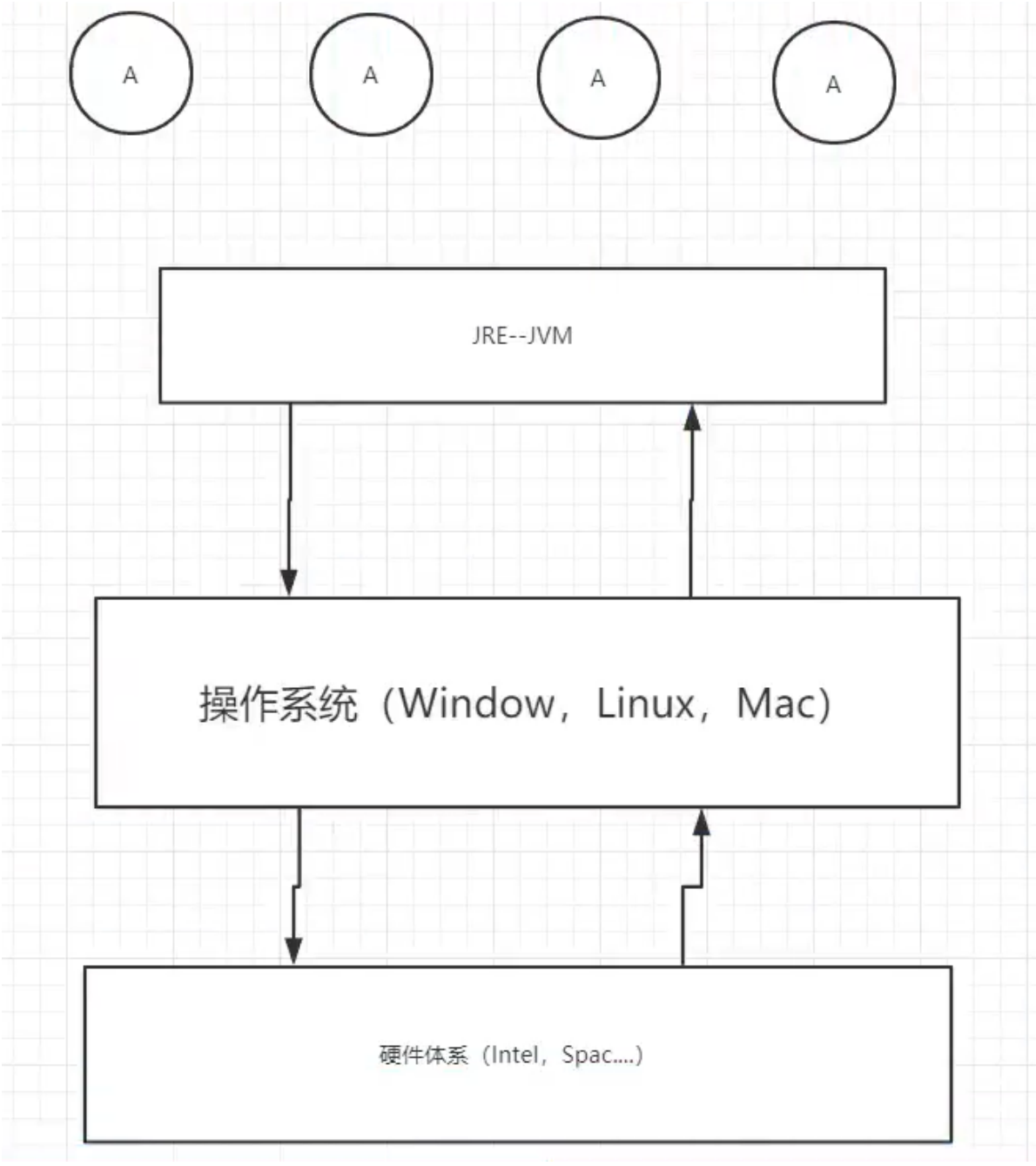
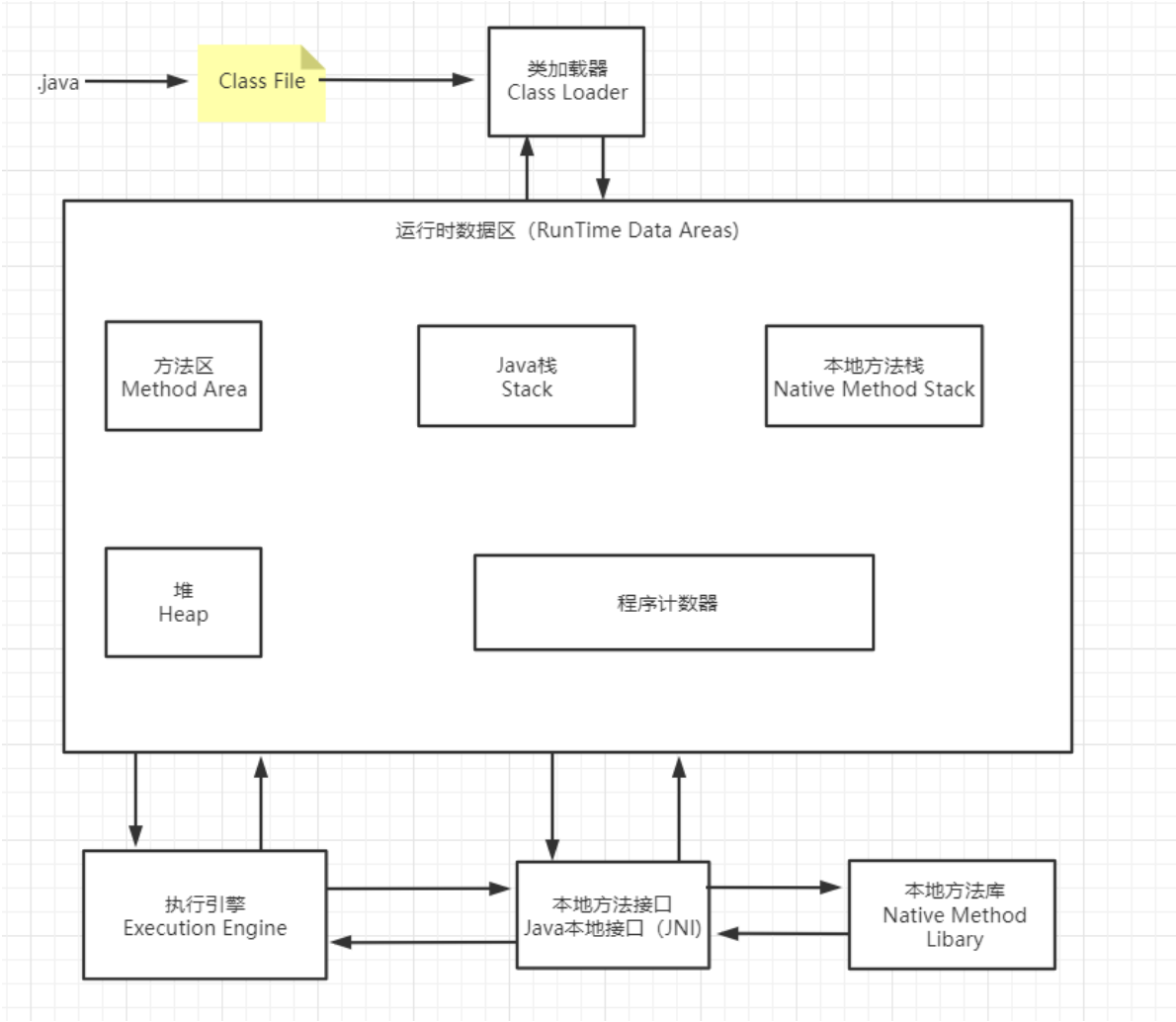
JVM 探究

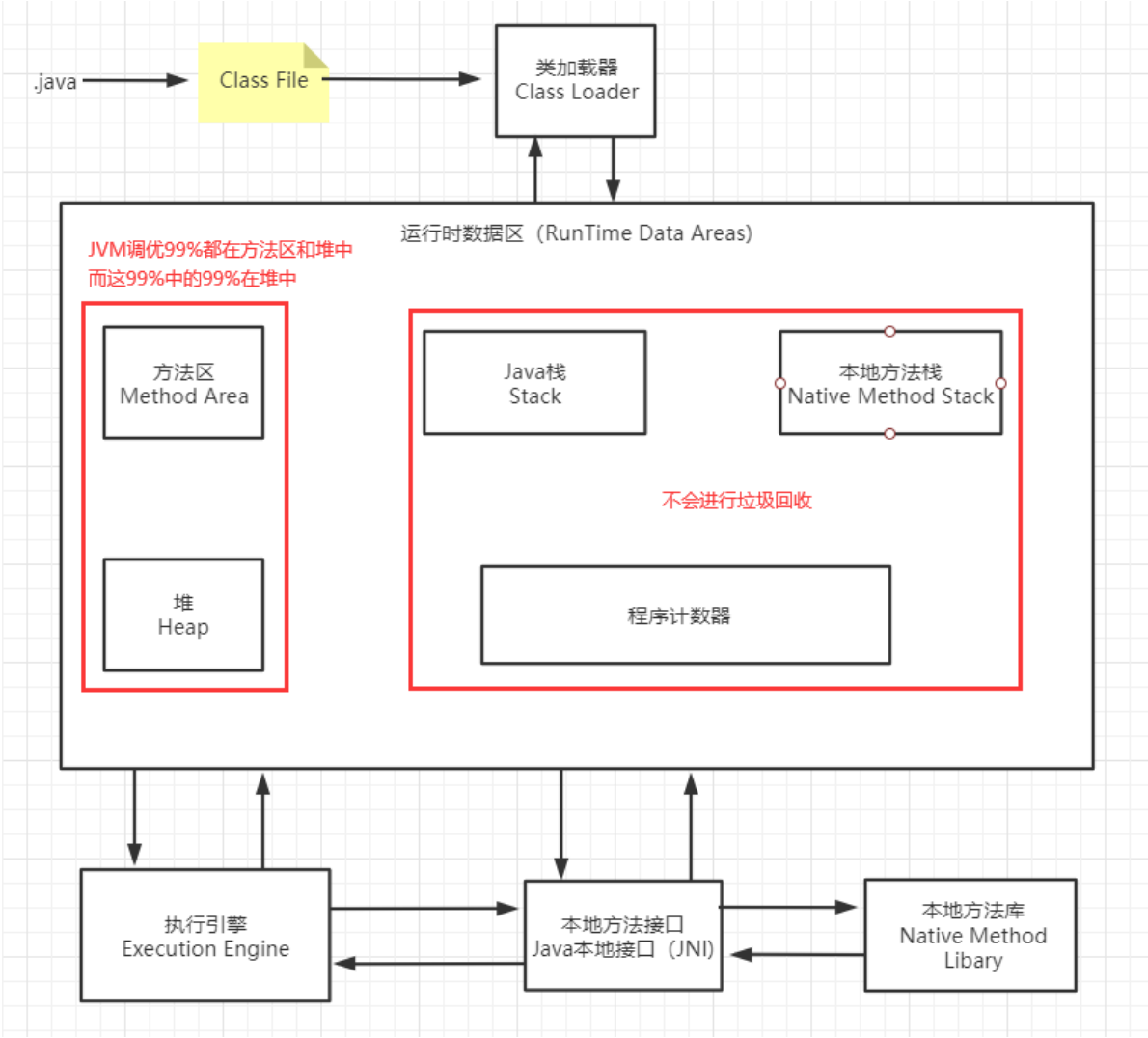
- 请你谈谈你对JVM的理解？ Java8虚拟机和之前的变化更新?
- 什么是 OOM ，什么是栈溢出 StackOverFlowError？ 怎么分析?
- JVM的常用调优参数有哪些?
- 内存快照如何抓取，怎么分析Dump文件?
- 谈谈你对 JVM 中，类加载器的认识?

1. JVM 的位置



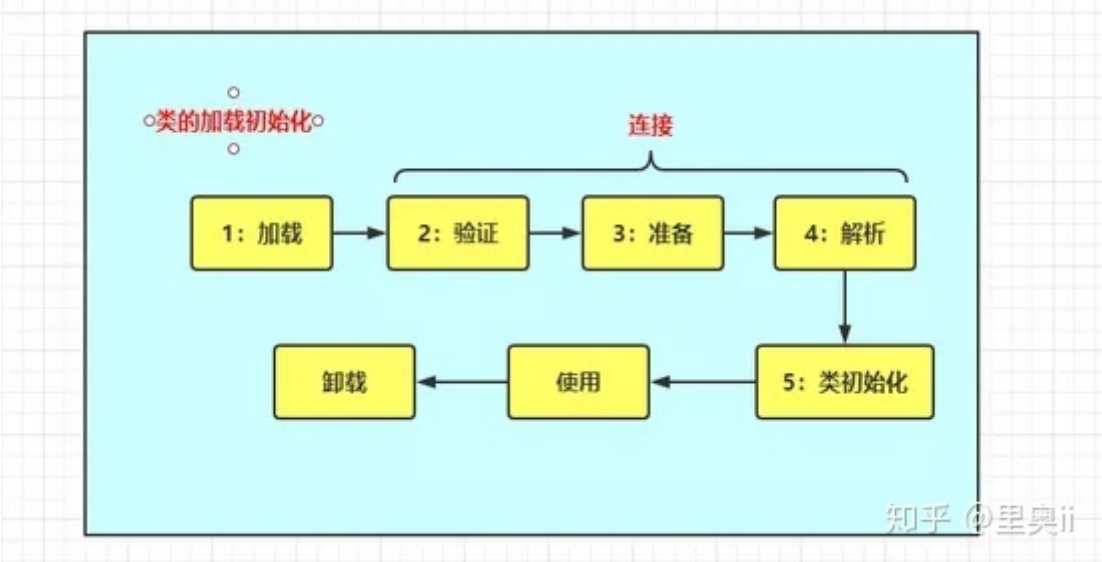
2. JVM 的体系结构


.java
Class File
类加载器
Class Loader
运行时数据区（RunTime Data Areas)
方法区
Method Area
Java栈
Stack
本地方法栈
Native Method Stack
堆
Heap
程序计数器
执行引擎
Execution Engine
本地方法接口
Java本地接口（JNI)
本地方法库
Native Method
Libary

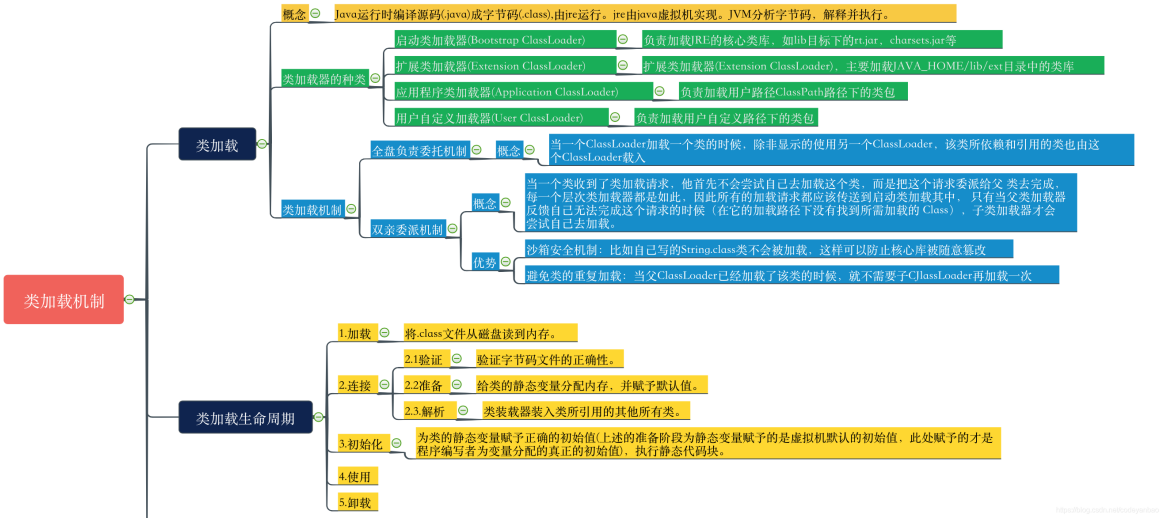


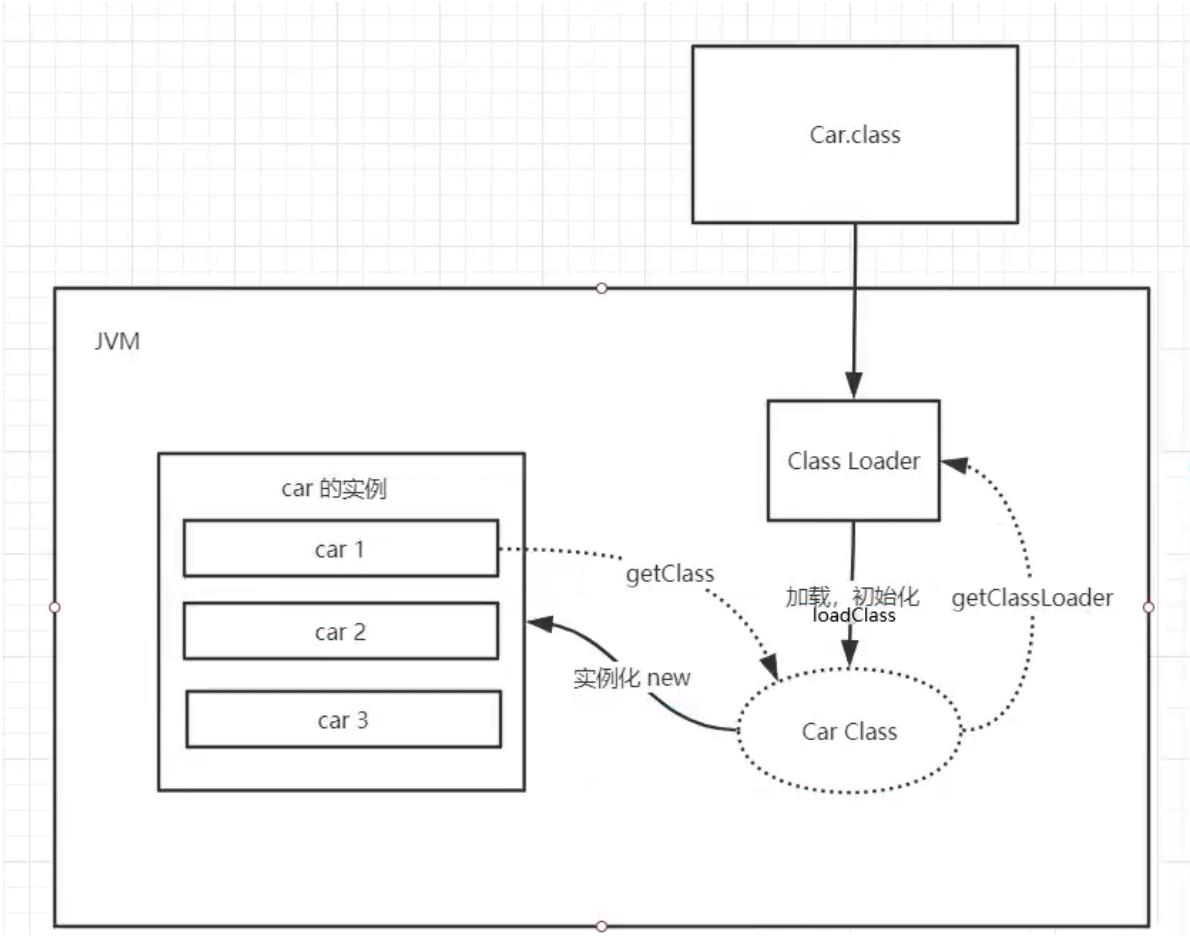
3. 类加载

一个.java文件在编译后会形成相应的一个或多个Class文件（若一个类中含有内部类，则编译后会产生多个Class文件），但这些Class文件中描述的各种信息，最终都需要加载到虚拟机中之后才能被运行和使用。事实上，虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验，转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型的过程就是虚拟机的 **类加载机制**。



类加载器作用：加载 Class 文件





类加载器的级别：

1. BootstrapClassLoader（启动类/根加载器）

   `C++` 编写，加载 `java` 核心库 `java.*`,构造 `ExtClassLoader` 和 `AppClassLoader`。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作

2. ExtClassLoader（扩展类加载器）

`java`编写，加载扩展库，如`classpath`中的`jre`，`javax.*`或者`java.ext.dir`指定位置中的类，开发者可以直接使用标准扩展类加载器

3. AppClassLoader（应用程序/系统类加载器）

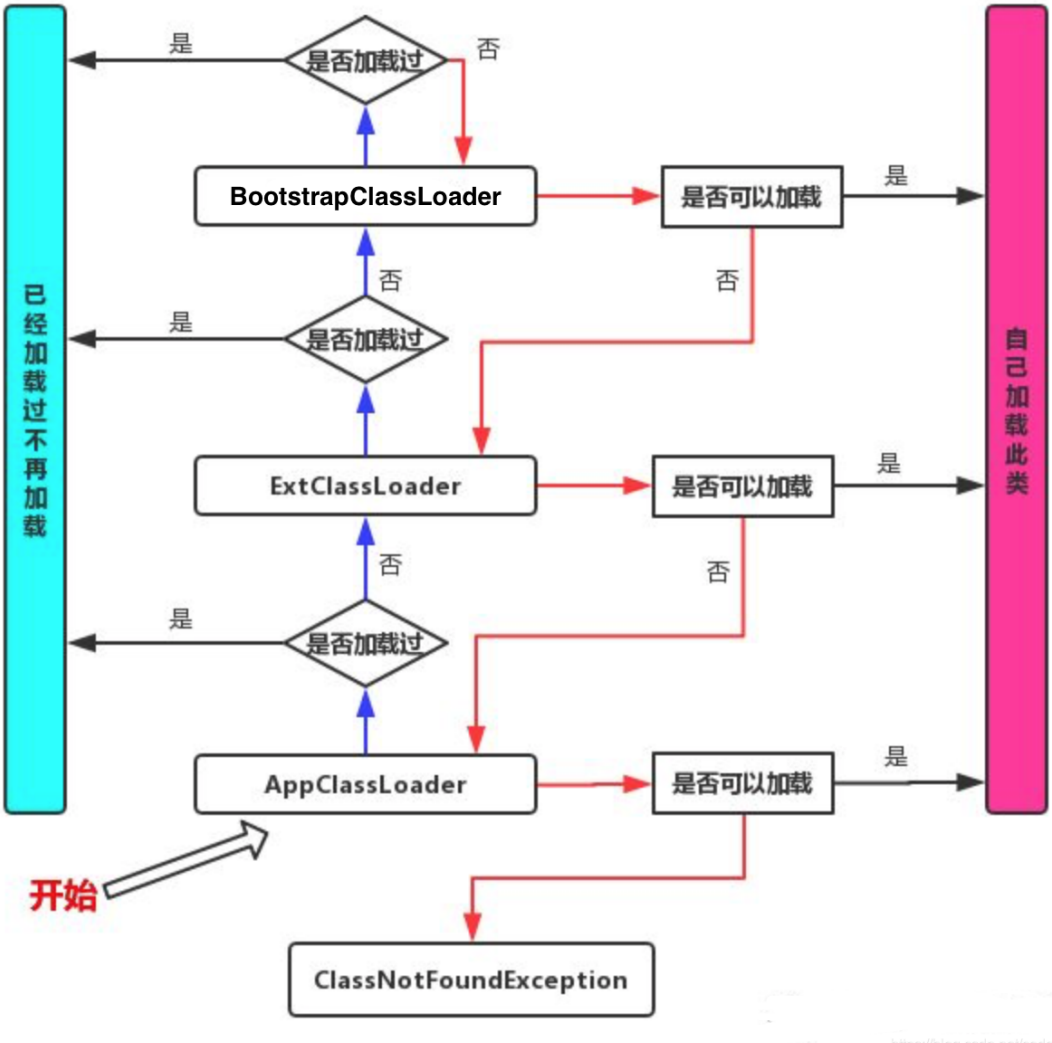
   java`编写，加载程序所在的目录，如`user.dir`所在的位置的`class

4. CustomClassLoader（用户自定义类加载器）

   `java`编写,用户自定义的类加载器,可加载指定路径的`class`文件

4. 双亲委派机制

从最内层JVM自带类加载器开始加载，外层恶意同名类得不到加载从而无法使用；



注意：应该先看CustomClassLoader有没有加载过，没有则进入ApplicationClassLoader

源码：

```
public Class<?> loadClass(String name) throws ClassNotFoundException {
    return loadClass(name, false);
```

```
}
//              -----??-----
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
        // 首先，检查是否已经被类加载器加载过
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            try {
                // 存在父加载器，递归的交由父加载器
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else {
                    // 直到最上面的Bootstrap类加载器
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }

            if (c == null) {
                // If still not found, then invoke findClass in order
                // to find the class.
                //如果BootstrapClassLoader 仍然没有加载过，则递归回来，尝试自己去加载
class
                c = findClass(name);
            }
        }
        return c;
}
```

5. 沙箱安全机制

沙箱安全：防止恶意代码污染Java源代码

例：比如我定义了一个类名为String所在包为java.lang，因为这个类本来是属于jdk的，如果没有沙箱安全机制的话，这个类将会污染到我所有的String,但是由于沙箱安全机制，所以就委托顶层的bootstrap加载器查找这个类，如果没有的话就委托extsion,extsion没有就到aapclassloader，但是由于String就是jdk的源代码，所以在bootstrap那里就加载到了，先找到先使用，所以就使用bootstrap里面的String,后面的一概不能使用，这就保证了不被恶意代码污染

详细

6. Native

native：凡是带了 native 关键字的，说明Java的作用范围打不到了，会去调用底层C语言的库！

- 调用本地方法本地接口 JNI
- JNI作用：扩展Java的使用，融合不同的编程语言为Java所用！ 最初：C、C++
- Java单身的时候 C、C++ 横行，想要立足，必须要有调用C、C++的程序
- 它在内存区域中专门开辟了一块标记区域：Native Method Stack，登记 Native 方法
- 在最终执行的时候，通过 JNI 加载本地方法库中的方法
- java程序驱动打印机，管理系统
- 调用其他接口：Socket、WebService、Http~

7. PC寄存器

程序计数器：Program Counter Register

每个线程都有一个程序计数器，是线程私有的，就是一个指针，指向方法区中的方法字节码（用来存储指向一条指令的地址，也即将要执行的指令代码），在执行引擎读取下一条指令，是一个非常小的内存空间，几乎可以忽略不计

8. 方法区

Method Area 方法区

- 又被称为静态区，它跟堆一样，被所有的线程共享，方法区包含所有的 class 信息 和 static修饰的变量。
- 方法区中包含的都是整个程序中永远唯一的元素，如：class、static变量。

方法区被所有线程共享，所有字段和方法字节码，以及一些特殊方法，如构造函数，接口代码也在此定义，简单说，所有定义的方法的信息都保存在该区域，**此区域属于共享区间**；

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

9. 栈

栈：数据结构

程序 = 数据结构 + 算法：持续学习~

程序 = 框架 + 业务逻辑：吃饭~

栈：先进后出

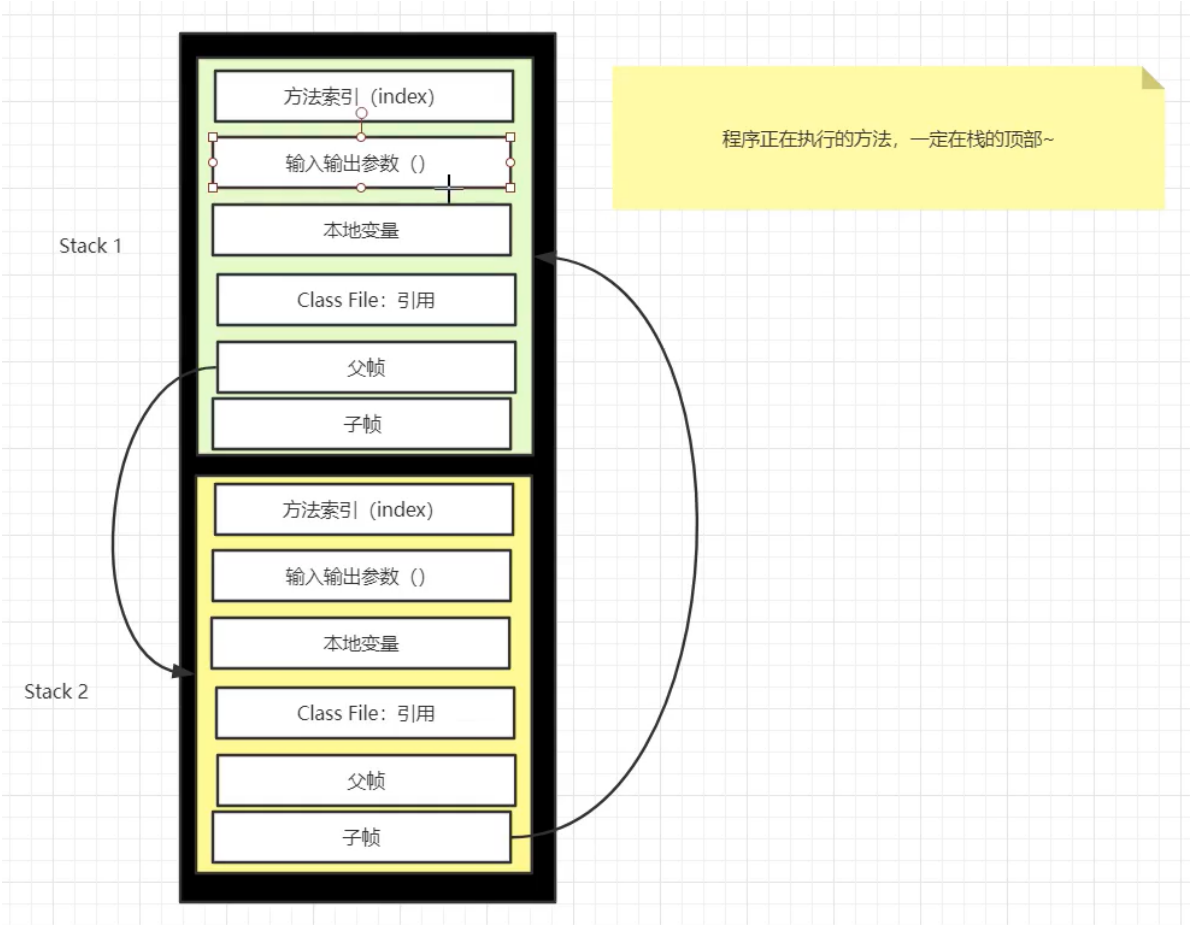
队列：先进先出（FIFO）

栈：栈内存，主管程序的运行，生命周期和线程同步；线程结束，栈内存也就释放。不存在垃圾回收

栈：8大基本类型 + 对象引用 + 实例的方法

栈运行原理：栈帧

栈满了：StackOverflowError



10. 三种 JVM

- Sun公司 HotSpot `Java HotSpot(TM) 64-Bit Server VM (build 25.241-b07, mixed mode)`
- BEA `JRockit`
- IBM `J9 VM`

学习基本都是：`HotSpot`

11. 堆

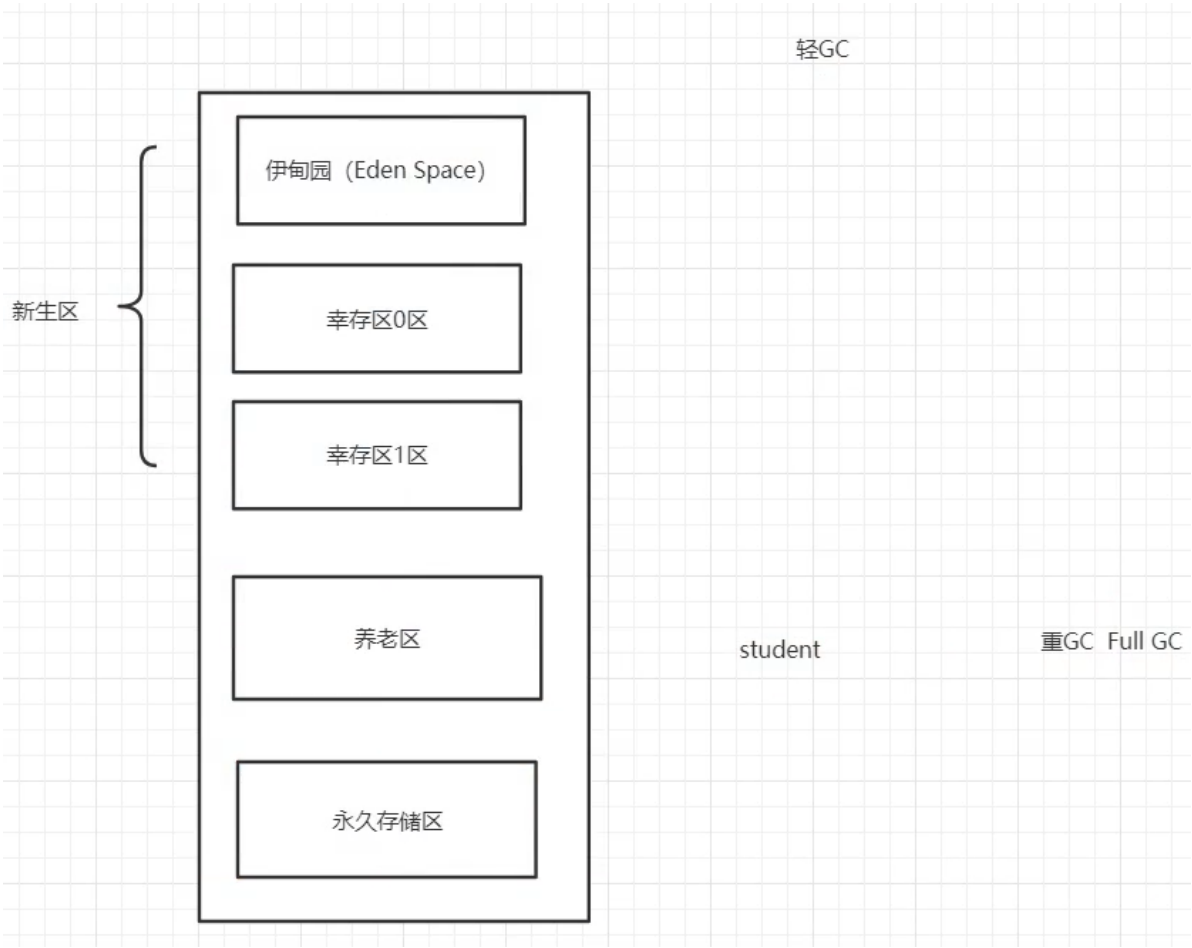
Heap，一个JVM只有一个堆内存，堆内存的大小是可以调节的。

类加载器读取了类文件后，一般会把什么东西放到堆中?

　类，方法，常量，变量~，保存我们所有引用类型的真实对象；

堆内存中还要细分为三个区域：

- 新生区　（伊甸园区） Young/New
- 养老区 Old
- 永久区 Perm



GC 垃圾回收主要在：伊甸园区和养老区

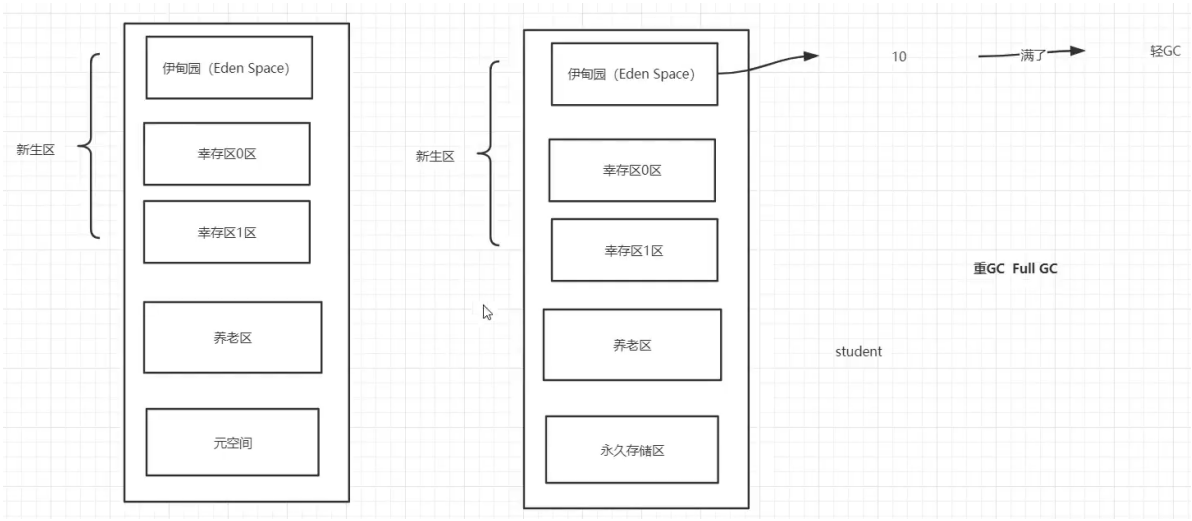
假设内存满了，OOM，堆内存不够！ java.lang.OutOfMemoryError: Java heap space

在 JDK8以后，永久存储区改了个名字（元空间）

11.1. 新生代（YoungGen）

- 类：诞生和成长的地方，甚至死亡
- 伊甸园区，所有的对象都是在 伊甸园区 new 出来的！
- 幸存者区（0，1）

11.2.老年代（OldGen）



真相：经过研究，99%的对象都是临时对象！

11.3. 元空间（Metaspace）

这个区域常驻内存。用来存放JDK自身携带的Class对象、Interface元数据，存储的是Java运行时的一些环境或类信息~，这个区域不存在垃圾回收！关闭VM就会释放这个区域的内存

一个启动类，加载了大量的第三方jar包。Tomcat部署了太多的应用。大量动态生成的反射类，不断被加载，直到内存满。就会出现OOM。

- jdk1.6之前：永久代，常量池在方法区
- jdk1.7　　：永久代，常量池在堆
- jdk1.8　　：无永久代，常量池在元空间

元空间逻辑上存在，物理上不存在：YoungGen+OldGen=totalMemory

15. 堆内存调优

在一个项目中，突然出现了OOM故障，那么该如何排除~研究为什么出错

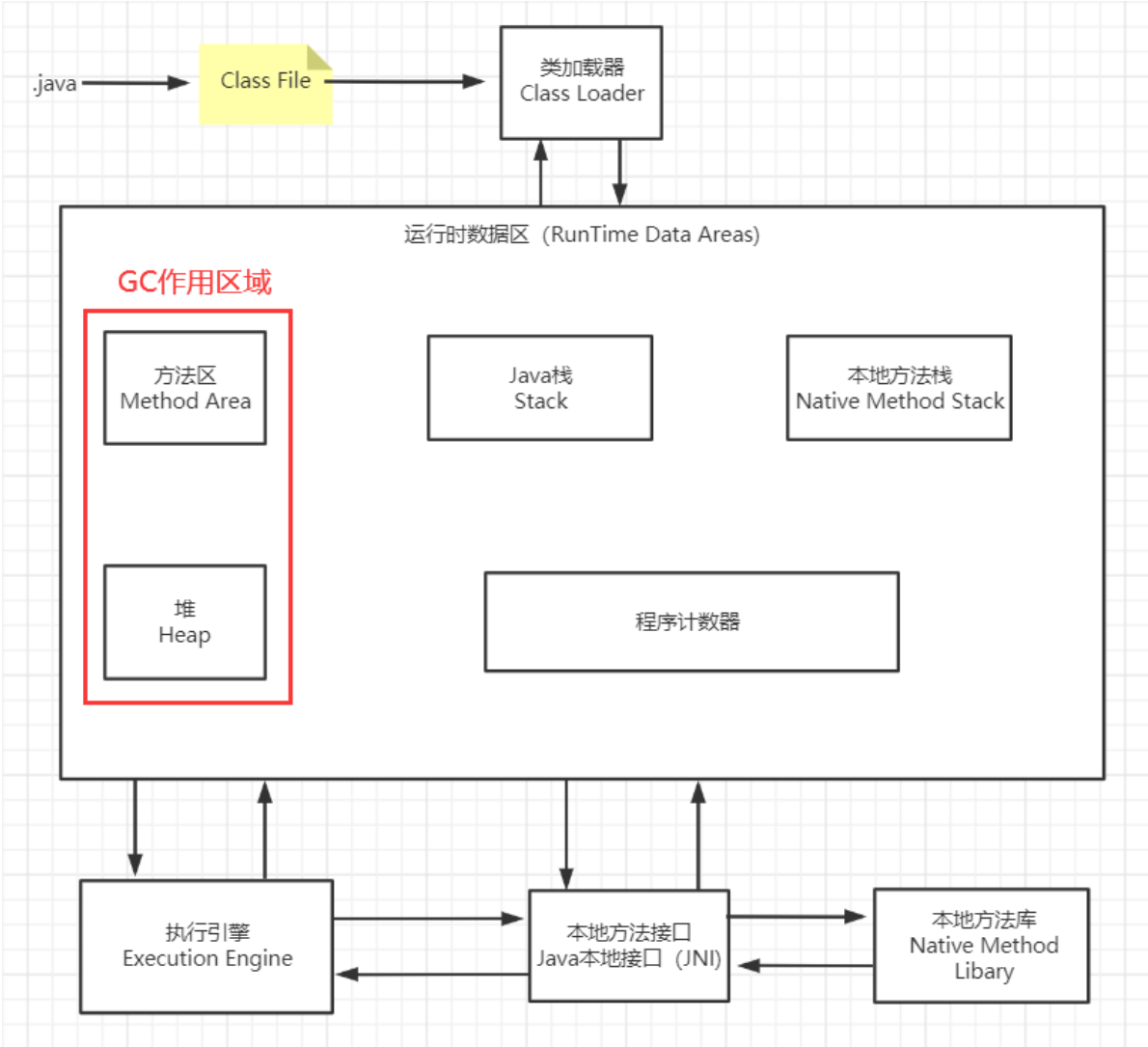
- 能够看到代码第几行出错：内存快照分析工具，MAT、Jprofiler
- Debug，一行行分析代码！

MAT、Jprofiler 作用：

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

```
/*
-Xms 设置初始化内存分配大小  默认1/64
-Xmx 设置最大分配内存    默认1/4
-XX:+PrintGCDetails //打印GC垃圾回收信息
-XX:+HeapDumpOnOutOfMemoryError //oom DUMP
 */
//-Xms1m -Xmx8m -XX:+HeapDumpOnOutOfMemoryError
```

12. GC



JVM在进行GC时，并不是对 新生代 和 老年代 统一回收，大部分时候，回收都是新生代

- 新生代（伊甸园区，幸存区（from，to））
- 老年代

GC两种类：轻GC（普通的GC），重GC（全局GC）

题目：

- JVM 的内存模型和分区~详细到每个区放什么?
- 堆里面的分区有哪些？新生代（Eden，from，to），老年代，说说他们的特点!
- GC的算法有哪些？标记清除法，标记压缩法，复制算法，分代整理法，引用计数法，怎么用?
- 轻GC和重GC分别在什么时候发生?

12.1常用算法

前提：Java运行时内存区，划分为线程私有区和线程共享区：

线程私有区：

- 程序计数器：记录正在执行的虚拟机字节码的地址；
- 虚拟机栈：方法执行的内存区，每个方法执行时会在虚拟机栈中创建栈帧；
- 本地方法栈：虚拟机的Native方法执行的内存区；

线程共享区：

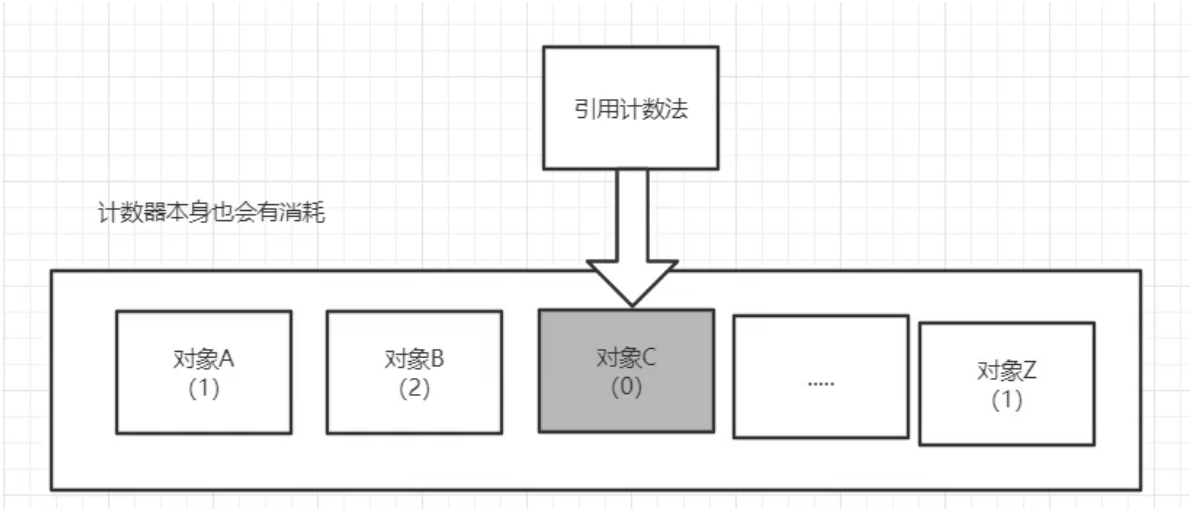
- java堆：对象分配内存的区域，这是垃圾回收的主战场；
- 方法区：存放类信息、常量、静态变量、编译器编译后的代码等数据，另外还有一个常量池。当然垃圾回收也会在这个区域工作。

目前虚拟机基本都是采用可达性算法。如Sun的Hotspot。

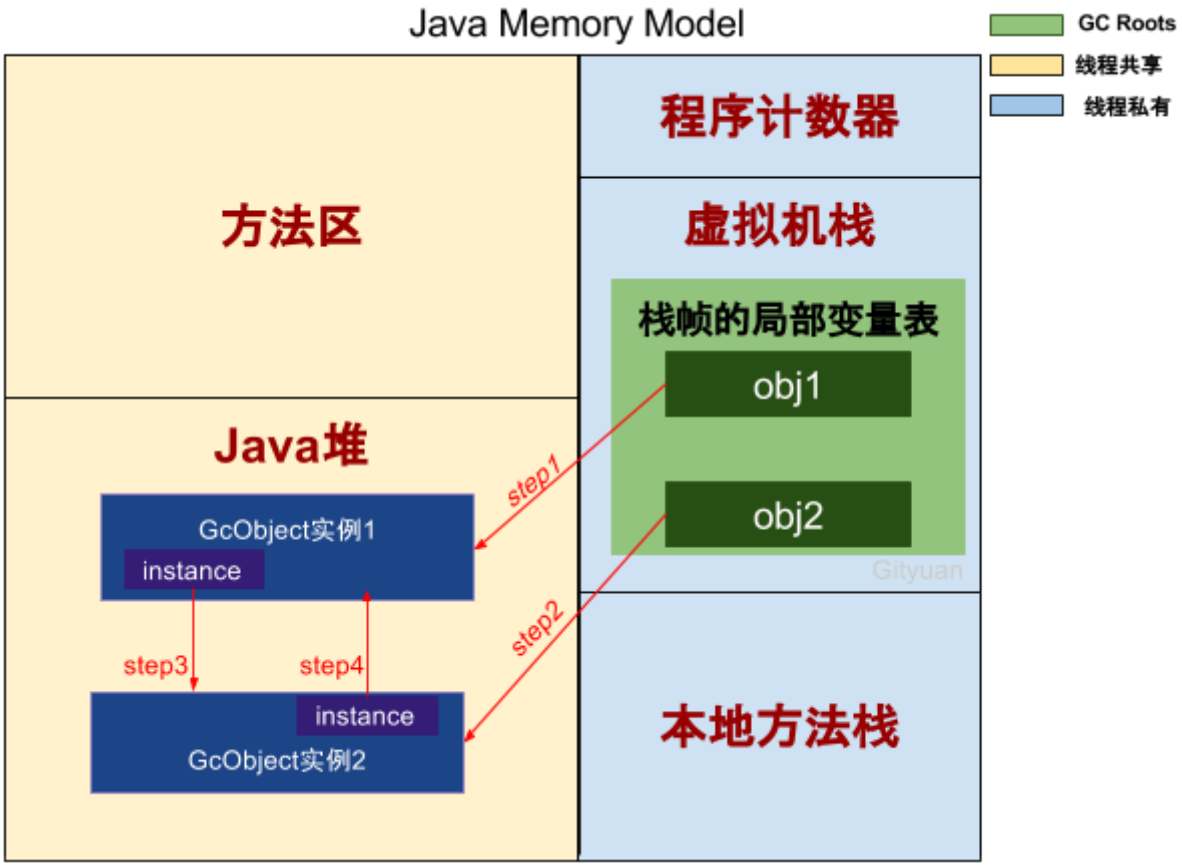
对象存活判断：*引用计数法，可达性分析算法*

引用计数法（reference-counting）

每个对象有一个引用计数器，当对象被引用一次则计数器加1，当对象引用失效一次则计数器减1，对于计数器为0得对象意味着是垃圾对象，可以被GC回收。



缺点：循环引用问题。



解决：可以使用 **弱引用** 来打破某些已知的循环引用。

扩展：强引用、软引用、弱引用、虚引用。

- 强引用（Strong Reference）：不会被GC回收。内存不足时抛OOM也不会回收此类对象。
- 软引用（Soft Reference）：内存不足时才会被GC回收。可以用来解决OOM问题。软引用可以和一个引用队列（ReferenceQueue）联合使用，软引用被回收时加入与之关联的引用队列中。java.lang.ref.SoftReference类
- 弱引用（Weak Reference）：当JVM进行垃圾回收时，无论内存充足与否，都会回收被弱引用关联的对象。可以和一个引用队列（ReferenceQueue）联合使用，弱引用被回收时加入与之关联的引用队列中。java.lang.ref.WeakReference类
- 虚引用（Phantom Reference）：不影响对象生命周期。任何时候都可能被GC回收。**虚引用必须和引用队列关联使用。**java.lang.ref.PhantomReference类

引用类型	被回收时间	用途	生存时间
强引用	从来不会	对象的一般状态	JVM停止运行时
软引用	内存不足时	对象缓存	内存不足时
弱引用	jvm垃圾回收时	对象缓存	gc运行后
虚引用	未知	未知	未知

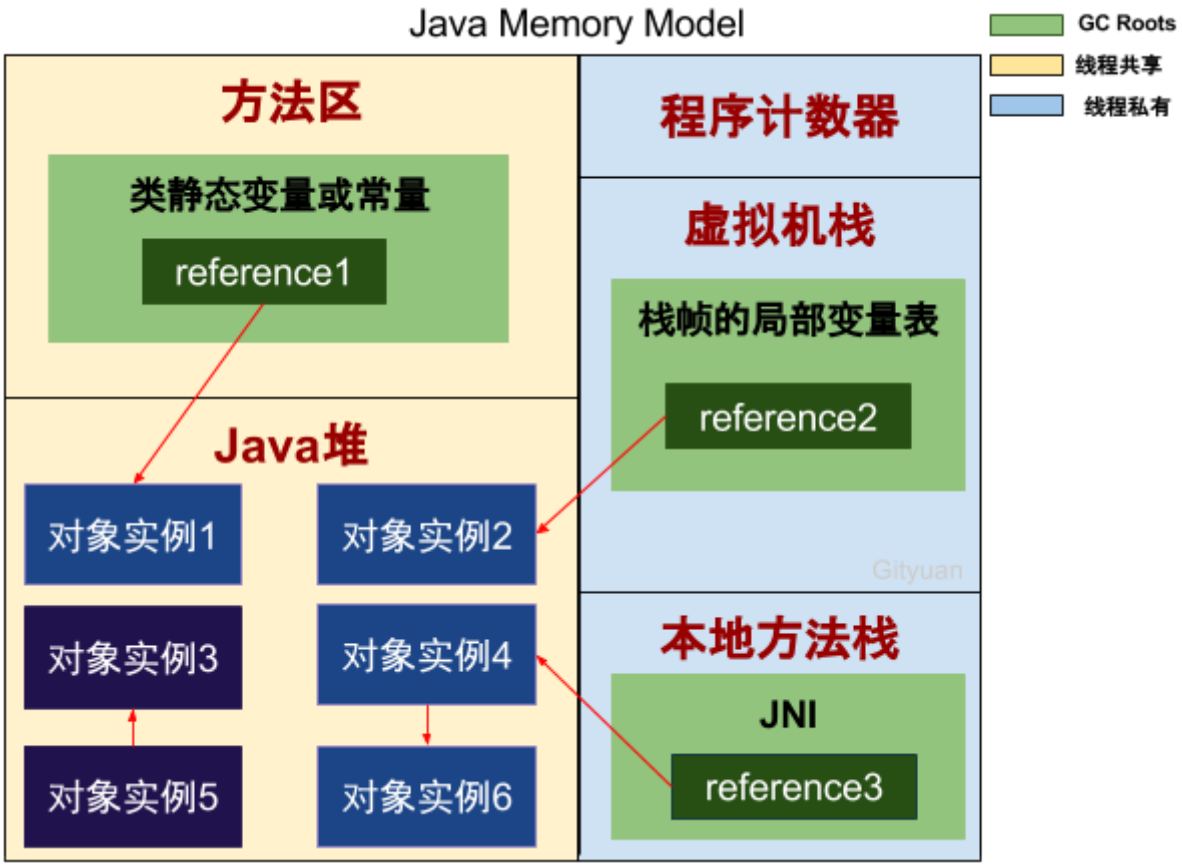
可达性分析算法（GC Roots Tracing）

从GC Roots作为起点开始搜索，那么整个连通图中的对象便都是存活对象，对于GC Roots无法到达的对象便成了垃圾回收对象，随时可被GC回收。

可以作为GC Roots的对象：

- 虚拟机栈的栈帧的局部变量表所引用的对象；
- 本地方法栈的 JNI 所引用的对象；
- 方法区的静态变量和常量所引用的对象；

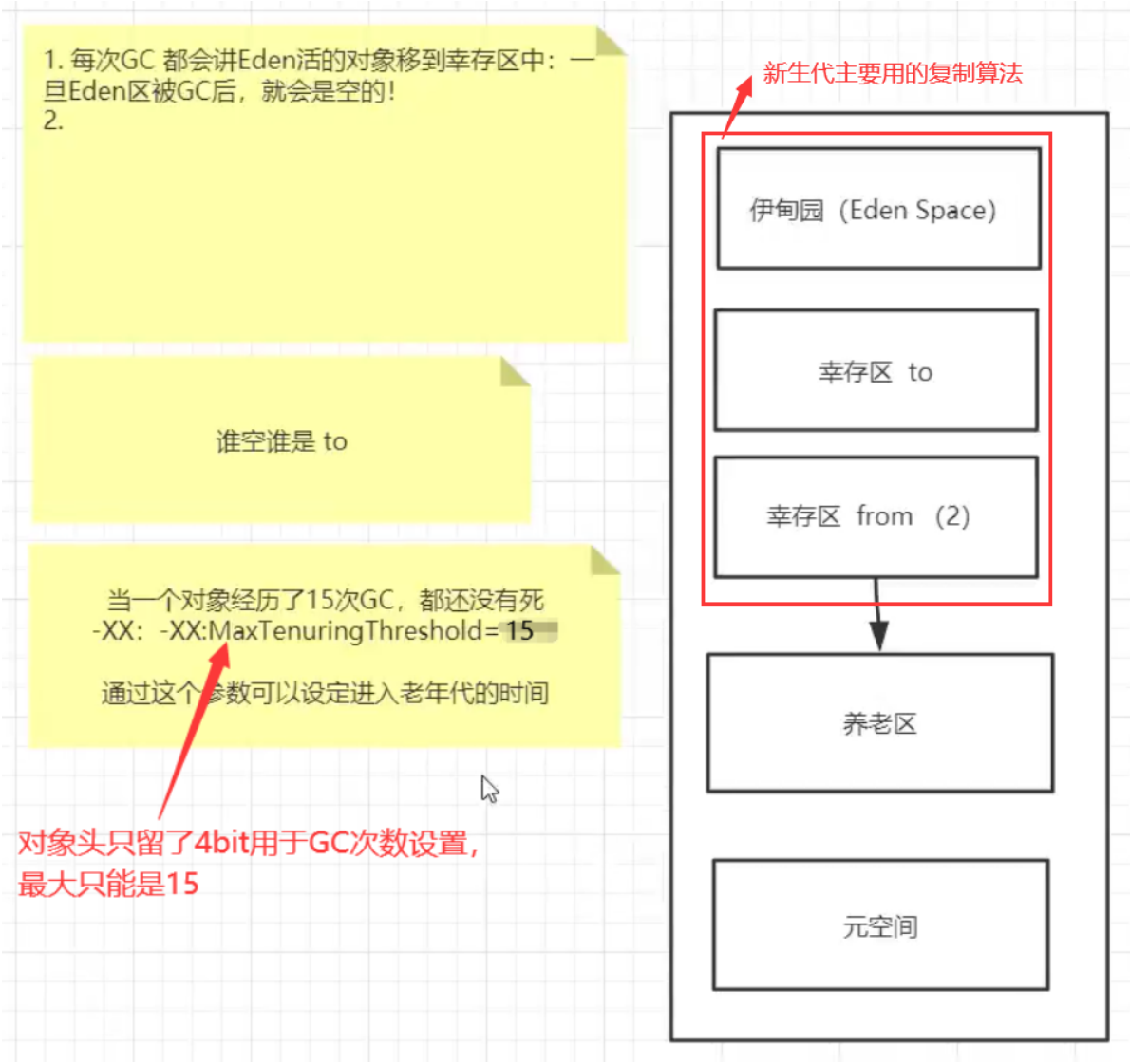
关于*可达性*的对象，便是能与GC Roots构成连通图的对象，如下图：

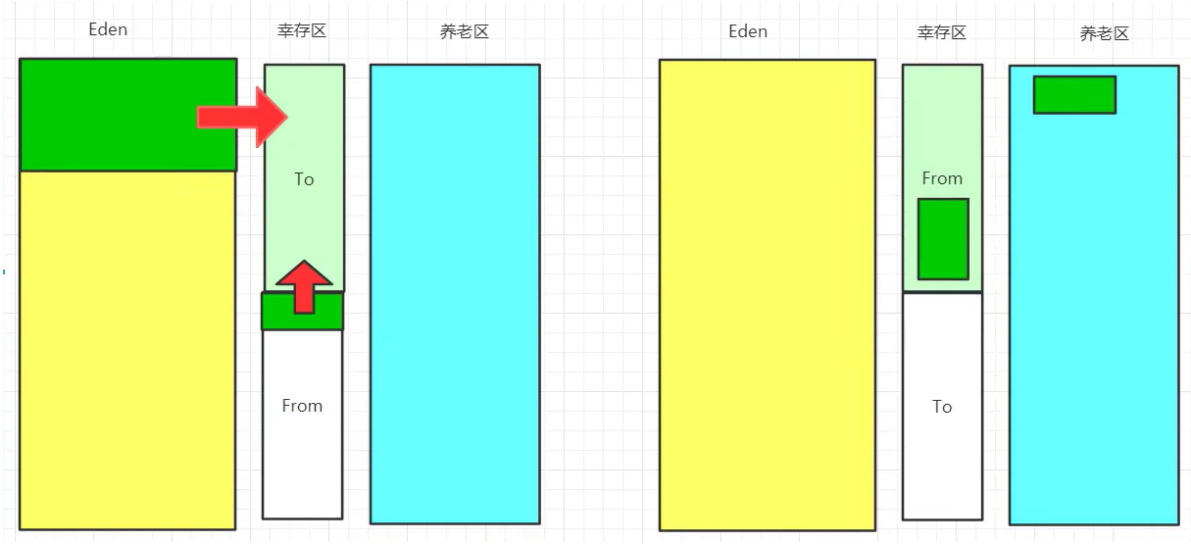


链接

复制算法

默认：Eden：from：to=8：1：1





- 好处：没有内存的碎片
- 坏处：浪费了内存空间：多了一半空间永远时空（to区）。假设100%存活（极端情况），from到to区每次都会移动大量的数据。

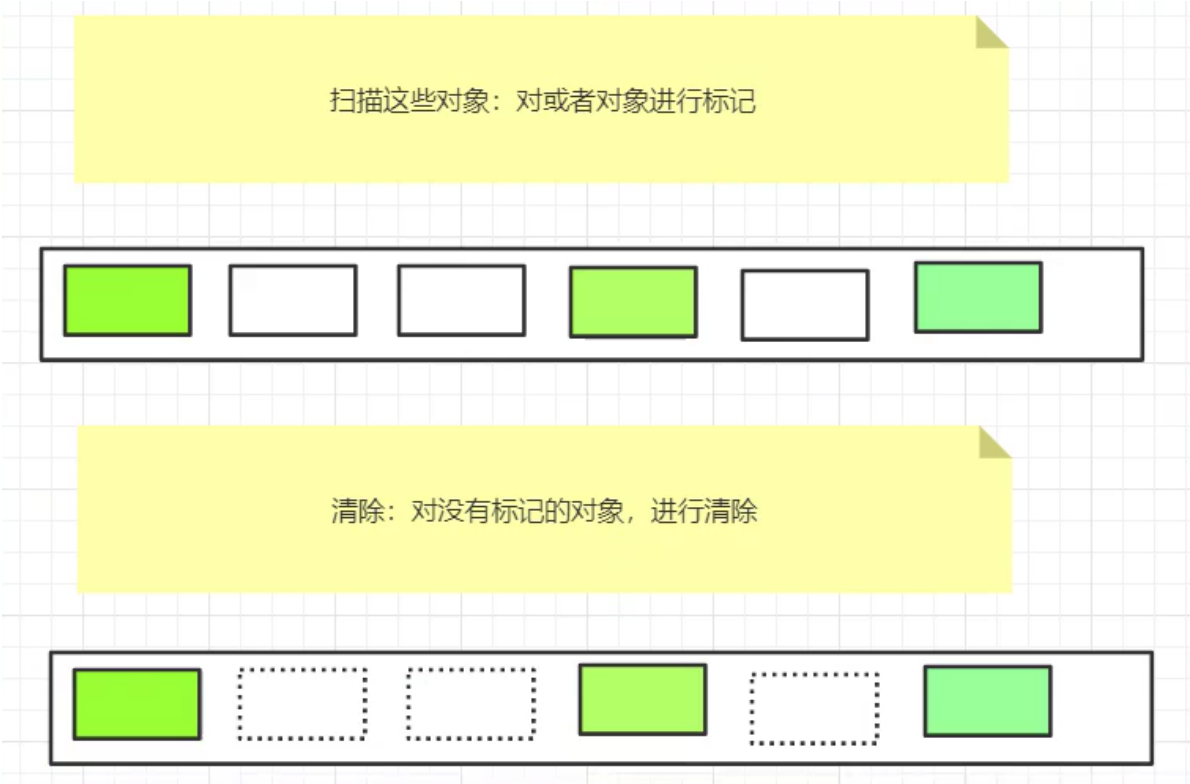
复制算法最佳使用场景：对象存活度较低的时候。即 **新生区**

扩展：

- 采用-XX:PretenureSizeThreshold参数可以设置多大的对象可以直接进入老年代内存区域。
- 采用-XX:MaxTenuringThreshold=数字 参数可以设置对象在经过多少次GC后会被放入老年代（年龄达到设置值，默认为15）。

标记清除算法

- 第一阶段--**标记：**遍历所有的GC Roots，然后将所有的GC Roots可达的对象标记为存活的对象。
- 第二阶段--**清除：**将遍历堆中所有的对象中没有标记的对象全部清除掉



优点：

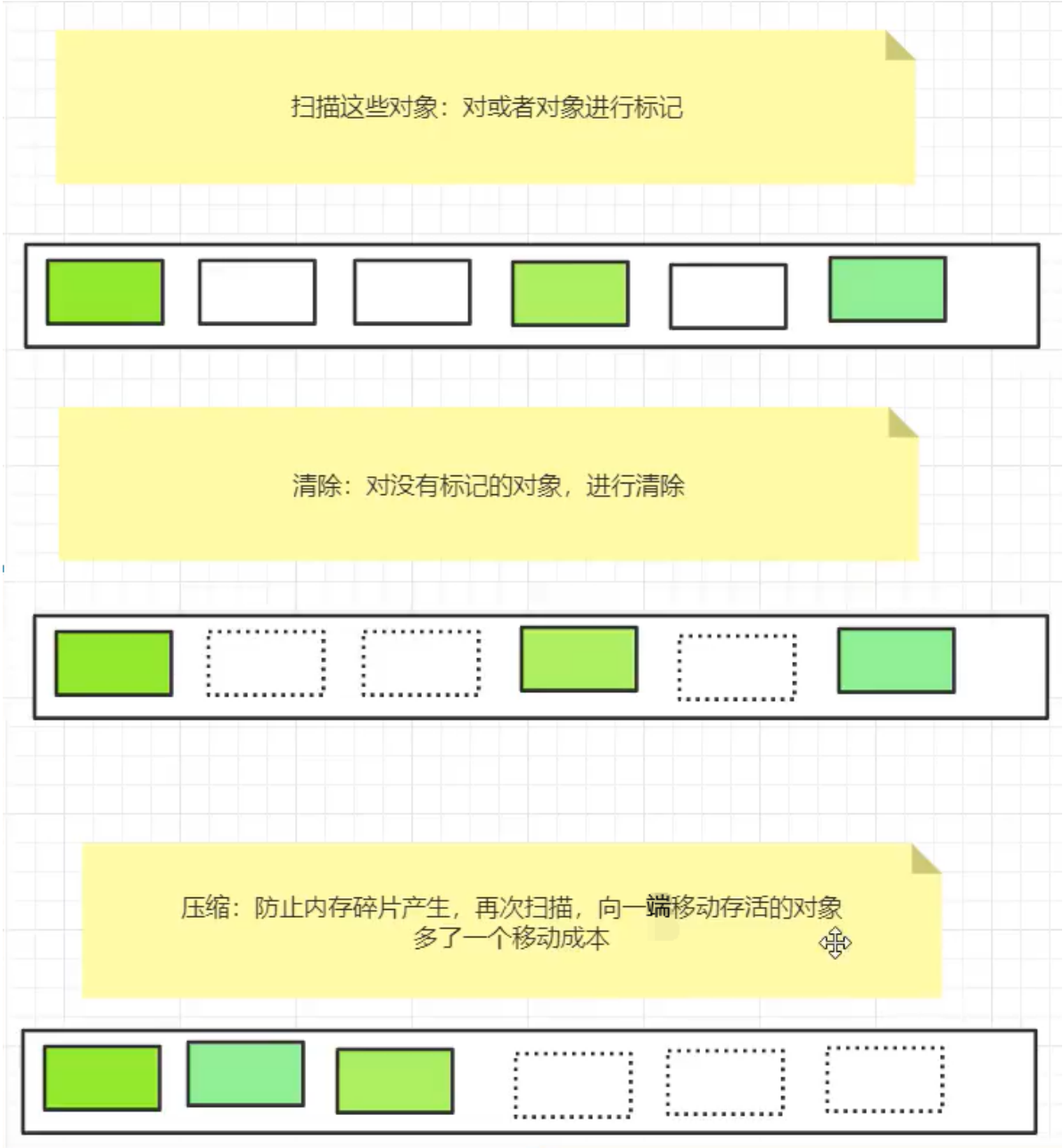
- 不需要额外的空间！
- 不移动对象

缺点：两次扫描，严重浪费时间，会产生内存碎片。

- 效率较低：因为标记和清除这两个过程效率都比较低
- 空间问题：标记清除后会产生大量不联系的内存空间（碎片），导致如果有大内存的对象，那么就无法找到足够更大的连续空间以供分配。

标记压缩/整理 算法

在标记清除算法的基础上进行优化



算法总结

内存效率：复制算法>标记清除算法>标记整理算法（时间复杂度）

内存整齐度：复制算法=标记整理算法>标记清除算法

内存利用率：标记清除算法=标记整理算法>复制算法（空间复杂度）

思考问题：难道没有最优的算法吗?

结论：没有最好的算法，只有最合适的算法-->GC：**分代收集算法**

年轻代：

- 存活率低-->复制算法!

老年代：

- 区域大：存活率高-->标记清除（内存碎片不是太多时）+标记压缩（内存碎片多时） 混合实现

扩展：垃圾收集器

Parellel Scavenge收集器

Parellel Scavenge收集器也是一个青年代，采用复制算法的收集器。和CMS收集器尽可能缩短垃圾收集时Stop The World停顿时间不同，Parellel Scavenge收集器的主要关注点在于达到一个可控制的吞吐量(Throughtput)。

> 吞吐量就是CPU用于运行用户代码的时间/CPU总消耗时间，即吞吐量 = 运行用户代码的时间/(运行用户代码+垃圾收集时间)

低停顿时间的关注点在于以良好的响应速度和低延迟来提升用户体验，适合需要和用户有较多交互的场景；而高吞吐量的关注点在于可以高效率地利用CPU时间以尽快完成运算任务，此场景主要适合较少用户交互多后台计算任务的场景。

Parellel Old收集器

是Parellel Scavenge的老年代版本，采用多线程和标记-压缩算法。

.CMS收集器

CMS收集器全名(Concurent Mark Sweep),从名字可以看出这款收集器是一款比较优秀的**基于标记-清除算法的并发收集器**。之前也提到过，此收集器的目标在于尽量小的Stop The World间隔时间，用于用户交互比较多的场景。它的收集过程分为4步：

- **初始标记**
- **并发标记**
- **重新标记**
- **并发清除**

其中初始标记和重新标记两个步骤仍需要Stop The World间隔。初始标记仅仅是标记一下GC Roots能直接关联到的对象，速度很快。并发标记阶段就是进行GC Roots追踪的过程，而重新标记则是为了修正并发标记期间由于用户程序继续执行可能产生变动的那部分对象的标记记录，此阶段会比初始标记长一些，但远小于并发标记的时间。

整个阶段并发标记和并发清除是耗时最长的两个阶段。但是由于CMS收集器是并发执行的，故可以和用户线程一起工作，所以从整体上CMS收集器的工作过程是和用户线程并发执行的。

优点：

- **GC收集间隔时间短，多线程并发。**

缺点：

- **1.并发时对CPU资源占用多，不适合CPU核心数较少的情况。**
- **2.且由于采用标记清除算法，所以会产生内存碎片。**
- **3.无法处理浮动垃圾。**

> 浮动垃圾：CMS并发清除阶段由于用户线程还可以继续执行，所以可能会产生新的垃圾）

> 以上垃圾收集器的内容摘自《深入理解Java虚拟机-周志明》-第二版，由于Java更新速度很快，最新的Java11已经出来了 ，所以书中部分垃圾收集器可能已经被HotSpot Vm弃用了，但主要在于学习其思想。
> jdk1.7 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）

> jdk1.8 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）
> jdk1.9 默认垃圾收集器G1

G1收集器(Garbage-First (G1))

G1收集器(Garbage-First (G1))是收集器技术发展最前沿的成果之一，HotSpot团队赋予它的使命是可以替换掉CMS的收集器，与其他GC收集器相比，G1收集器拥有以下特点：

并行与并发：G1能充分利用多CPU下的优势来缩短Stop The World的时间，同时在其他部分收集器需要停止Java线程来执行GC动作时，G1收集器仍然可以通过并发来让Java线程同步执行。

分代收集：与其他收集器一样，分代的概念在G1中任然被保留。可以不需要配合其他的垃圾收集器，就独立管理整个Java堆内存的所有分代区域，且采用不同的方式来获得更好的垃圾收集效果。

空间整合：G1从整体来看，使用的是标记-压缩算法实现的，从局部两个Region来看，采用的是复制算法实现的，对内存空间的利用非常高效，不会像CMS一样产生内存碎片。

可以预测的停顿：除了追求低停顿以外，G1的停顿时间可以被指定在一个时间范围内。
如果不计算维护Remenbered Set的操作，G1收集器的工作阶段大致区分如下：

- 初始标记
- 并发标记
- 最终标记
- 筛选回收

其实，Java11官网描述中已经说明：G1取代了Concurrent Mark-Sweep（CMS）收集器。它也是默认的收集器。表明在Java11中G1是默认的垃圾收集器，而CMS收集器从JDK 9开始就不推荐使用了（deprecated）。

参数	内容
-Xms	初始堆大小。如：-Xms256m
-Xmx	最大堆大小。如：-Xmx512m
-Xmn	新生代大小。通常为 Xmx 的 1/3 或 1/4。新生代 = Eden + 2 个 Survivor 空间。实际可用空间为 = Eden + 1 个 Survivor，即 90%
-Xss	JDK1.5+ 每个线程堆栈大小为 1M，一般来说如果栈不是很深的话， 1M 是绝对够用了的。
-XX:NewRatio	新生代与老年代的比例，如 –XX:NewRatio=2，则新生代占整个堆空间的1/3，老年代占2/3
-XX:SurvivorRatio	新生代中 Eden(8) 与 Survivor(1+1) 的比值。默认值为 8。即 Eden 占新生代空间的 8/10，另外两个 Survivor 各占 1/10
-XX:PermSize	永久代(方法区)的初始大小
-XX:MaxPermSize	永久代(方法区)的最大值
-XX:+PrintGCDetails	打印 GC 信息
-XX:+HeapDumpOnOutOfMemoryError	让虚拟机在发生内存溢出时 Dump 出当前的内存堆转储快照，以便分析用
-XX:HeapDumpPath	自定义指定oom时dump文件存储路径，不配置时保存到默认路径

13. JMM

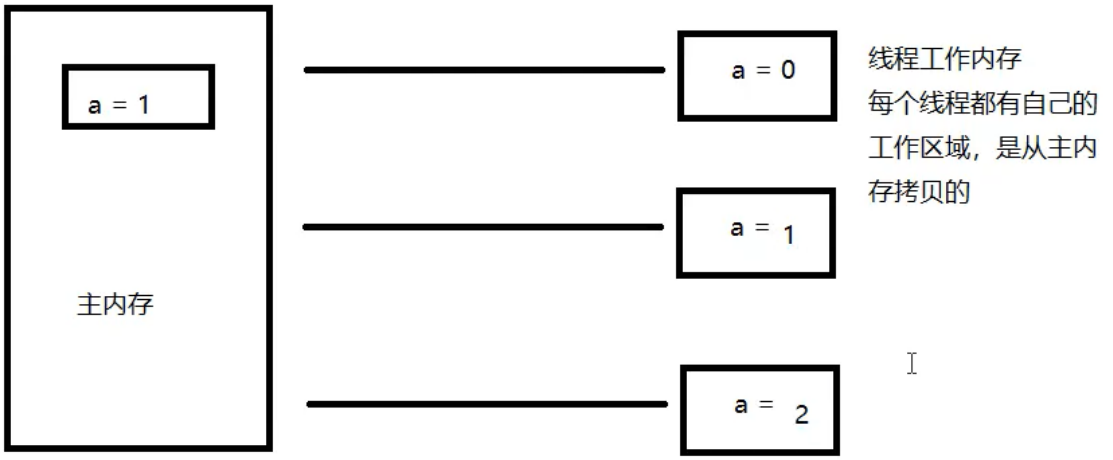
1、什么是 JMM?

JMM： (Java Memory Model 的缩写)

2、JMM 是干嘛的？：官方，其他人的博客，对应的视频！

作用：缓存一致性协议，用于定义数据读写的规则（遵守，找到这个规则）。

JMM 定义了线程工作内存和主内存之间的抽象关系：线程之间的共享变量存储在主内存（Main Memory）中，每个线程都有一个私有的本地内存（Local Memory）



产生问题：数据不一致！

解决共享对象可见性这个问题：volatile

3、JMM 该如何学习?

JMM：抽象的概念，理论

内存交互操作:

内存交互操作有8种，虚拟机实现必须保证每一个操作都是原子的，不可在分的（对于double和long类型的变量来说，load、store、read和write操作在某些平台上允许例外）

- lock （锁定）：作用于主内存的变量，把一个变量标识为线程独占状态
- unlock （解锁）：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定
- read （读取）：作用于主内存变量，它把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用
- load （载入）：作用于工作内存的变量，它把read操作从主存中变量放入工作内存中
- use （使用）：作用于工作内存中的变量，它把工作内存中的变量传输给执行引擎，每当虚拟机遇到一个需要使用到变量的值，就会使用到这个指令
- assign （赋值）：作用于工作内存中的变量，它把一个从执行引擎中接受到的值放入工作内存的变量副本中
- store （存储）：作用于工作内存中的变量，它把一个从工作内存中一个变量的值传送到主内存中，以便后续的write使用
- write （写入）：作用于主内存中的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中

JMM对这八种指令的使用，制定了如下规则：

- 不允许read和load、store和write操作之一单独出现。即使用了read必须load，使用了store必须write
- 不允许线程丢弃他最近的assign操作，即工作变量的数据改变了之后，必须告知主存
- 不允许一个线程将没有assign的数据从工作内存同步回主内存
- 一个新的变量必须在主内存中诞生，不允许工作内存直接使用一个未被初始化的变量。就是怼变量实施use、store操作之前，必须经过assign和load操作
- 一个变量同一时间只有一个线程能对其进行lock。多次lock后，必须执行相同次数的unlock才能解锁
- 如果对一个变量进行lock操作，会清空所有工作内存中此变量的值，在执行引擎使用这个变量前，必须重新load或assign操作初始化变量的值

- 如果一个变量没有被lock，就不能对其进行unlock操作。也不能unlock一个被其他线程锁住的变量
- 对一个变量进行unlock操作之前，必须把此变量同步回主内存

14. 总结

学习新东西是常态：

面试：

3/10==pass，面经=10，分析这10个题，触类旁通

通过大量的面试总结，得出一套解题思路

学习途径：

1.百度

2.思维导图