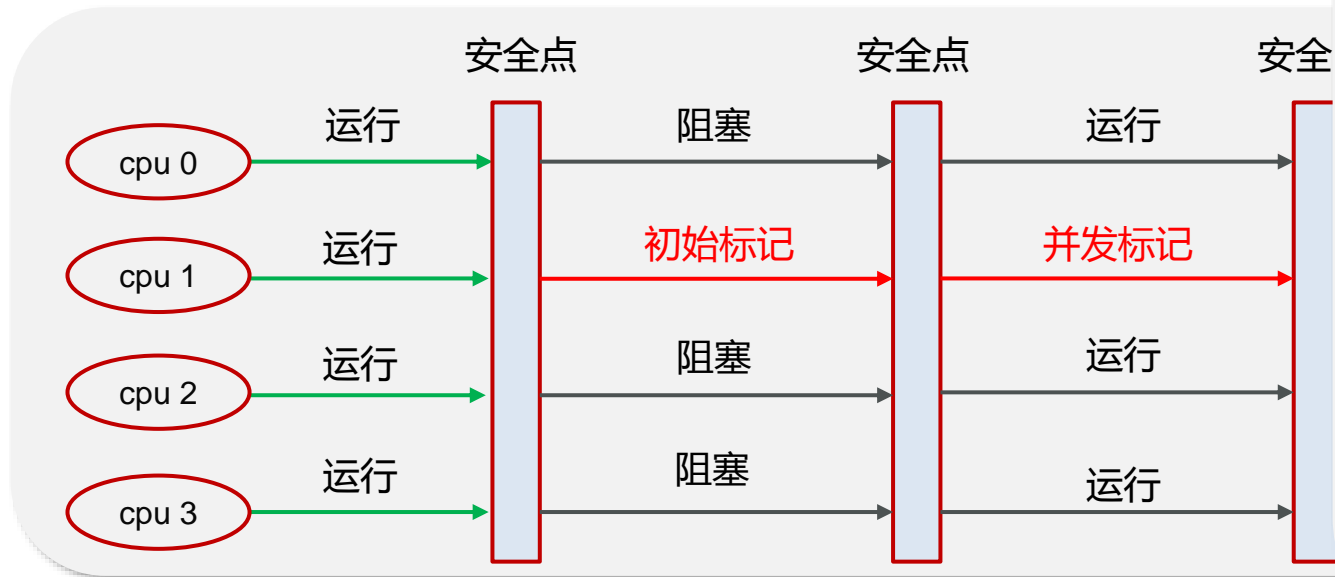
- Parallel New作用于新生代，采用复制算法
- Parallel Old作用于老年代，采用标记-整理算法

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。



## 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。





# 总结

说一下JVM有哪些垃圾回收器？

在jvm中，实现了多种垃圾收集器，包括：

- 串行垃圾收集器：Serial GC、Serial Old GC
- 并行垃圾收集器：Parallel Old GC、ParNew GC
- CMS（并发）垃圾收集器：CMS GC，作用在老年代
- G1垃圾收集器，作用在新生代和老年代

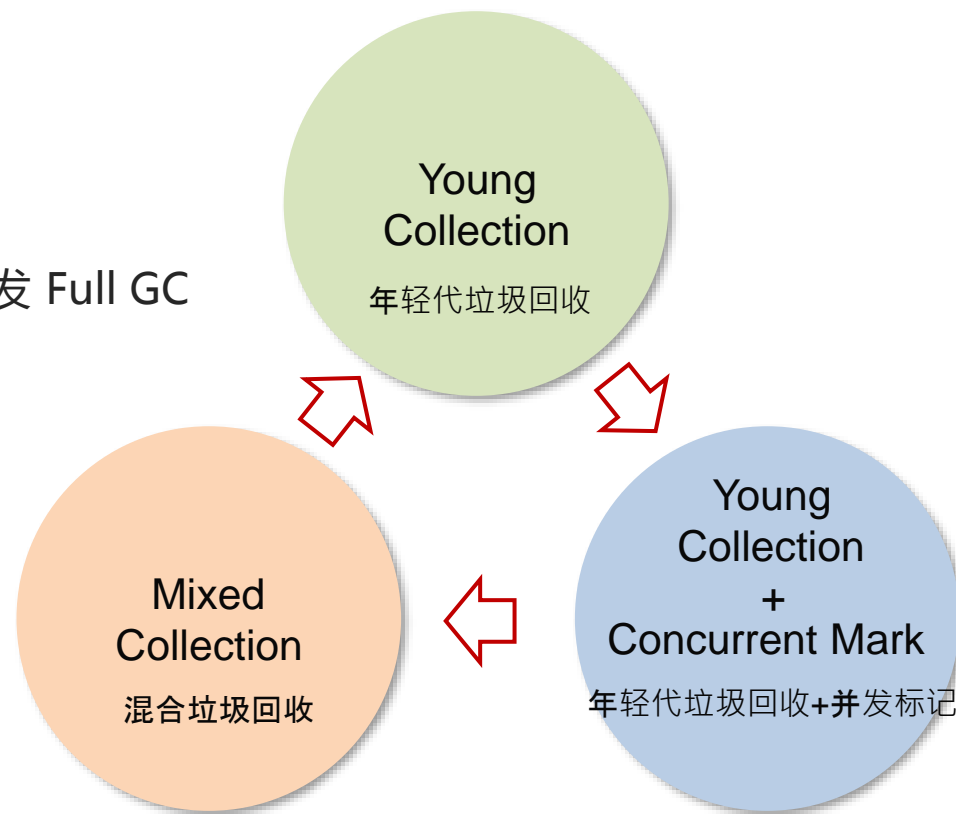
# 详细聊一下G1垃圾回收器

难易程度：★★★★☆

出现频率：★★★★☆

## G1垃圾收集器

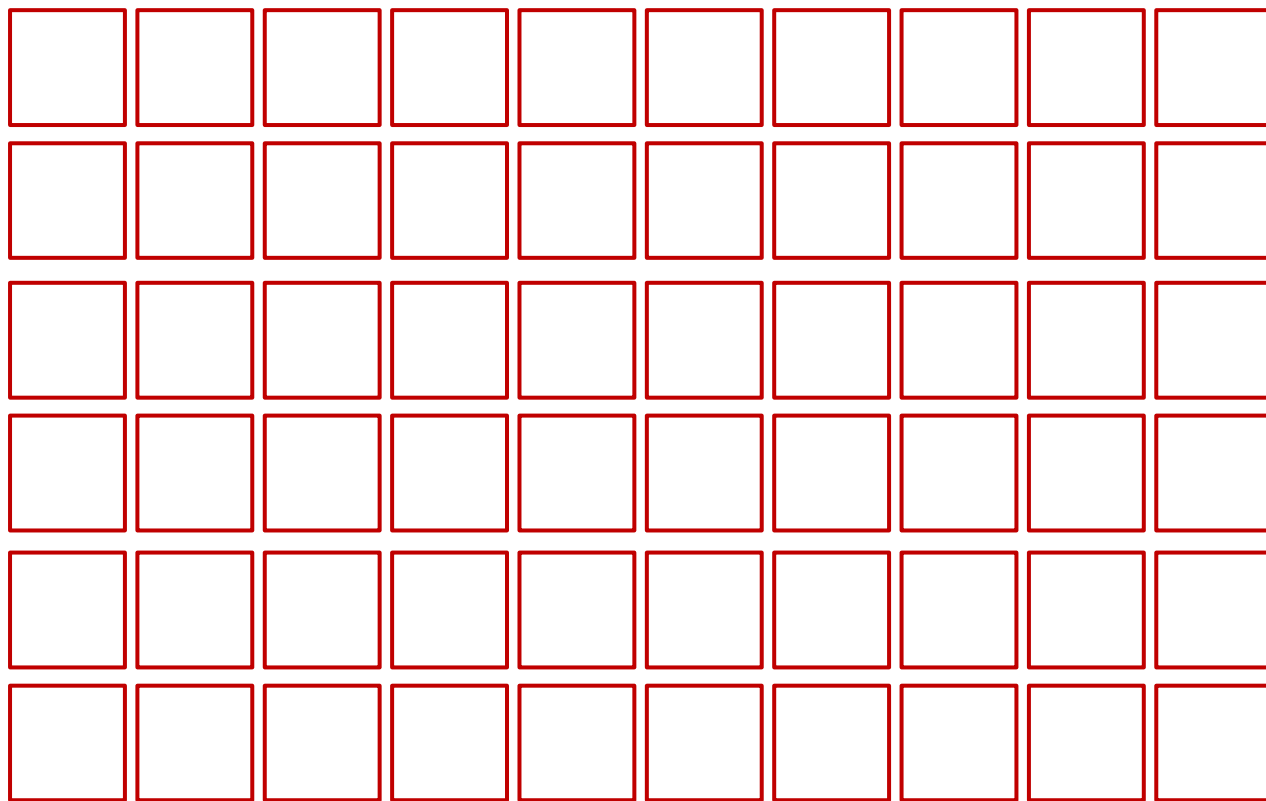
- 应用于新生代和老年代，**在JDK9之后默认使用G1**
- 划分成多个区域，每个区域都可以充当 eden，survivor，old，humongous，其中 humongous 专为大对象准备
- 采用复制算法
- 响应时间与吞吐量兼顾
- 分成三个阶段：新生代回收、并发标记、混合收集
- 如果并发失败（即回收速度赶不上创建新对象速度），会触发 Full GC





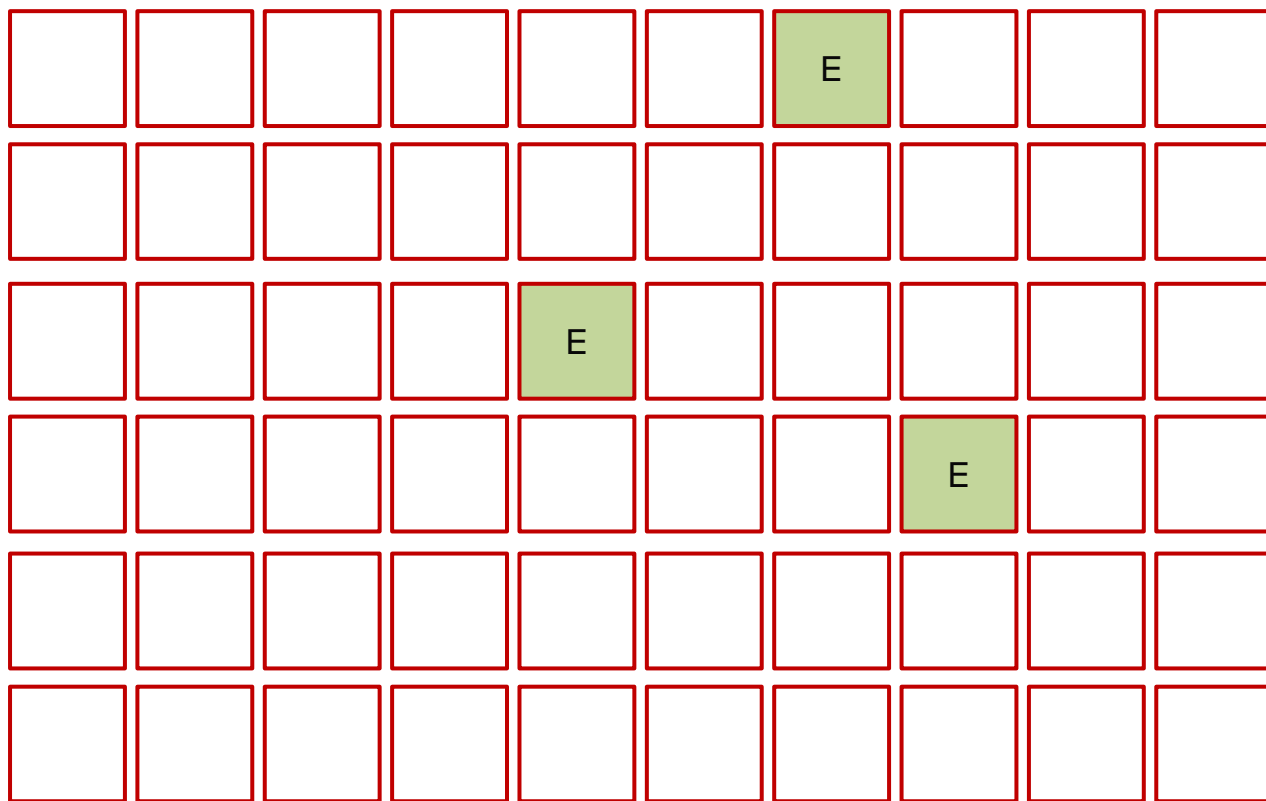
## Young Collection(年轻代垃圾回收)

- 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



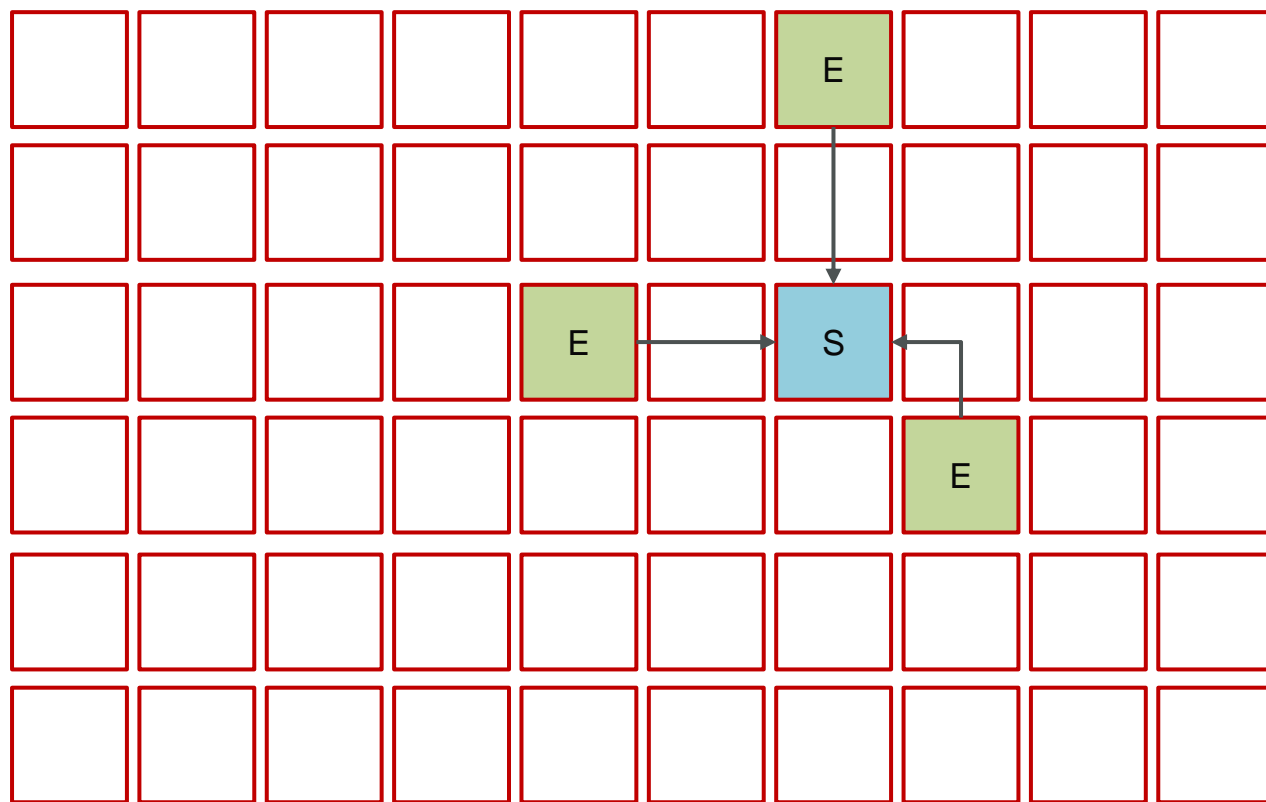
## Young Collection(年轻代垃圾回收)

- 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



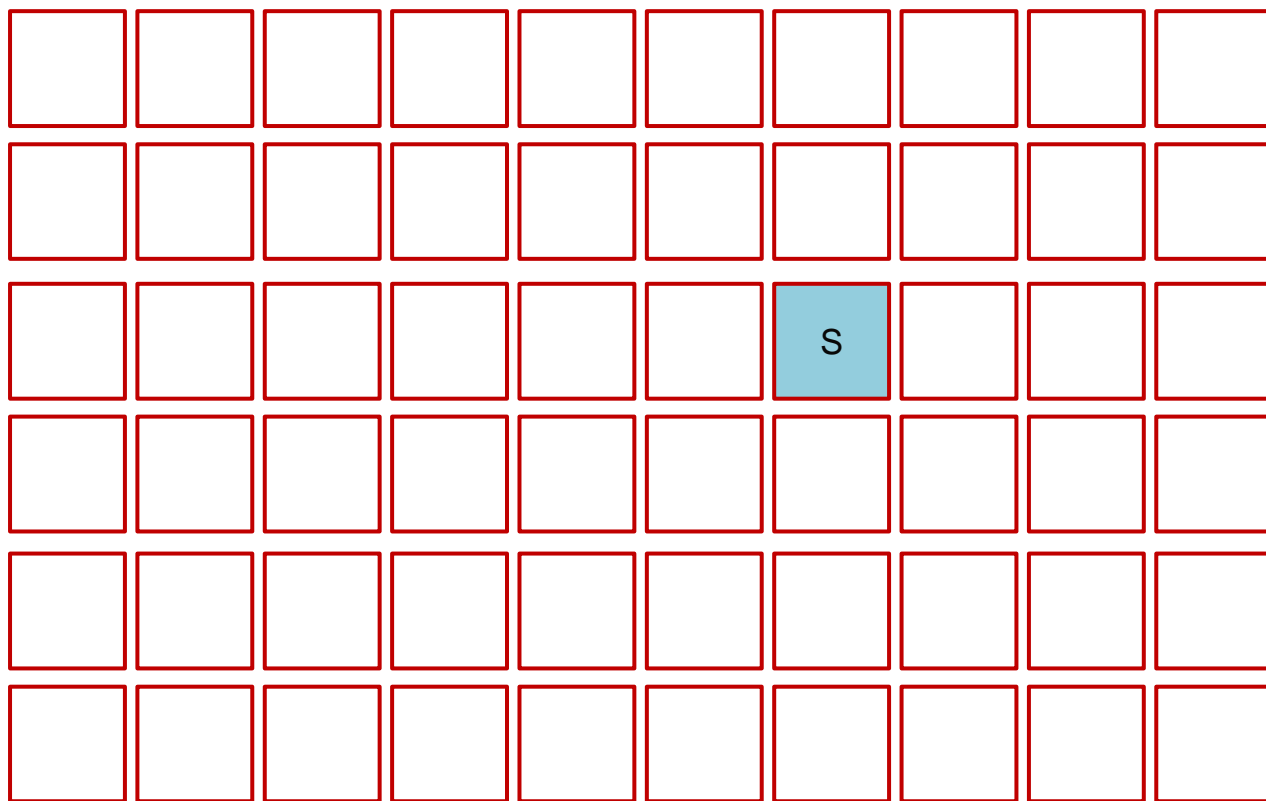
## Young Collection(年轻代垃圾回收)

- 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



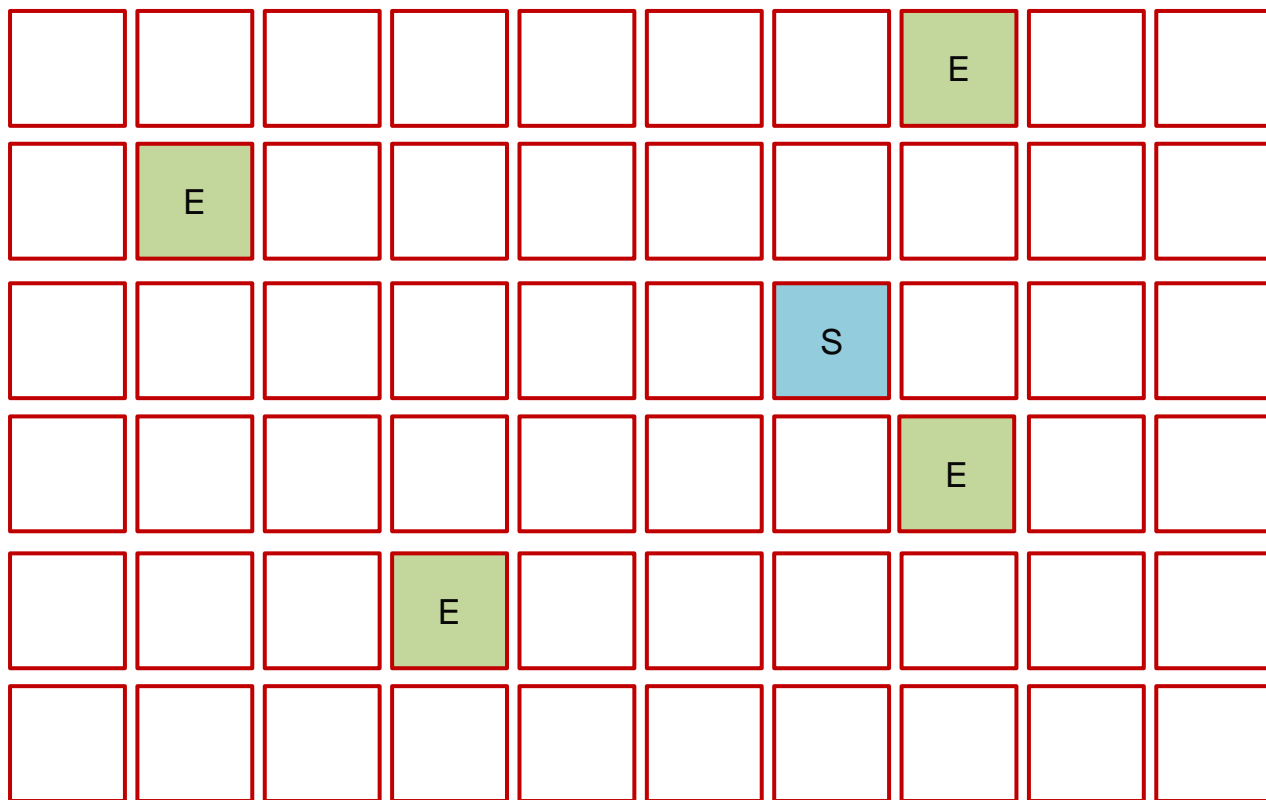
## Young Collection(年轻代垃圾回收)

- 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



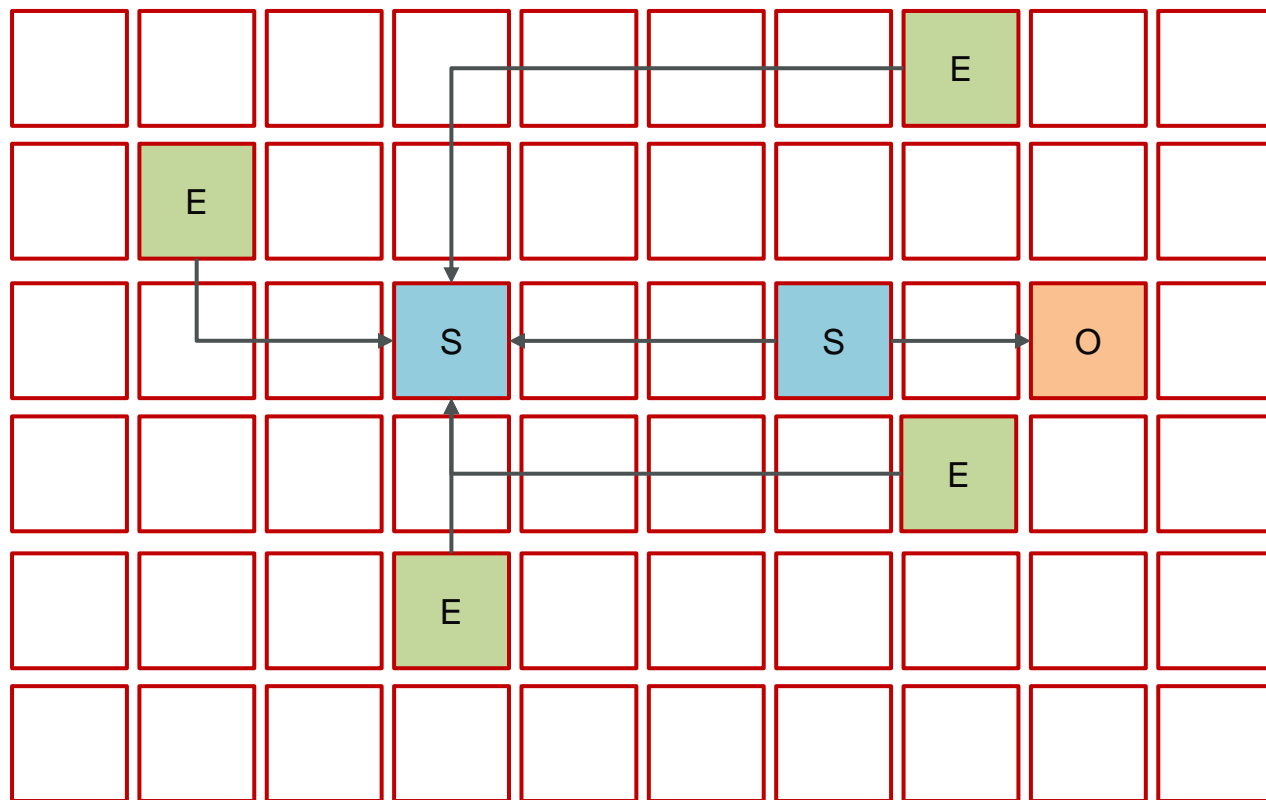
## Young Collection(年轻代垃圾回收)

- 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



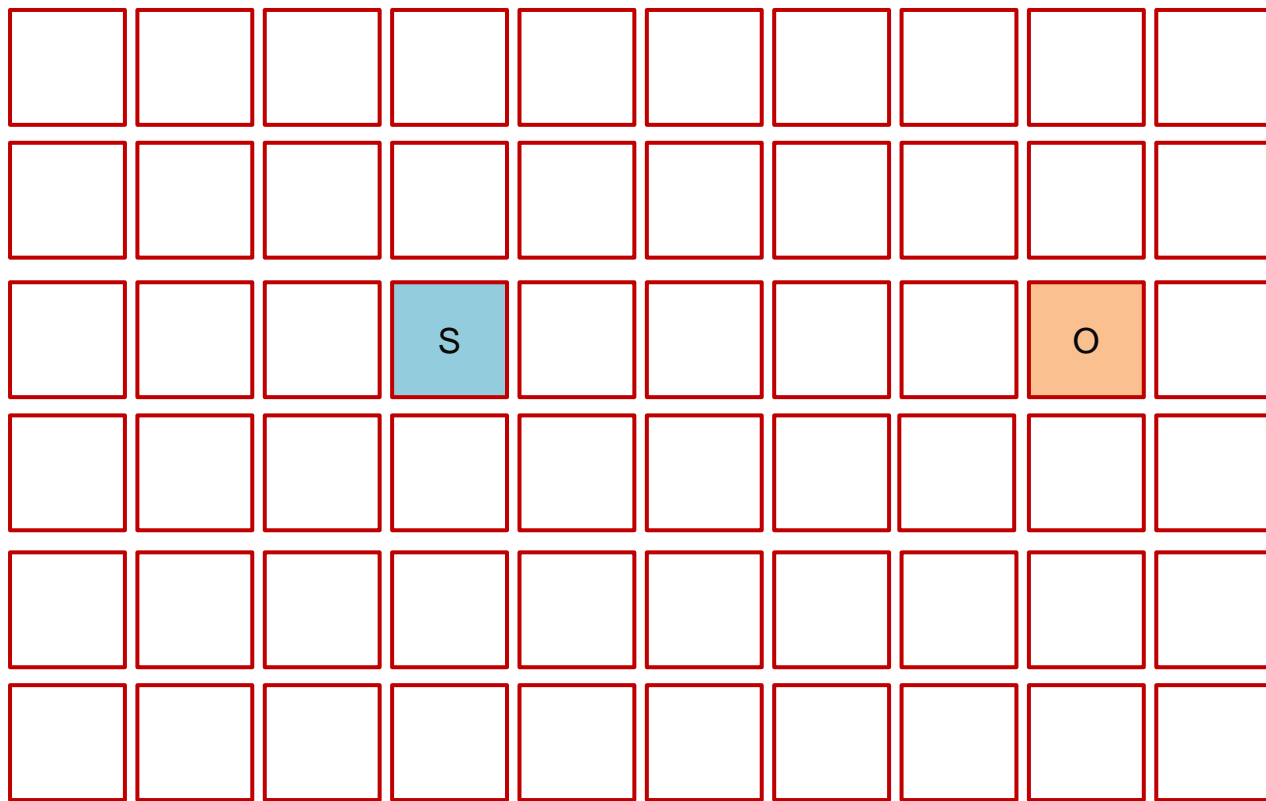
## Young Collection(年轻代垃圾回收)

- 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



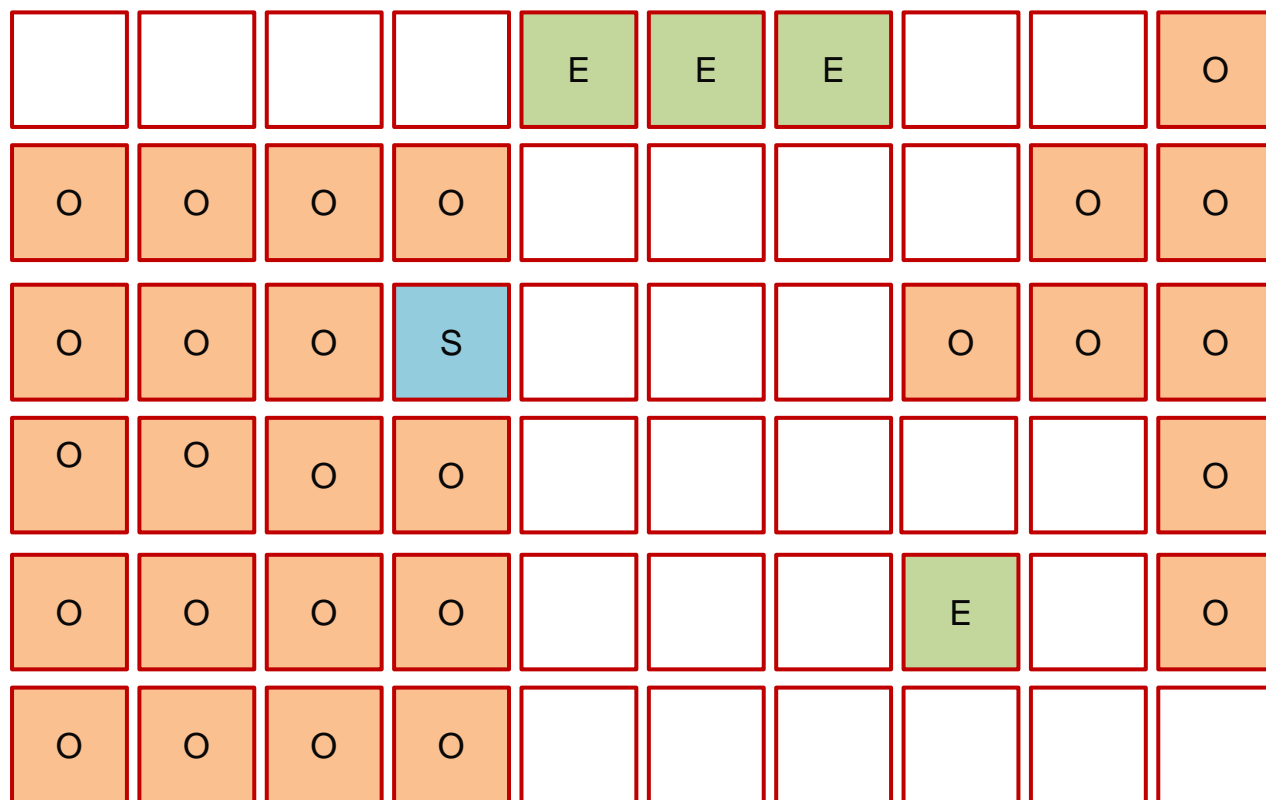
## Young Collection(年轻代垃圾回收)

- 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



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

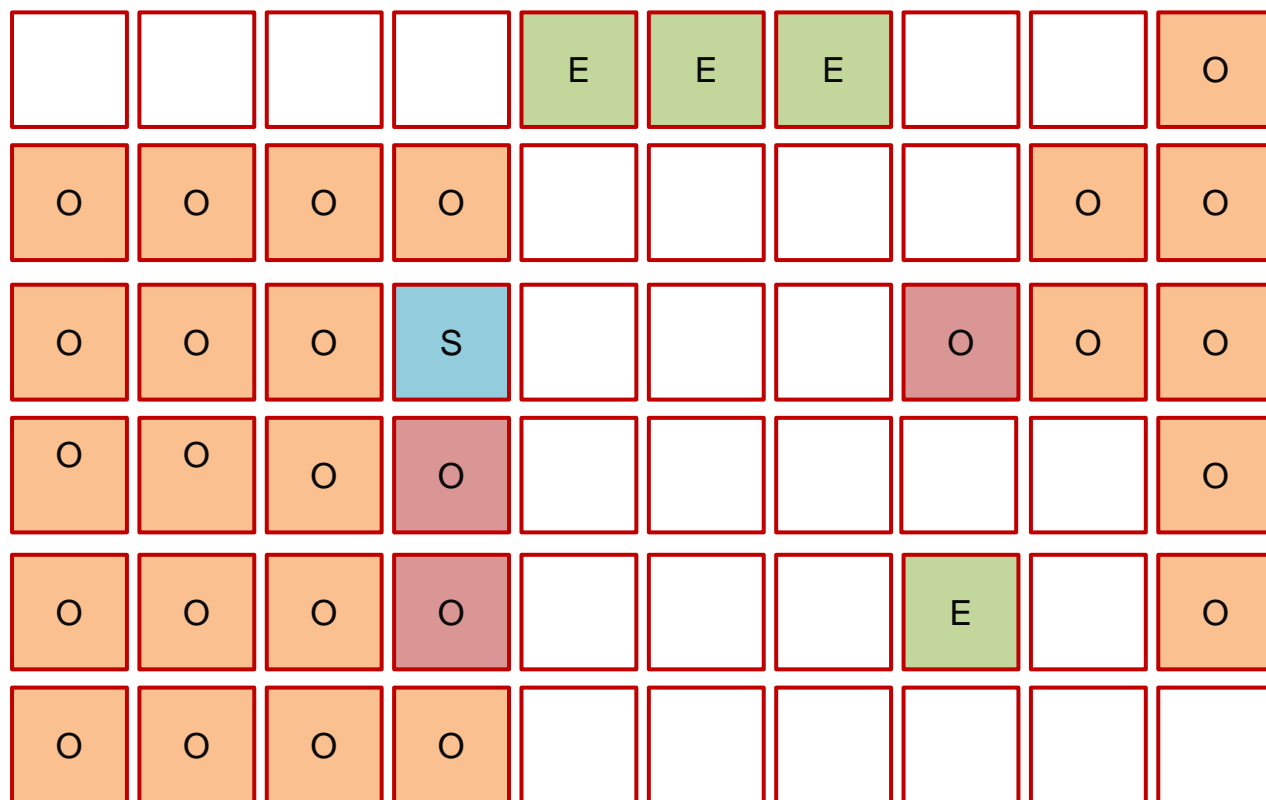
当老年代占用内存超过阈值(默认是45%)后，触发并发标记，这时无需暂停用户线程





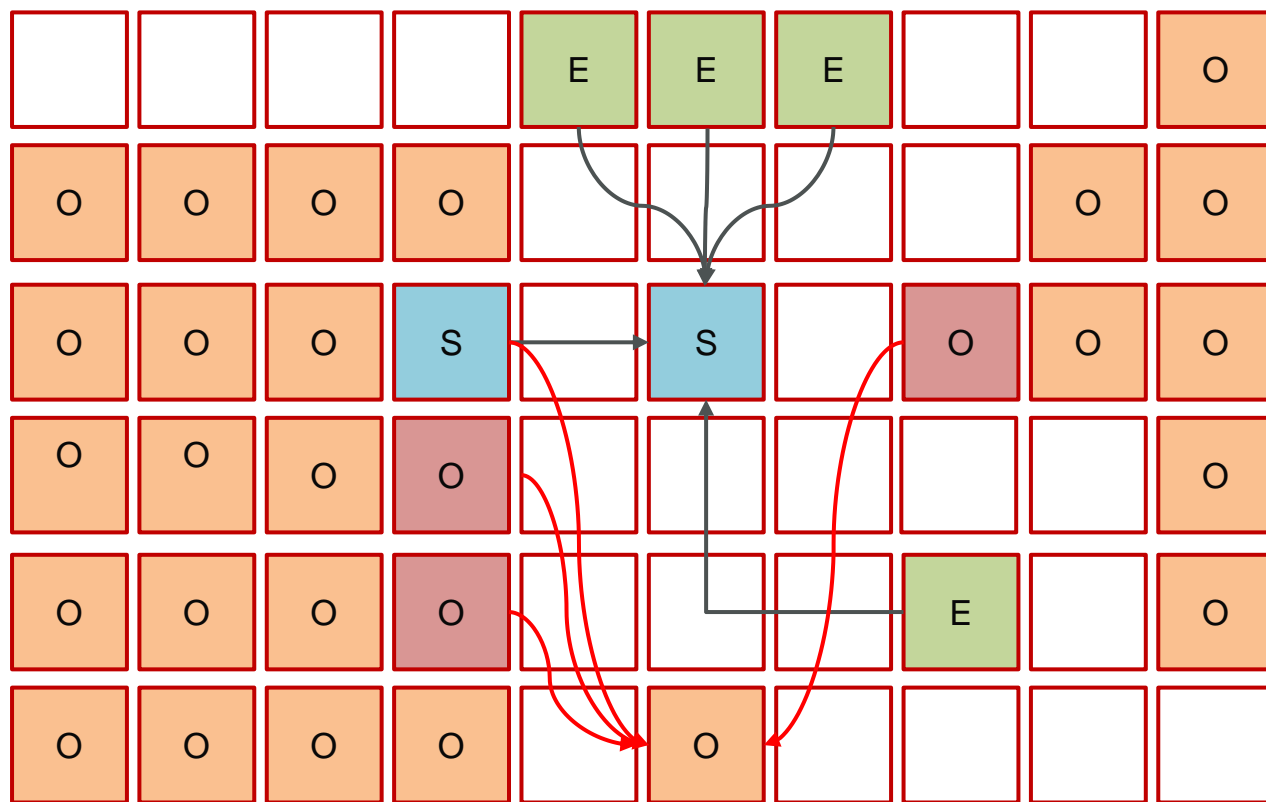
## Young Collection + Concurrent Mark (年轻代垃圾回收+并发标记)

- 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）。



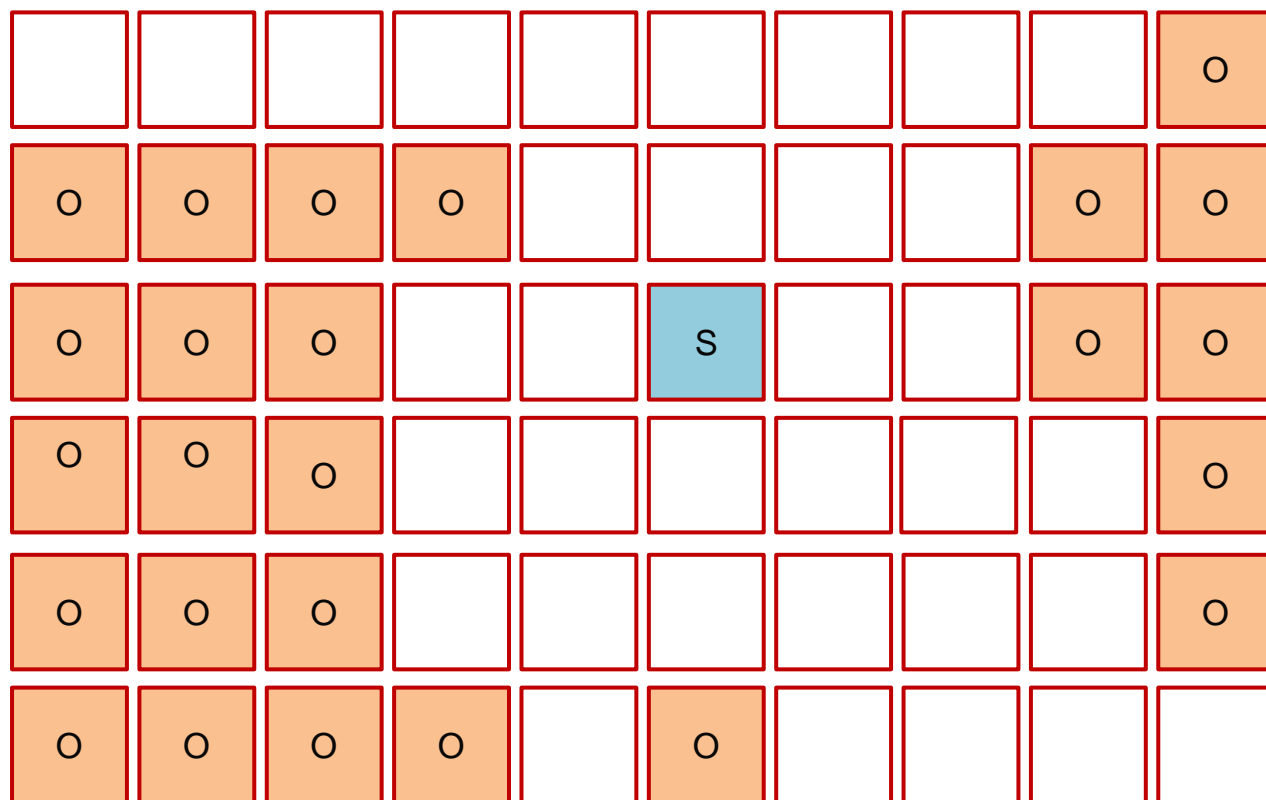
## Mixed Collection (混合垃圾回收)

混合收集阶段中，参与复制的有 eden、survivor、old



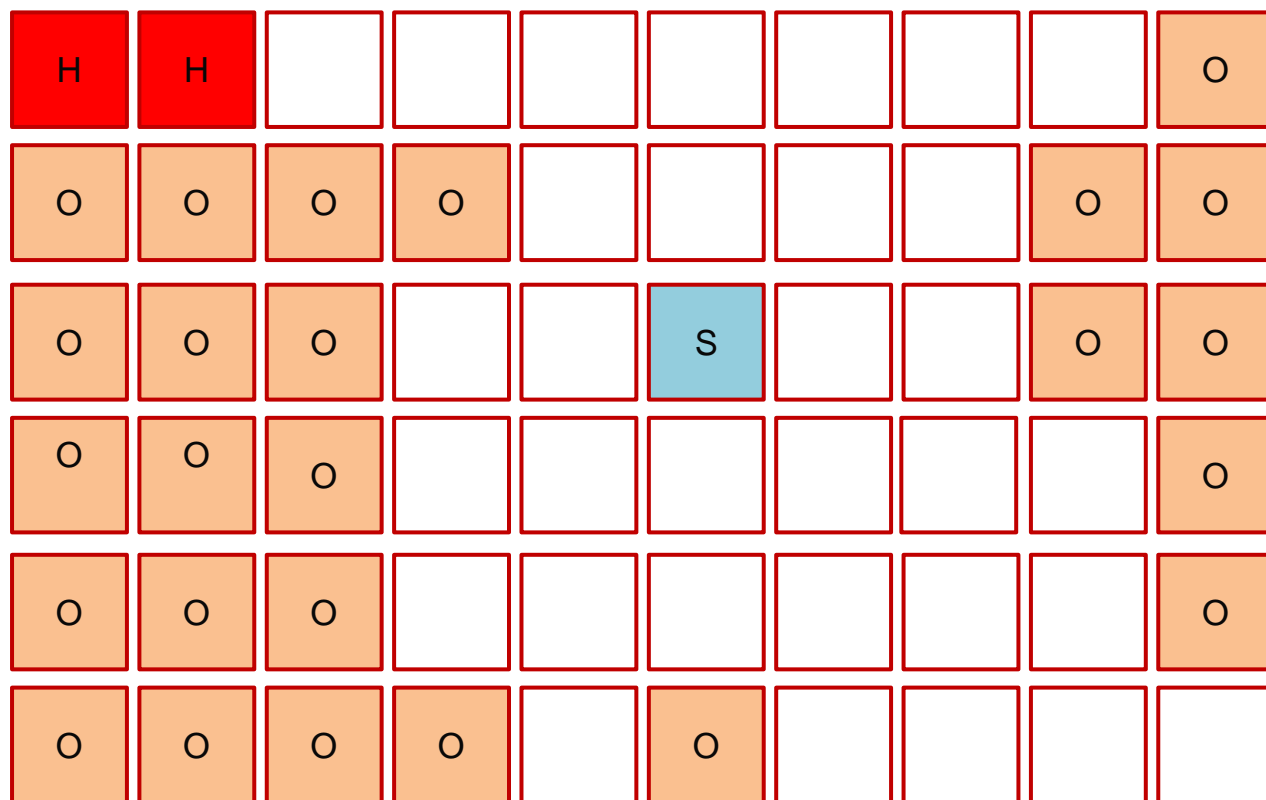
## Mixed Collection (混合垃圾回收)

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



## 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





# 总结

## 详细聊一下G1垃圾回收器

- 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(stw)、并发标记(重新标记stw)、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

# 强引用、软引用、弱引用、虚引用的区别

难易程度：★★★★☆

出现频率：★★★☆☆

## 强引用、软引用、弱引用、虚引用的区别

- **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();
```



- **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

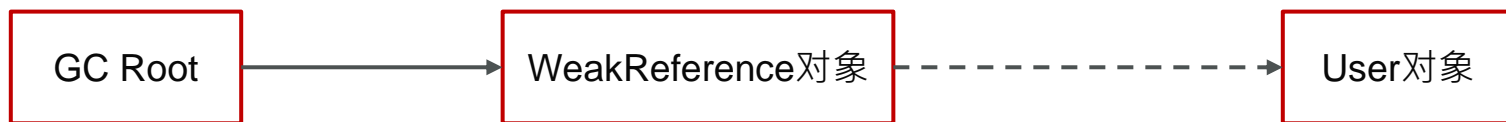
```
User user = new User();  
SoftReference softReference = new SoftReference(user);
```



## 强引用、软引用、弱引用、虚引用的区别

**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



```
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  
    }  
}
```



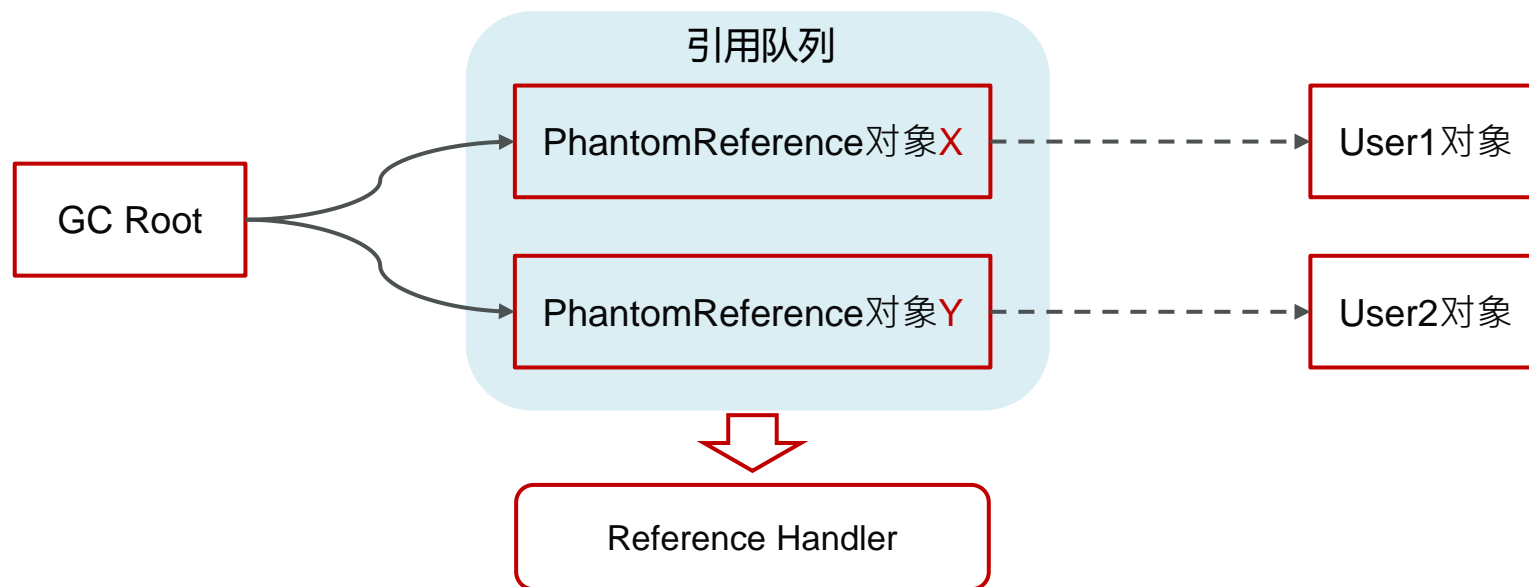
## 强引用、软引用、弱引用、虚引用的区别

**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);
```

软引用

弱引用





# 总结

## 强引用、软引用、弱引用、虚引用的区别?

- 强引用：只要所有 GC Roots 能找到，就不会被回收
- 软引用：需要配合SoftReference使用，当垃圾多次回收，内存依然不够的时候会回收软引用对象
- 弱引用：需要配合WeakReference使用，只要进行了垃圾回收，就会把弱引用对象回收
- 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

## JVM组成

完成

什么是程序计数器

你能给我详细的介绍下堆吗？

能不能介绍一下方法区

你听过直接内存吗

什么是虚拟机栈

垃圾回收是否涉及栈内存？

栈内存分配越大越好吗？

方法内的局部变量是否线程安全？

什么情况下会导致栈内存溢出？

堆栈的区别是什么

## 类加载器

完成

什么是类加载器，类加载器有哪些

什么是双亲委派模型？

JVM为什么采用双亲委派机制？

说一下类装载的执行过程

## 垃圾回收

完成

强引用、软引用、弱引用、虚对象

什么时候可以被垃圾器回收

JVM 垃圾回收算法有哪些？

说一下JVM中的分代回收

说一下JVM有哪些垃圾回收器？

详细聊一下G1垃圾回收器

## JVM实践

JVM 调优的参数可以在哪里设置

用的 JVM 调优的参数都有哪些？

说一下 JVM 调优的工具？

Java内存泄露的排查思路？

CPU飙高排查方案与思路？

# JVM 调优的参数可以在哪里设置参数值

难易程度： ★★☆☆☆

出现频率： ★★☆☆☆

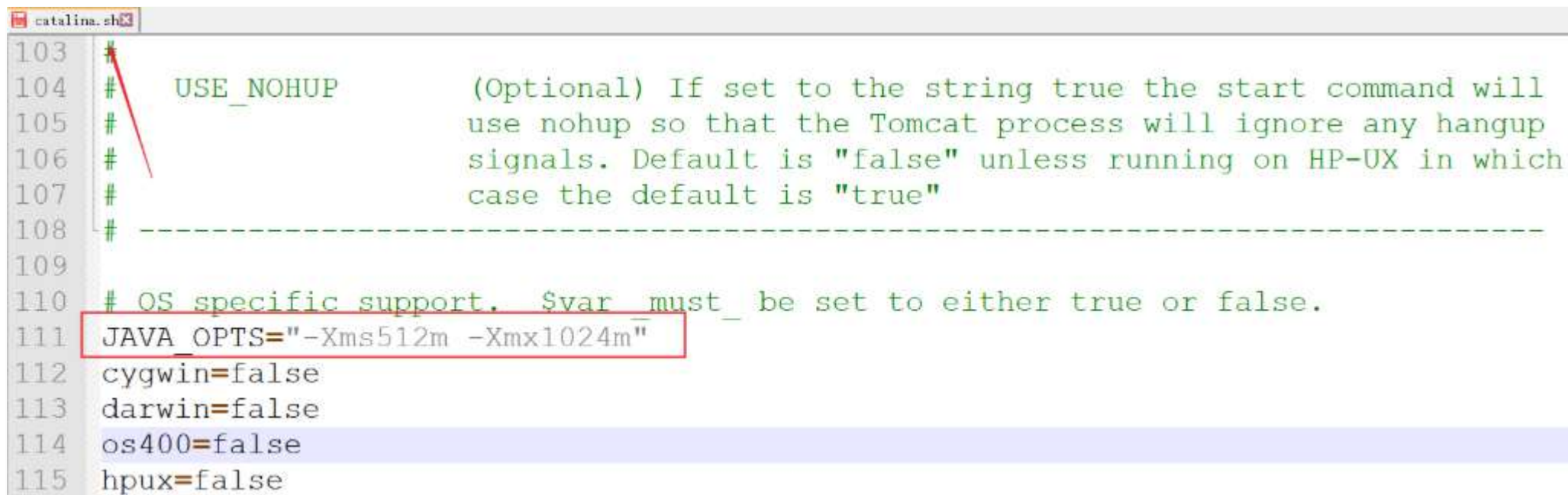
## JVM 调优的参数可以在哪里设置参数值

- war包部署在tomcat中设置
- jar包部署在启动参数设置

## JVM 调优的参数可以在哪里设置参数值

- war包部署在tomcat中设置

修改TOMCAT\_HOME/bin/catalina.sh文件



```
103 #
104 # USE_NOHUP      (Optional) If set to the string true the start command will
105 #                use nohup so that the Tomcat process will ignore any hangup
106 #                signals. Default is "false" unless running on HP-UX in which
107 #                case the default is "true"
108 # -----
109
110 # OS specific support. $var must be set to either true or false.
111 JAVA_OPTS="-Xms512m -Xmx1024m"
112 cygwin=false
113 darwin=false
114 os400=false
115 hpux=false
```

## JVM 调优的参数可以在哪里设置参数值

- jar包部署在启动参数设置

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。

# 总结

## JVM 调优的参数可以在哪里设置参数值

- war包部署在tomcat中设置

修改TOMCAT\_HOME/bin/catalina.sh文件

- jar包部署在启动参数设置

java -Xms512m -Xmx1024m -jar xxxx.jar



# JVM 调优的参数都有哪些？

难易程度： ★★☆☆☆

出现频率： ★★★☆☆

## 用的 JVM 调优的参数都有哪些？

对于JVM调优，主要就是调整年轻代、老年代、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>

- 设置堆空间大小
- 虚拟机栈的设置
- 年轻代中Eden区和两个Survivor区的大小比例
- 年轻代晋升老年代阈值
- 设置垃圾回收收集器

## 用的 JVM 调优的参数都有哪些？

- 设置堆空间大小

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

-Xms: 1024  
-Xms: 1024k  
-Xms: 1024m  
-Xms: 1g

不指定单位默认为字节

指定单位，按照指定的单位设置

堆空间设置多少合适？

- 最大大小的默认值是物理内存的1/4，初始大小是物理内存的1/64
- 堆太小，可能会频繁的导致年轻代和老年代的垃圾回收，会产生stw，暂停用户线程
- 堆内存大肯定是好的，存在风险，假如发生了fullgc,它会扫描整个堆空间，暂停用户线程的时间长
- 设置参考推荐：尽量大，也要考察一下当前计算机其他程序的内存使用情况

## 用的 JVM 调优的参数都有哪些？

- 虚拟机栈的设置

虚拟机栈的设置：**每个线程默认会开启1M的内存**，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
```

## 用的 JVM 调优的参数都有哪些？

- 年轻代中Eden区和两个Survivor区的大小比例

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。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=8, 表示年轻代中的分配比率: survivor:eden = 2:8
```

- 年轻代晋升老年代阈值

```
-XX:MaxTenuringThreshold=threshold
```

- 默认为15
- 取值范围0-15

## 用的 JVM 调优的参数都有哪些？

- 设置垃圾回收收集器

通过增大吞吐量提高系统性能，可以通过设置并行垃圾回收收集器。

```
-XX:+UseParallelGC  
-XX:+UseParallelOldGC
```

```
-XX:+UseG1GC
```



# 总结

用的 JVM 调优的参数都有哪些？

- 设置堆空间大小
- 虚拟机栈的设置
- 年轻代中Eden区和两个Survivor区的大小比例
- 年轻代晋升老年代阈值
- 设置垃圾回收收集器

# 说一下 JVM 调优的工具？

难易程度： ★★☆☆☆

出现频率： ★★☆☆☆



## 说一下 JVM 调优的工具？

- 命令工具

- jps 进程状态信息
- jstack 查看java进程内线程的堆栈信息
- jmap 查看堆转信息
- jhat 堆转储快照分析工具
- jstat JVM统计监测工具

- 可视化工具

- jconsole 用于对jvm的内存，线程，类 的监控
- VisualVM 能够监控线程，内存情况

## 说一下 JVM 调优的工具？

- jps

### 进程状态信息

```
C:\Users\yuhon>jps
27920 Jps
27348 Launcher
28472 Application
6140
```

- jstack

### 查看java进程内线程的堆栈信息

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

```
"Reference Handler" #2 daemon prio=10 os_prio=2 tid=0x000001657fbfe000 nid=0x3274 in Object.wait()
  java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    - waiting on <0x0000000716a06c00> (a java.lang.ref.Reference$Lock)
    at java.lang.Object.wait(Object.java:502)
    at java.lang.ref.Reference.tryHandlePending(Reference.java:191)
    - locked <0x0000000716a06c00> (a java.lang.ref.Reference$Lock)
    at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)

"main" #1 prio=5 os_prio=0 tid=0x000001657a445800 nid=0x41d4 runnable [0x0000001d359ff000]
  java.lang.Thread.State: RUNNABLE
    at com.heima.jvm.Application.main(Application.java:9)
```

## 说一下 JVM 调优的工具？

- jmap

用于生成堆转内存快照、内存使用情况

```
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= <filename> 用于指定快照dump文件的文件名。

### 知识小贴士

它是一个进程或系统在某一给定的时间的快照。比如在进程崩溃时，甚至是任何时候，我们都可以通过工具将系统或某进程的内存备份出来供调试分析用。

dump文件中包含了程序运行的模块信息、线程信息、堆栈调用信息、异常信息等数据，方便系统技术人员进行错误排查。

## 说一下 JVM 调优的工具？

### ● 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。

#### ①：总结垃圾回收统计

```
jstat -gcutil pid
```

```
D:\code\jvm-demo\target\classes\com\heima\jvm>jstat -gcutil 28472
```

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

#### ②：垃圾回收统计

```
jstat -gc pid
```

```
D:\code\jvm-demo\target\classes\com\heima\jvm>jstat -gc 28472
```

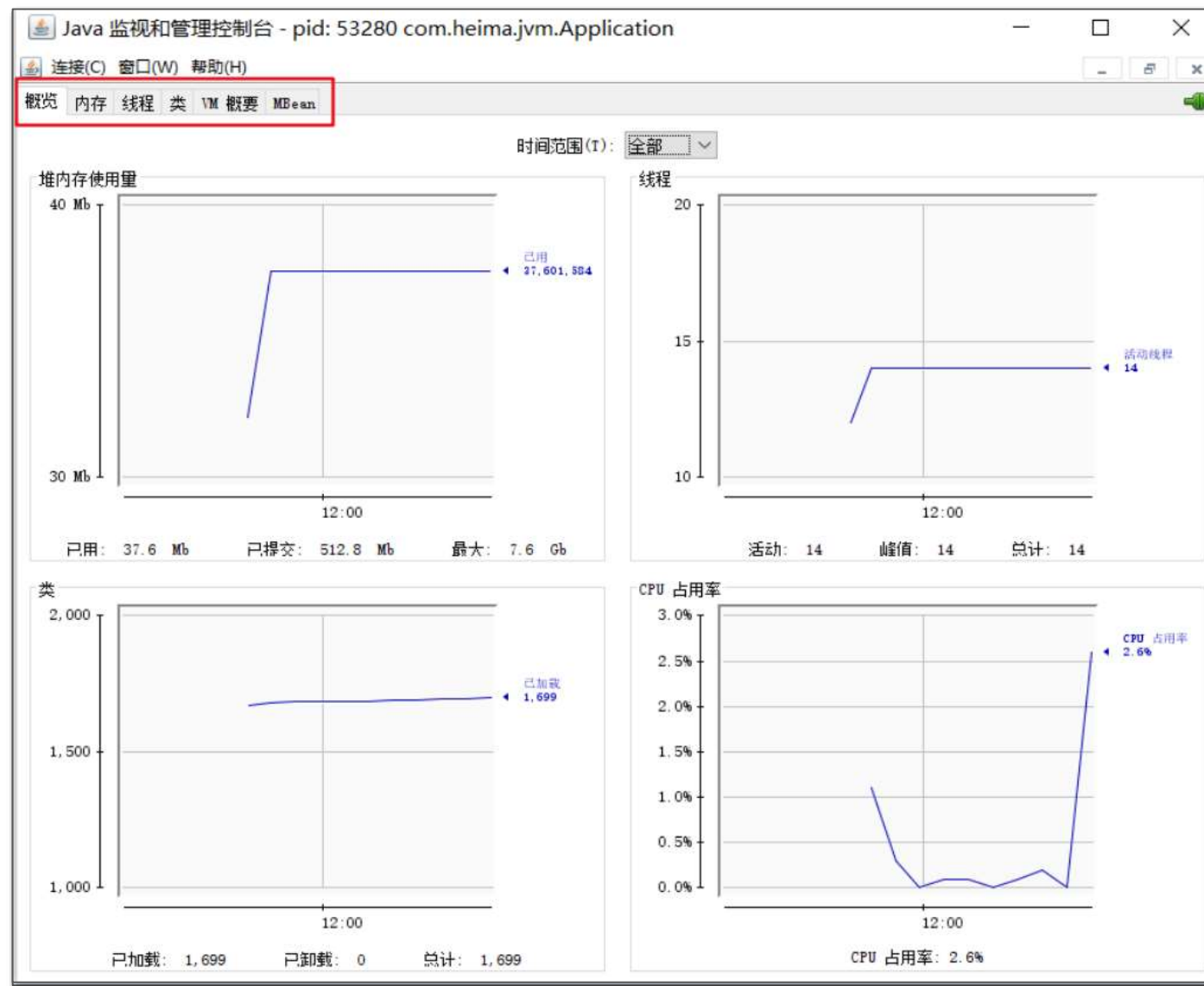
S0C	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU	CCSC	CCSU	YGC	YGCT	FGC	FGCT	GCT
21504.0	21504.0	0.0	0.0	131072.0	10485.8	348160.0	0.0	4480.0	778.5	384.0	76.6	0	0.000	0	0.000	0.000

## 说一下 JVM 调优的工具？

- 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

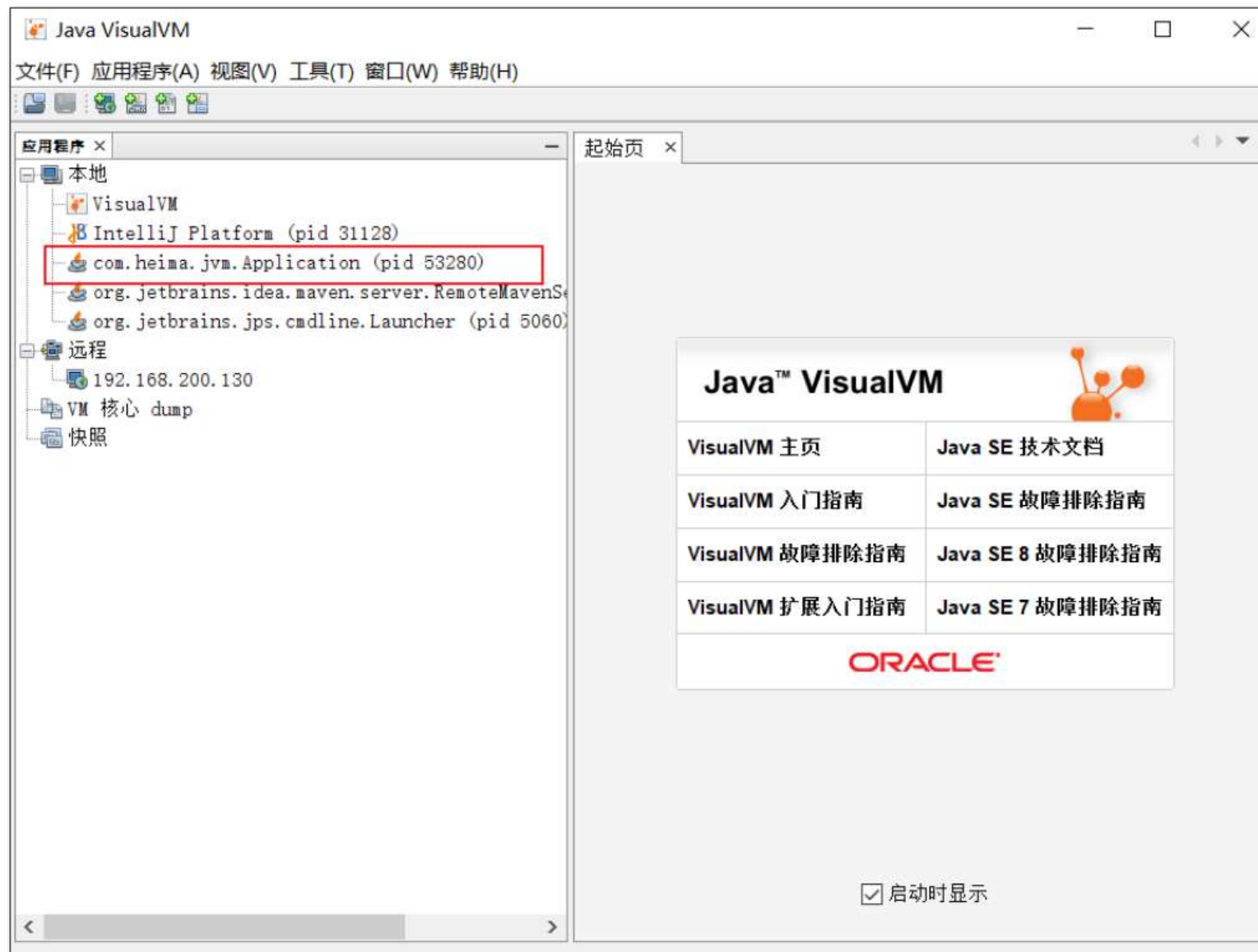


## 说一下 JVM 调优的工具？

- VisualVM

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



## 说一下 JVM 调优的工具？

### 监控程序运行情况

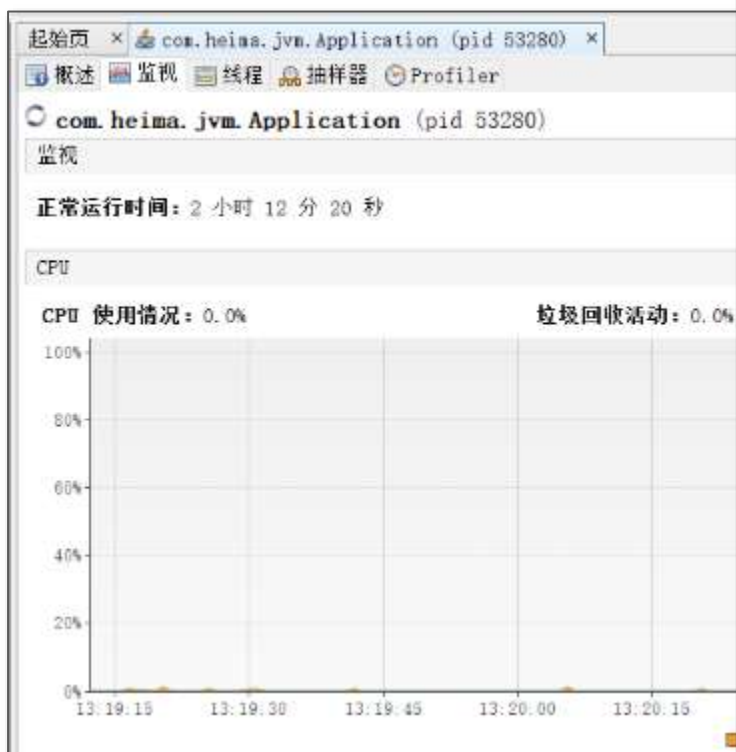




## 说一下 JVM 调优的工具？

查看运行中的dump

Dump文件是进程的内存镜像。可以把程序的执行



The screenshot shows the VisualVM interface for a Java application (pid 53280). The 'Heap Dump' tab is selected, displaying basic information and environment details. The 'Overview' sub-tab is active, showing the following information:

- 基本信息:**
  - 生成的日期: Sun Sep 04 13:21:41 CST 2022
  - 文件: C:\Users\yuhon\AppData\Local\Temp\visualvm.dat\localhost\_53280\heapdump-1662268901283.hprof
  - 文件大小: 4.5 MB
  - 字节总数: 2,713,326
  - 类总数: 1,928
  - 实例总数: 41,417
  - 类加载器: 136
  - 垃圾回收根节点: 1,641
  - 等待结束的暂挂对象数: 0
- 环境:**
  - 操作系统: Windows 10 (10.0)
  - 体系结构: amd64 64bit
  - Java 主目录: C:\Program Files\Java\jdk1.8.0\_321\jre
  - Java 版本: 1.8.0\_321
  - JVM: Java HotSpot(TM) 64-Bit Server VM (25.321-b07, mixed mode)
  - Java 供应商: Oracle Corporation
- 系统属性:**
  - [显示系统属性](#)
- 堆转储上的线程:**
  - "JMX server connection timeout 22" daemon prio=5 tid=22 TIMED\_WAITING
  - at java.lang.Object.wait(Native Method)
  - at com.sun.jmx.remote.internal.ServerCommunicatorAdmin\$Timeout.run(ServerCommunicatorAdmin.java:168)
  - Local Variable: [com.sun.jmx.remote.internal.ServerCommunicatorAdmin\\$Timeout#1](#)
  - at java.lang.Thread.run(Thread.java:750)



# 总结

## 说一下 JVM 调优的工具？

### 命令工具

- jps 进程状态信息
- jstack 查看java进程内线程的堆栈信息
- jmap 查看堆转信息
- jhat 堆转储快照分析工具
- jstat JVM统计监测工具

### 可视化工具

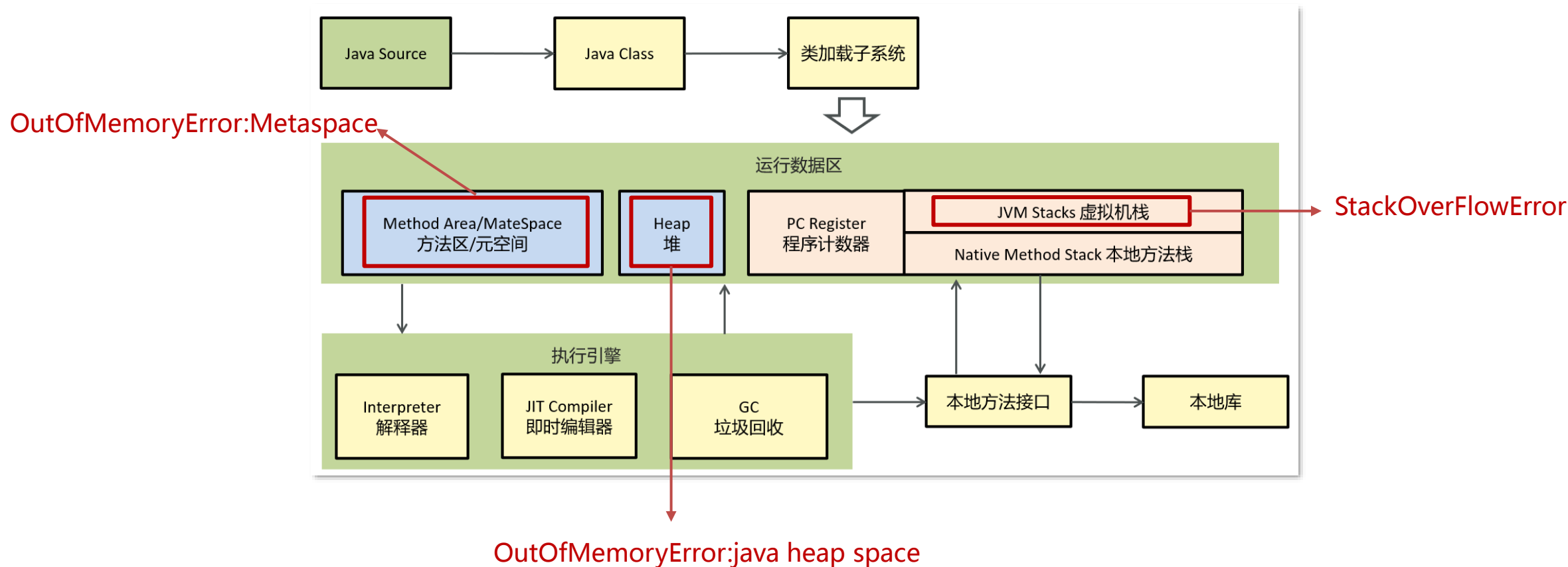
- jconsole 用于对jvm的内存，线程，类 的监控
- VisualVM 能够监控线程，内存情况

# Java内存泄露的排查思路？

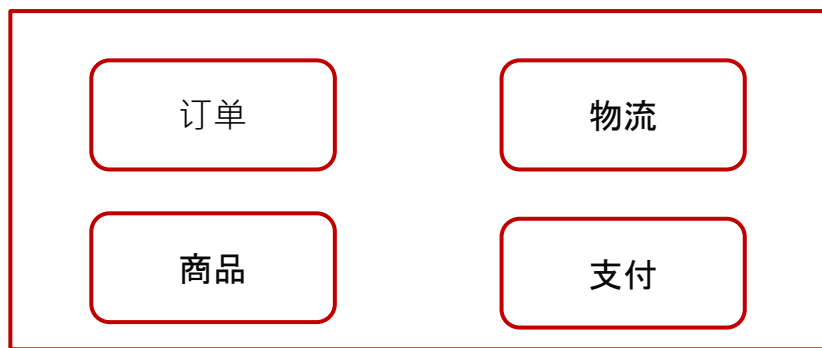
难易程度： ★★★★★

出现频率： ★★★★★

## java内存泄露的排查思路？



## java内存泄露的排查思路？



服务器

- 启动闪退
- 运行一段时间宕机

1. 获取堆内存快照dump
2. VisualVM去分析dump文件
3. 通过查看堆信息的情况，定位内存溢出问题

## java内存泄露的排查思路？

1、通过jmap指定打印他的内存快照dump(Dump文件是进程的内存镜像。可以把程序的执行状态通过调试器保存到dump文件中)

- 使用jmap命令获取运行中程序的dump文件

```
jmap -dump:format=b,file=heap.hprof pid
```

- 使用vm参数获取dump文件

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

```
-XX:+HeapDumpOnOutOfMemoryError  
-XX:HeapDumpPath=/home/app/dumps/
```

## java内存泄露的排查思路？

2、通过工具，VisualVM去分析dump文件，VisualVM可以加载离线的dump文件

文件-->装入--->选择dump文件即可查看堆快照信息



## java内存泄露的排查思路？

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



The screenshot shows the JVisualVM interface for analyzing a heap dump file named `java_pid53616.hprof`. The 'Overview' tab is selected, displaying basic information about the dump. A red box highlights the 'main' thread's stack trace, which is the thread that caused the `OutOfMemoryError`.

**堆 Dump**  
[heapdump] java\_pid53616.hprof 离线dump文件

堆 Dump  
← → 概要 类 实例数 OQL 控制台

概述

基本信息:

- 生成的日期: Sat Aug 27 11:50:54 CST 2022
- 文件: D:\develop\java\_pid53616.hprof
- 文件大小: 4,097.8 MB
- 字节总数: 4,295,963,250
- 类总数: 698
- 实例总数: 13,335
- 类加载器: 3
- 垃圾回收节点: 673
- 等待结束的暂挂对象数: 0

在出现 `OutOfMemoryError` 异常错误时进行了堆转储  
导致 `OutOfMemoryError` 异常错误的线程: `main`

堆转储上的线程:

- "Signal Dispatcher" daemon prio=9 tid=4 RUNNABLE
- "main" prio=5 tid=1 RUNNABLE
  - at java.lang.OutOfMemoryError.<init>(OutOfMemoryError.java:48)
  - at java.util.Arrays.copyOfRange(Arrays.java:3664)
  - Local Variable: `char[]=4494`
  - at java.lang.String.<init>(String.java:207)
  - at java.lang.StringBuilder.toString(StringBuilder.java:413)
  - at com.heimajava.Application.main(Application.java:17) ←
  - Local Variable: `java.lang.String[]=15`
  - Local Variable: `java.util.ArrayList=6`
  - Local Variable: `java.lang.String=180`

## java内存泄露的排查思路？

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



[heapdump] java\_pid53616.hprof 离线dump文件

堆 Dump

← → 概要 类 实例数 OQL 控制台

概述

基本信息:

生成的日期: Sat Aug 27 11:50:54 CST 2022  
文件: D:\develop\java\_pid53616.hprof  
文件大小: 4,097.8 MB

字节总数: 4,295,963,250  
类总数: 698  
实例总数: 13,335  
类加载器: 3  
垃圾回收根节点: 673  
等待结束的暂挂对象数: 0

在出现 OutOfMemoryError 异常错误时进行了堆转储  
导致 OutOfMemoryError 异常错误的线程: [main](#)

堆转储上的线程:

"Signal Dispatcher" daemon prio=9 tid=4 RUNNABLE

"main" prio=5 tid=1 RUNNABLE  
at java.lang.OutOfMemoryError.<init>(OutOfMemoryError.java:48)  
at java.util.Arrays.copyOfRange(Arrays.java:3664)  
Local Variable: [char\[\]#4494](#)  
at java.lang.String.<init>(String.java:207)  
at java.lang.StringBuilder.toString(StringBuilder.java:413)  
at com.heima.jvm.Application.main(Application.java:17) ←  
Local Variable: [java.lang.String\[\]#15](#)  
Local Variable: [java.util.ArrayList#6](#)  
Local Variable: [java.lang.String#186](#)

4、找到对应的代码，通过阅读上下文的情况，进行修复即可





# 总结

## java内存泄露的排查思路？

内存泄漏通常是指堆内存，通常是指一些大对象不被回收的情况

- 1、通过jmap或设置jvm参数获取堆内存快照dump
- 2、通过工具， VisualVM去分析dump文件， VisualVM可以加载离线的dump文件
- 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

# CPU飙高排查方案与思路？

难易程度：★★★★☆

出现频率：★★★★☆

## CPU飙高排查方案与思路？

### 1.使用top命令查看占用cpu的情况

```
top
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
40940	root	20	0	2243512	28436	11604	S	88.3	2.8	0:32.06	java
1442	root	20	0	159268	6100	4304	S	0.7	0.6	0:08.58	sshd
1895	root	20	0	159164	6032	4304	S	0.7	0.6	0:07.77	sshd
24446	root	20	0	159164	6020	4312	R	0.3	0.6	0:02.42	sshd

### 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：40940

## CPU飙高排查方案与思路？

### 3.查看进程中的线程信息

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

```
[root@myhbase ~]# ps H -eo pid,tid,%cpu | grep 40940
40940 40940 0.0
40940 40941 0.0
40940 40942 0.0
40940 40943 0.0
40940 40944 0.0
40940 40945 0.0
40940 40946 0.0
40940 40947 0.0
40940 40948 0.0
40940 40949 0.0
40940 40950 87.9
40940 40955 0.0
40940 40958 0.0
```

通过以上分析，在进程40940中的线程40950占用cpu较高

## CPU飙高排查方案与思路？

4.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

jstack 40940 此处是进程id

```
"thread1" #8 prio=5 os_prio=0 tid=0x00007fcf580f5000 nid=0x9ff6 runnable [0x00007fcf355b7000]
  java.lang.Thread.State: RUNNABLE
    at Application.lambda$main$0(Application.java:9)
    at Application$$Lambda$1/531885035.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:748)

"Service Thread" #7 daemon prio=9 os_prio=0 tid=0x00007fcf580b3800 nid=0x9ff4 runnable [0x0000000000000000]
  java.lang.Thread.State: RUNNABLE

"C1 CompilerThread1" #6 daemon prio=9 os_prio=0 tid=0x00007fcf580b0800 nid=0x9ff3 waiting on condition [0x0000000000000000]
  java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" #5 daemon prio=9 os_prio=0 tid=0x00007fcf580ae800 nid=0x9ff2 waiting on condition [0x0000000000000000]
  java.lang.Thread.State: RUNNABLE
```

十六进制

40940	40949	0.0
40940	40950	87.9
40940	40955	0.0
40940	40958	0.0

十进制

十进制转换为十六进制

printf "%x\n" 40955

```
[root@myhbase ~]# printf "%x\n" 40955
9ffb
```



# 总结

## CPU飙高排查方案与思路？

- 1.使用top命令查看占用cpu的情况
- 2.通过top命令查看后，可以查看是哪一个进程占用cpu较高
- 3.使用ps命令查看进程中的线程信息
- 4.使用jstack命令查看进程中哪些线程出现了问题，最终定位问题



传智教育旗下高端IT教育品牌