源码到类文件

> 编译: javac Person.java ---> Person.class

编译过程

Person.java -> 词法分析器 -> tokens流 -> 语法分析器 -> 语法树/抽象语法树 -> 语义分析器 -> 注解抽象语法树 -> 字节码生成器 -> Person.class文件

类文件(Class文件)

.class字节码文件

> 魔数与class文件版本
> 常量池
> 访问标志
> 类索引、父类索引、接口索引
> 字段表集合
> 方法表集合
> 属性表集合

类加载机制

类文件到虚拟机:Person.class -- > 类加载机制--> 虚拟机

- 装载 (load)

  查找和导入class文件

  (1)通过一个类的全限定名获取定义此类的二进制字节流 (找到类文件所在的位置)
  (2)将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构(类文件信息交给JVM)
  (3)在Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口 (类文件所应的对象class->JVM)

- 链接(Link)
  - 验证(Verify)

    保证被加载类的正确性

    > 文件格式验证
    > 元数据验证
    > 字节码验证
    > 符号引用验证

  - 准备(Prepare)

    为类的静态变量分配内存，并将其初始化为默认值

    例如：static int a=10; 这个阶段 a=0

  - 解析(Resolve)

    把类中的符号引用转换为直接引用

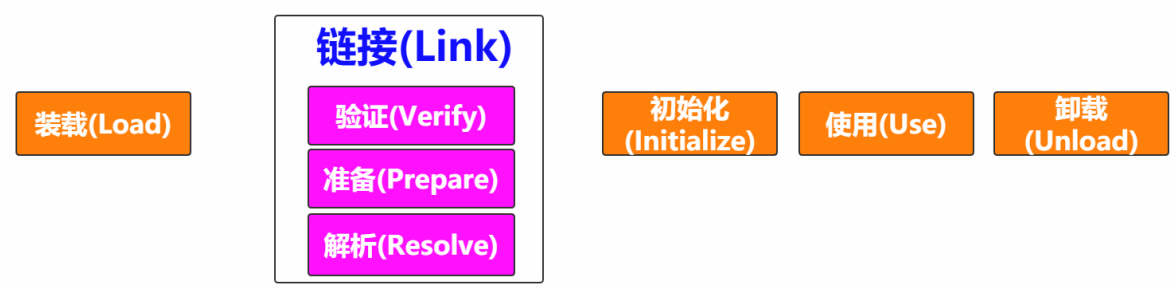
- 初始化(Initialize)

对类的静态变量，静态代码块执行初始化操作

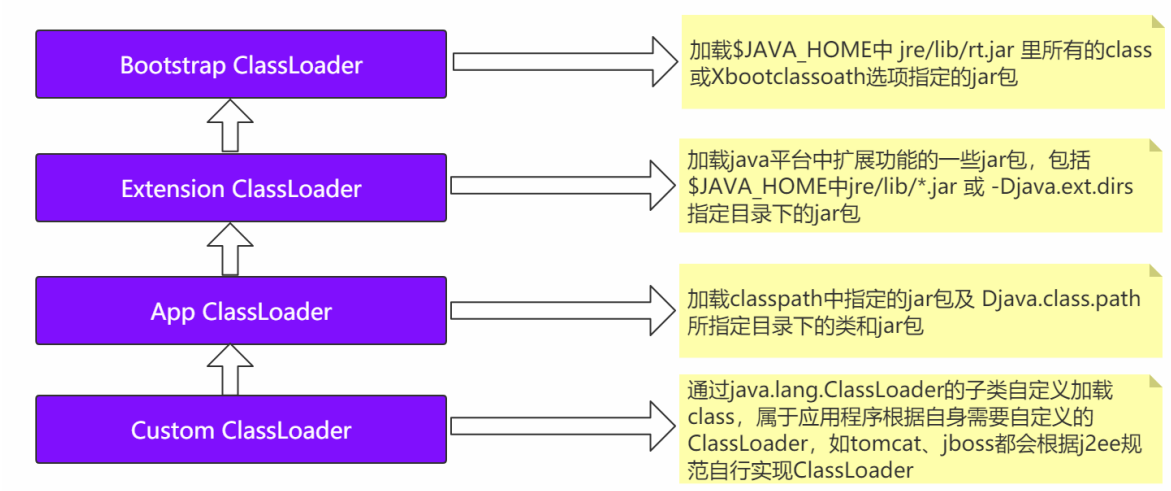
例如：static int a=10; 这个阶段 a=10

类加载机制图解

使用和卸载不算是类加载过程中的阶段，只是画完整了一下



类装载器ClassLoader



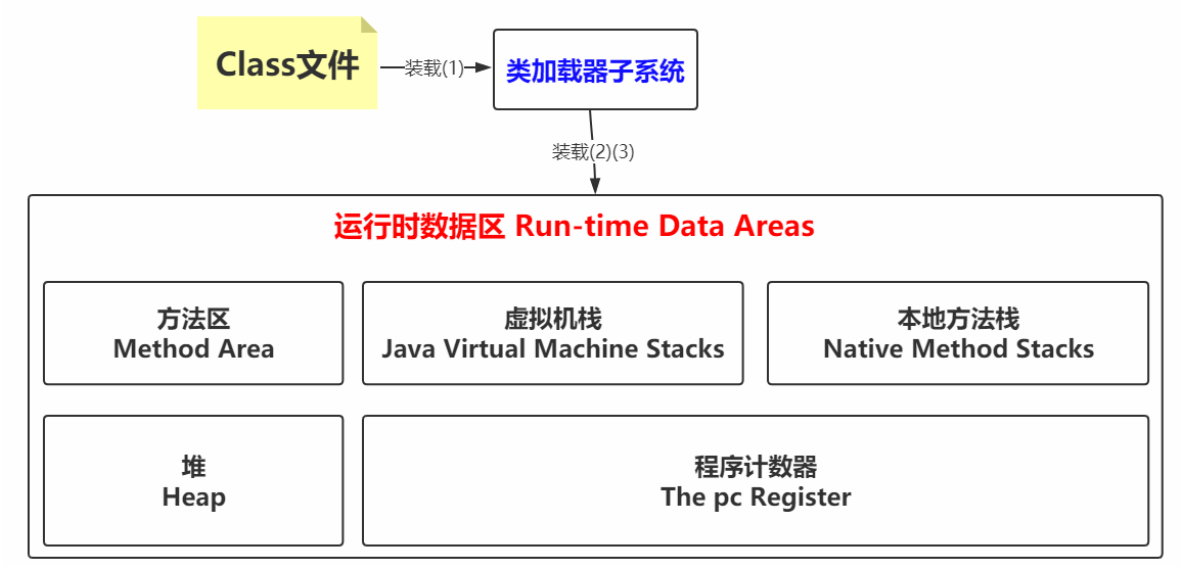
加载原则

检查某个类是否已经加载：顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个Classloader已加载，就视为已加载此类，保证此类只所有ClassLoader加载一次。
加载的顺序：加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

双亲委派机制

定义：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，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。
优势：Java类随着加载它的类加载器一起具备了一种带有优先级的层次关系。比如，Java中的Object类，它存放在rt.jar之中,无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此Object在各种类加载环境中都是同一个类。如果不采用双亲委派模型，那么由各个类加载器自己取加载的话，那么系统中会存在多种不同的Object类。
破坏：可以继承ClassLoader类，然后重写其中的loadClass方法，其他方式大家可以自己了解拓展一下。

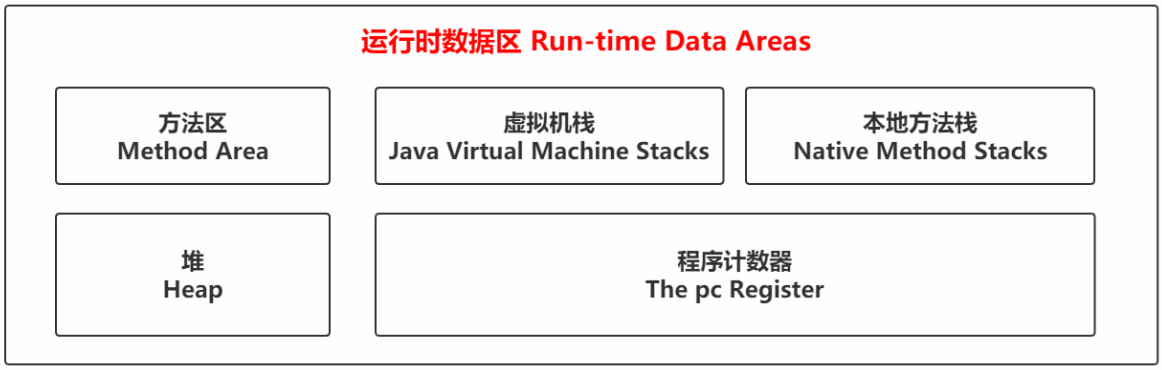
Class文件装载



运行时数据区(Run-Time Data Areas)

官网 https://docs.oracle.com/javase/specs/jvms/se8/html/index.html

就是类文件被类装载器装载进来之后，类中的内容(比如变量，常量，方法，对象等这些数据得要有个去处，也就是要存储起来，存储的位置肯定是在JVM中有对应的空间)



Method Area(方法区)

方法区是各个线程共享的内存区域，在虚拟机启动时创建。
用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据

虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却又一个别名叫做Non-Heap(非堆)，目的是与Java堆区分开来。
当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常

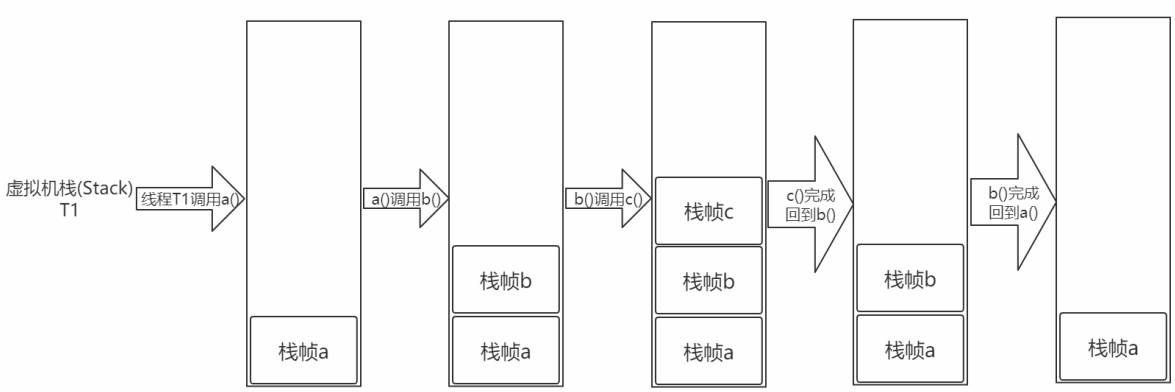
> (1)方法区在JDK 8中就是Metaspace，在JDK6或7中就是Perm Space
> (2)Run-Time Constant Pool
> Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池，用于存放编译时期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放

Heap(堆)

Java堆是Java虚拟机所管理内存中最大的一块，在虚拟机启动时创建，被所有线程共享。
Java对象实例以及数组都在堆上分配。

Java Virtual Machine Stacks(虚拟机栈)

虚拟机栈是一个线程执行的区域，保存着一个线程中方法的调用状态。换句话说，一个Java线程的运行状态，由一个虚拟机栈来保存，所以虚拟机栈肯定是线程私有的，独有的，随着线程的创建而创建。每一个被线程执行的方法，为该栈中的栈帧，即每个方法对应一个栈帧。调用一个方法，就会向栈中压入一个栈帧；一个方法调用完成，就会把该栈帧从栈中弹出。



The pc Register(程序计数器)

> 我们都知道一个JVM进程中有多个线程在执行，而线程中的内容是否能够拥有执行权，是根据CPU调度来的。
> 假如线程A正在执行到某个地方，突然失去了CPU的执行权，切换到线程B了，然后当线程A再获得CPU执行权的时候，怎么能继续执行呢？这就是需要在线程中维护一个变量，记录线程执行到的位置。

程序计数器占用的内存空间很小，由于Java虚拟机的多线程是通过线程轮流切换，并分配处理器执行时间的方式来实现的，在任意时刻，一个处理器只会执行一条线程中的指令。因此，为了线程切换后能够恢复到正确的执行位置，每条线程需要有一个独立的程序计数器(线程私有)。如果线程正在执行Java方法，则计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Native方法，则这个计数器为空。

Native Method Stacks(本地方法栈)

如果当前线程执行的方法是Native类型的，这些方法就会在本地方法栈中执行

理解Java虚拟机栈和栈帧

栈帧：每个栈帧对应一个被调用的方法，可以理解为一个方法的运行空间

局部变量表:方法中定义的局部变量以及方法的参数存放在这张表中
局部变量表中的变量不可直接使用，如需要使用的话，必须通过相关指令将其加载至操作数栈中作为操作数使用。

操作数栈:以压栈和出栈的方式存储操作数的

动态链接:每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接(Dynamic Linking)。

方法返回地址:当一个方法开始执行后,只有两种方式可以退出，一种是遇到方法返回的字节码指令；一种是遇见异常，并且这个异常没有在方法体内得到处理。

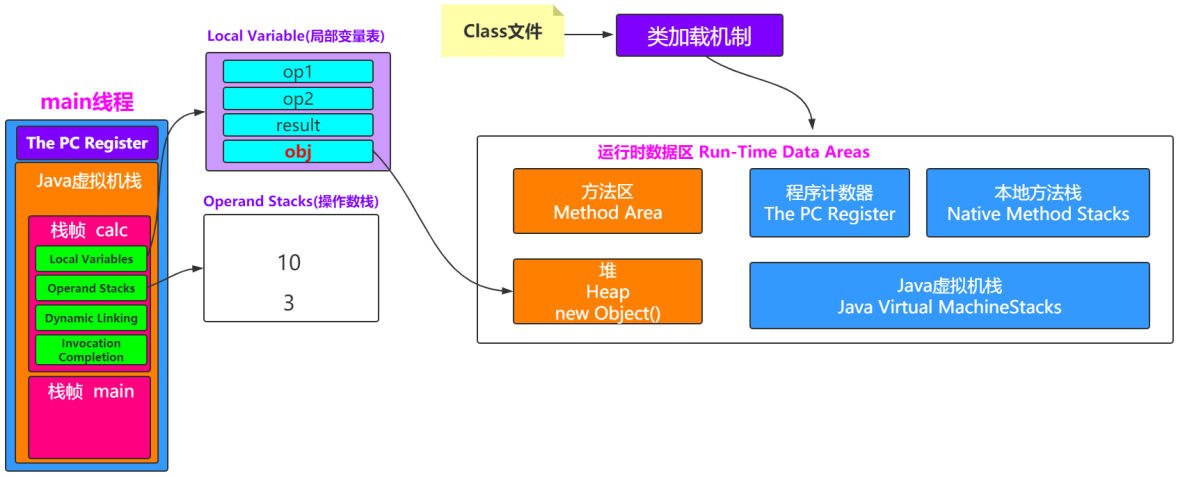
演示代码

```
class Person{
    private String name="Jack";
    private int age;
    private final double salary=100;
    private static String address;
    private final static String hobby="Programming";
    public void say(){
    System.out.println("person say...");
    }
    public static int calc(int op1,int op2){
        op1=3;
        int result=op1+op2;
        return result;
    }
    public static void order(){
    }
    public static void main(String[] args){
        calc(1,2);
        order();
    }
}
```

```
Compiled from "Person.java"
class Person {
...
public static int calc(int, int);
Code:
    0: iconst_3 //将int类型常量3压入[操作数栈]
    1: istore_0 //将int类型值存入[局部变量0]
    2: iload_0 //从[局部变量0]中装载int类型值入栈
    3: iload_1 //从[局部变量1]中装载int类型值入栈
    4: iadd //将栈顶元素弹出栈，执行int类型的加法，结果入栈
    5: istore_2 //将栈顶int类型值保存到[局部变量2]中
    6: iload_2 //从[局部变量2]中装载int类型值入栈
    7: ireturn //从方法中返回int类型的数据
...
}
```

栈指向堆

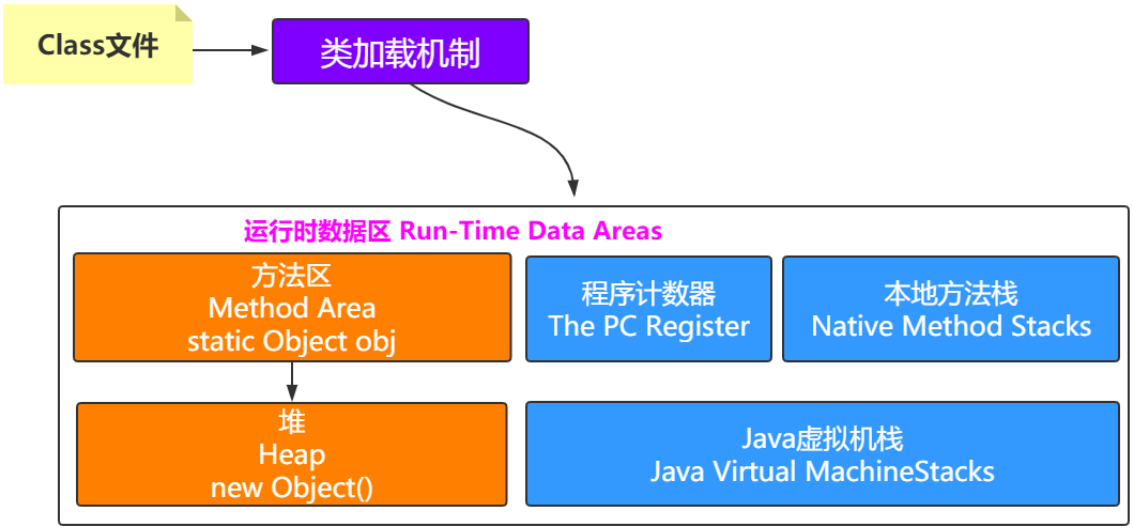
如果在栈帧中有一个变量，类型为引用类型，比如Object obj=new Object()，这时候就是典型的栈中元素指向堆中的对象。



方法区指向堆

方法区中会存放静态变量，常量等数据。如果是下面这种情况，就是典型的方法区中元素指向堆中的对象。

private static Object obj=new Object();



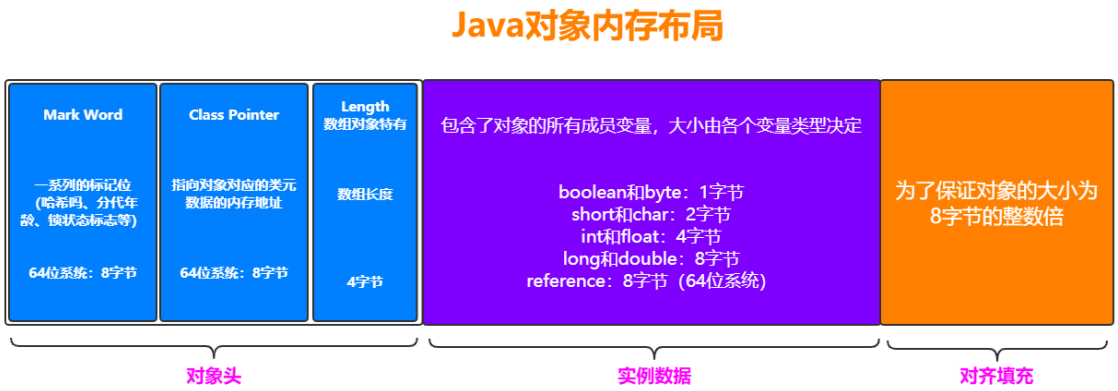
堆指向方法区

方法区中会包含类的信息，堆中会有对象，那怎么知道对象是哪个类创建的呢?

一个对象怎么知道它是由哪个类创建出来的？怎么记录？就需要了解一个Java对象的具体信息

Java对象内存布局

一个Java对象在内存中包括3个部分：对象头、实例数据和对齐填充



内存模型

一块是非堆区，一块是堆区。堆区分为两大块，一个是Old区，一个是Young区。
Young区分为两大块，一个是Survivor区（S0+S1），一块是Eden区。 Eden:S0:S1=8:1:1
S0和S1一样大，也可以叫From和To。

对象创建所在区域

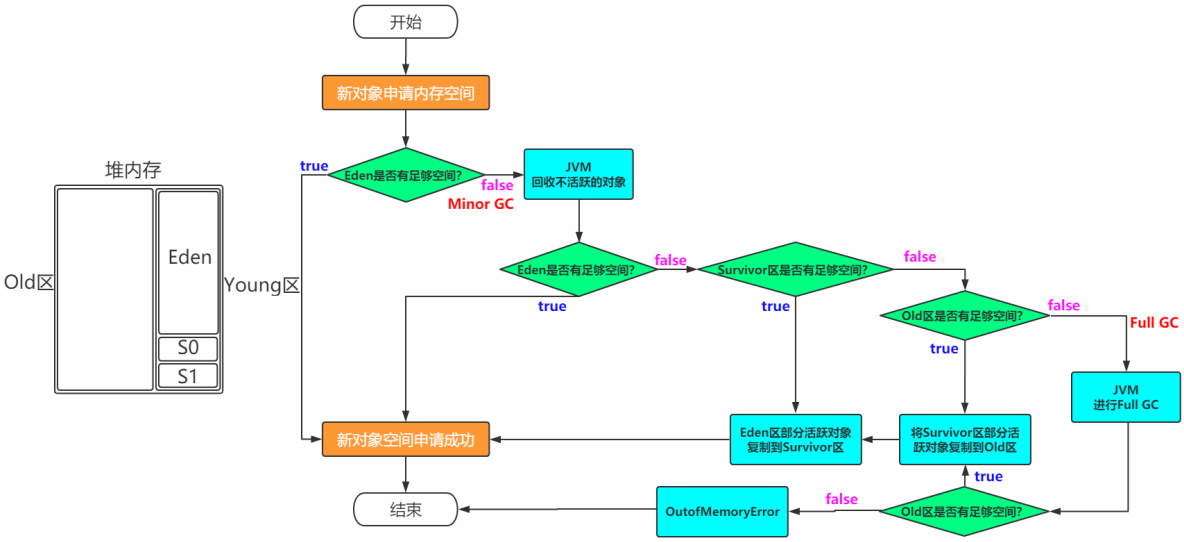
一般情况下，新创建的对象都会被分配到Eden区，一些特殊的大的对象会直接分配到Old区。
比如有对象A，B，C等创建在Eden区，但是Eden区的内存空间肯定有限，比如有100M，假如已经使用了
100M或者达到一个设定的临界值，这时候就需要对Eden内存空间进行清理，即垃圾收集(Garbage Collect)，
这样的GC我们称之为Minor GC，Minor GC指得是Young区的GC。
经过GC之后，有些对象就会被清理掉，有些对象可能还存活着，对于存活着的对象需要将其复制到Survivor
区，然后再清空Eden区中的这些对象。

Survivor区详解

Survivor区分为两块S0和S1，也可以叫做From和To。
在同一个时间点上，S0和S1只能有一个区有数据，另外一个是空的。

Old区详解

一般Old区都是年龄比较大的对象，或者相对超过了某个阈值的对象。
在Old区也会有GC的操作，Old区的GC我们称作为Major GC。



> Minor GC:新生代
> Major GC:老年代
> Full GC:新生代+老年代

Garbage Collect(垃圾回收)

如何确定一个对象是垃圾

- 引用计数法

  要应用程序中持有该对象的引用，就说明该对象不是垃圾，如果一个对象没有任何指针对其引用，它就是垃圾

  > 弊端 :如果AB相互持有引用，导致永远不能被回收。

- 可达性分析

  通过GC Root的对象，开始向下寻找，看某个对象是否可达

  > 能作为GC Root:类加载器、Thread、虚拟机栈的本地变量表、static成员、常量引用、本地方法
  > 栈的变量等

垃圾收集算法

- 标记-清除(Mark-Sweep)
  - 标记:找出内存中需要回收的对象，并且把它们标记出来

    > 此时堆中所有的对象都会被扫描一遍，从而才能确定需要回收的对象，比较耗时

  - 清除

    清除掉被标记需要回收的对象，释放出对应的内存空间

  > 缺点:
  >
  > 标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。
  > (1)标记和清除两个过程都比较耗时，效率不高
  > (2)会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作

- 复制(Copying)

将内存划分为两块相等的区域，每次只使用其中一块；

当其中一块内存使用完了，就将还存活的对象复制到另外一块上面，然后把已经使用过的内存空间一次清除掉。

> 缺点: 空间利用率降低

- 标记-整理(Mark-Compact)

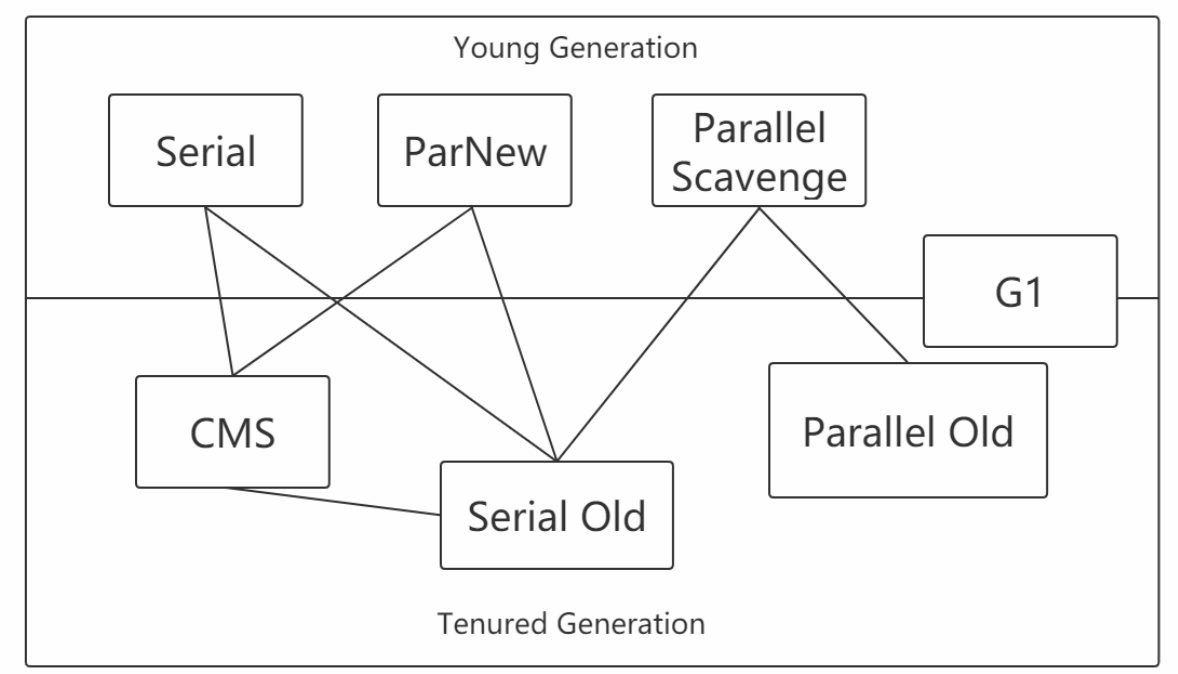
  标记过程仍然与"标记-清除"算法一样，但是后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

分代收集算法

Young区：复制算法(对象在被分配之后，可能生命周期比较短，Young区复制效率比较高)
Old区：标记清除或标记整理(Old区对象存活时间比较长，复制来复制去没必要，不如做个标记再清理)

垃圾收集器

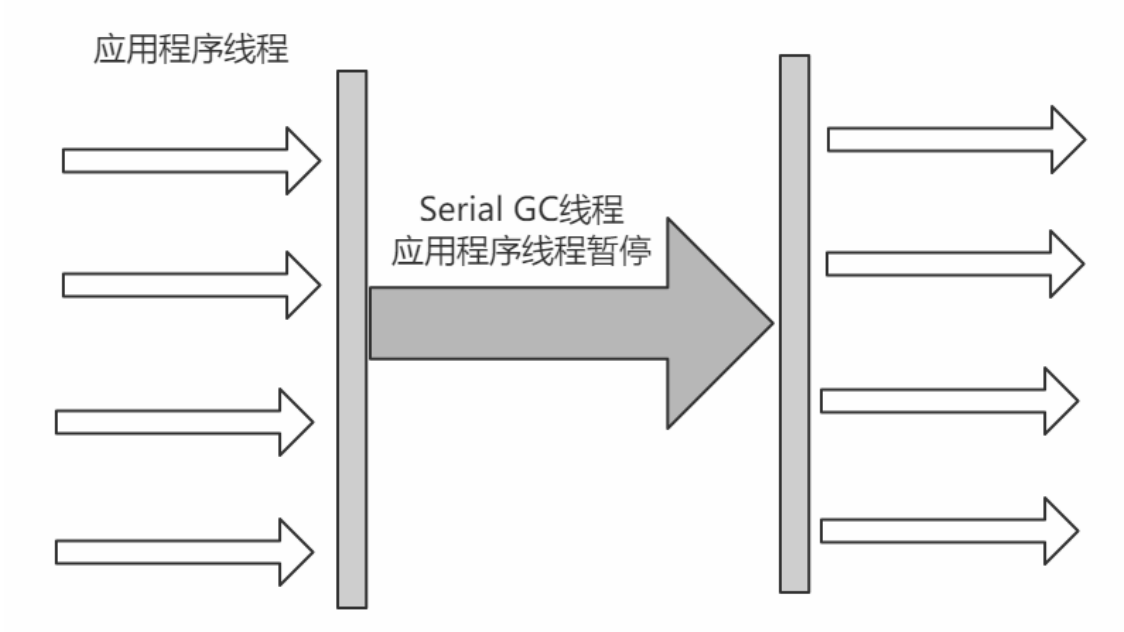
如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现，说白了就是落地。



- Serial收集器

  Serial收集器是最基本、发展历史最悠久的收集器，曾经（在JDK1.3.1之前）是虚拟机新生代收集的唯一选择。它是一种单线程收集器，不仅仅意味着它只会使用一个CPU或者一条收集线程去完成垃圾收集工作，更重要的是其在进行垃圾收集的时候需要暂停其他线程。优点：简单高效，拥有很高的单线程收集效率

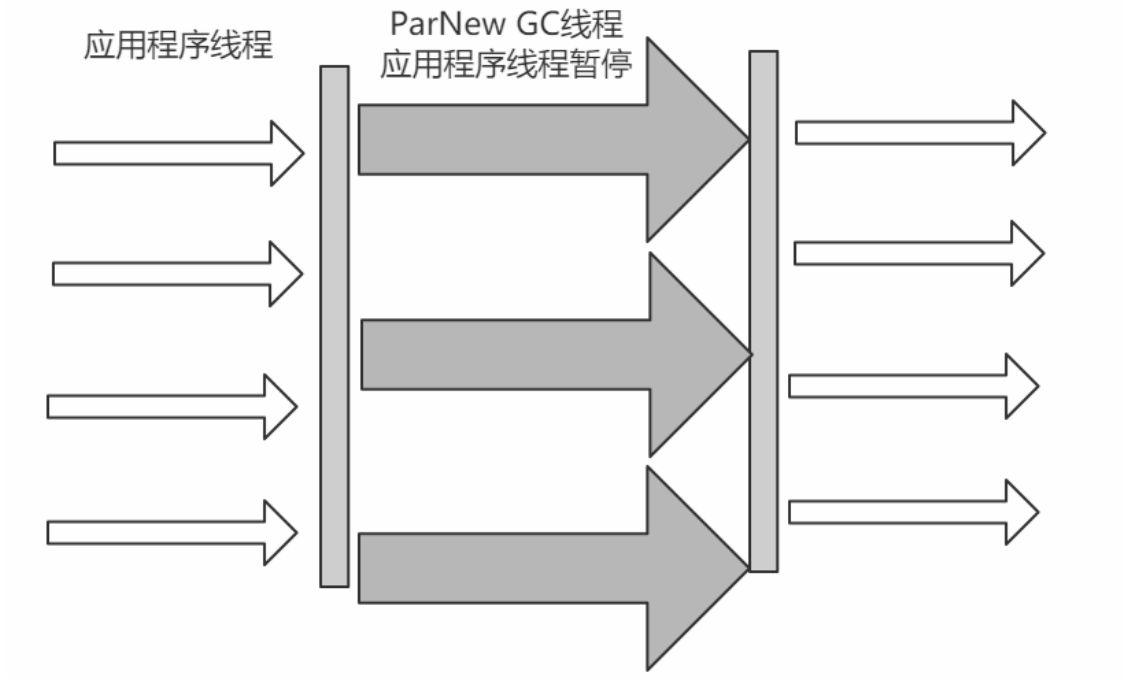
  > 缺点：收集过程需要暂停所有线程
  > 算法：复制算法
  > 适用范围：新生代
  > 应用：Client模式下的默认新生代收集器



- ParNew收集器

  可以把这个收集器理解为Serial收集器的多线程版本。

  > 优点：在多CPU时，比Serial效率高。
  > 缺点：收集过程暂停所有应用程序线程，单CPU时比Serial效率差。
  > 算法：复制算法
  > 适用范围：新生代
  > 应用：运行在Server模式下的虚拟机中首选的新生代收集器



- Parallel Scavenge收集器

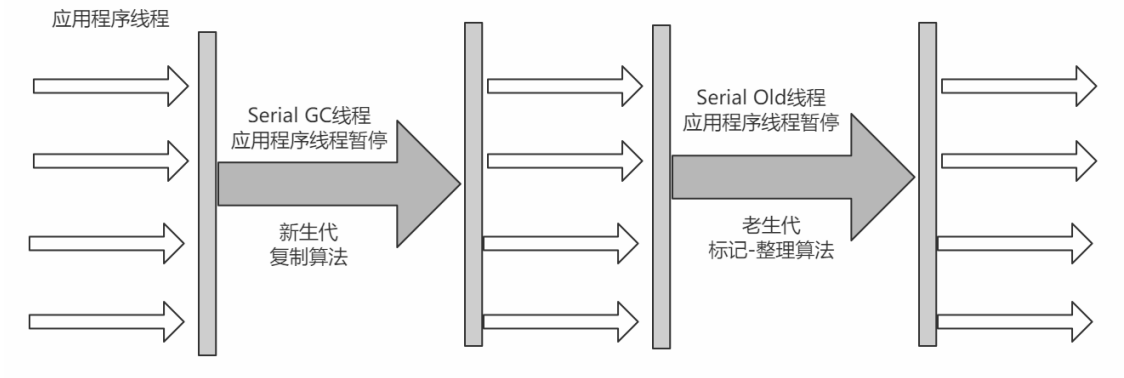
  Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器，看上去和ParNew一样，但是Parallel Scanvenge更关注 系统的吞吐量

  > 吞吐量=运行用户代码的时间/(运行用户代码的时间+垃圾收集时间)
  > 比如虚拟机总共运行了100分钟，垃圾收集时间用了1分钟，吞吐量=(100-1)/100=99%。
  > 若吞吐量越大，意味着垃圾收集的时间越短，则用户代码可以充分利用CPU资源，尽快完成程序的运算任务。

```
-XX:MaxGCPauseMillis控制最大的垃圾收集停顿时间，
-XX:GCTimeRatio直接设置吞吐量的大小。
```

- Serial Old收集器

  Serial Old收集器是Serial收集器的老年代版本，也是一个单线程收集器，不同的是采用"标记-整理算法"，运行过程和Serial收集器一样。



- Parallel Old收集器

  Parallel Old收集器是Parallel Scavenge收集器的老年代版本，使用多线程和"标记-整理算法"进行垃圾回收。

  > 吞吐量优先

- CMS收集器

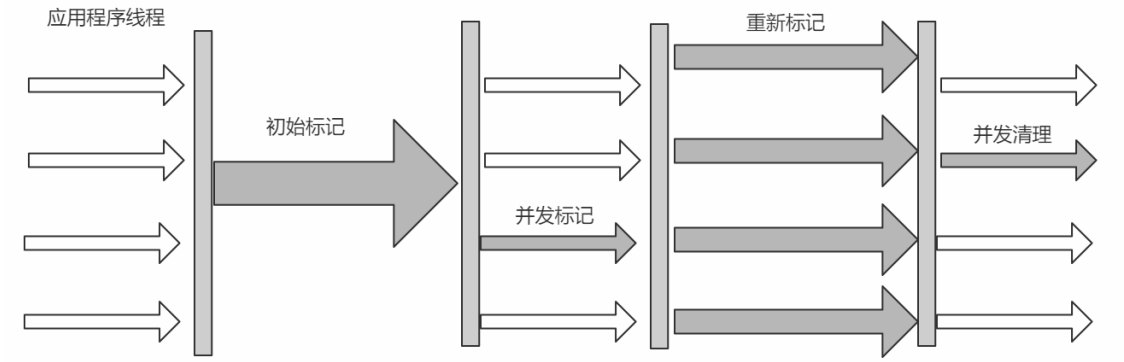
  CMS(Concurrent Mark Sweep)收集器是一种以获取 **最短回收停顿时间** 为目标的收集器。

  采用的是"标记-清除算法",整个过程分为4步

  (1)初始标记 CMS initial mark 标记GC Roots能关联到的对象 Stop The World--
  ->速度很快
  (2)并发标记 CMS concurrent mark 进行GC Roots Tracing
  (3)重新标记 CMS remark 修改并发标记因用户程序变动的内容 Stop The
  World
  (4)并发清除 CMS concurrent sweep

  由于整个过程中，并发标记和并发清除，收集器线程可以与用户线程一起工作，所以总体上来说，CMS收集器的内存回收过程是与用户线程一起并发地执行的

  > 优点：并发收集、低停顿
  > 缺点：产生大量空间碎片、并发阶段会降低吞吐量



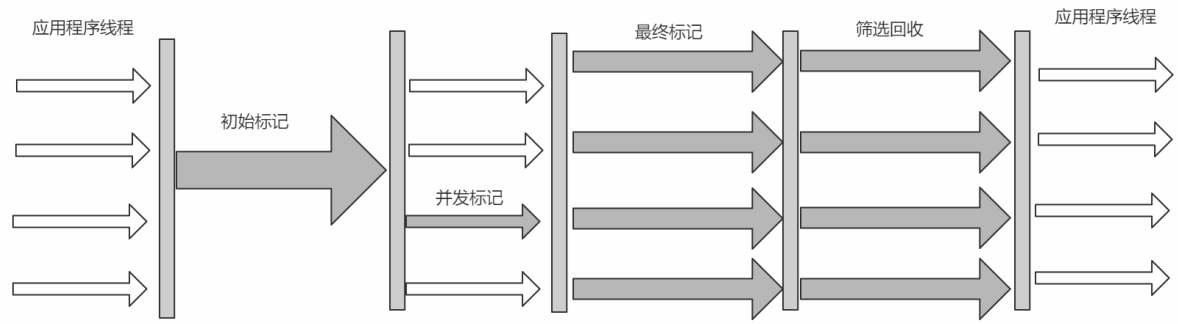
- G1收集器

特点：
并行与并发
分代收集（仍然保留了分代的概念）
空间整合（整体上属于“标记-整理”算法，不会导致空间碎片）
可预测的停顿（比CMS更先进的地方在于能让使用者明确指定一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒）

> 使用G1收集器时，Java堆的内存布局与就与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分Region（不需要连续）的集合。

工作过程可以分为如下几步：

初始标记（Initial Marking） 标记一下GC Roots能够关联的对象，并且修改TAMS的值，需要暂停用户线程
并发标记（Concurrent Marking） 从GC Roots进行可达性分析，找出存活的对象，与用户线程并发执行
最终标记（Final Marking） 修正在并发标记阶段因为用户程序的并发执行导致变动的数据，需暂停用户线程
筛选回收（Live Data Counting and Evacuation） 对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间制定回收计划



垃圾收集器分类

- 串行收集器->Serial和Serial Old
  只能有一个垃圾回收线程执行，用户线程暂停。 适用于内存比较小的嵌入式设备 。
- 并行收集器[吞吐量优先]->Parallel Scanvenge、Parallel Old
  多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。 适用于科学计算、后台处理等若交互场景。
- 并发收集器[停顿时间优先]->CMS、G1
  用户线程和垃圾收集线程同时执行(但并不一定是并行的，可能是交替执行的)，垃圾收集线程在执行的时候不会停顿用户线程的运行。 适用于相对时间有要求的场景，比如Web 。
-

理解吞吐量和停顿时间

- 停顿时间->垃圾收集器 进行 垃圾回收终端应用执行响应的时间
- 吞吐量->运行用户代码时间/(运行用户代码时间+垃圾收集时间)

> 停顿时间越短就越适合需要和用户交互的程序，良好的响应速度能提升用户体验；
> 高吞吐量则可以高效地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

这两个指标也是评价垃圾回收器好处的标准，其实调优也就是在观察者两个变量

如何选择合适的垃圾收集

官网：https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/collectors.html#sthref28

> 优先调整堆的大小让服务器自己来选择
> 如果内存小于100M，使用串行收集器
> 如果是单核，并且没有停顿时间要求，使用串行或JVM自己选
> 如果允许停顿时间超过1秒，选择并行或JVM自己选
> 如果响应时间最重要，并且不能超过1秒，使用并发收集器
> 对于G1收集

再次理解G1

JDK 9默认的垃圾收集器，适用于新老生代

判断是否需要使用G1收集器?

（1）50%以上的堆被存活对象占用
（2）对象分配和晋升的速度变化非常大
（3）垃圾回收时间比较长

如何开启需要的垃圾收集器

> （1）串行
> -XX：+UseSerialGC
> -XX：+UseSerialOldGC
> （2）并行(吞吐量优先)：
> -XX：+UseParallelGC
> -XX：+UseParallelOldGC
> （3）并发收集器(响应时间优先)
> -XX：+UseConcMarkSweepGC
> -XX：+UseG1GC