前言

> **先给大家看几道面试题?**

1、请你谈谈你对JVM的理解？Java8的虚拟机有什么更新?

2、什么是OOM？什么是StackOverFlowError？有哪些方法分析?

3、JVM的常用参数调优你知道哪些?

4、内存快照抓取和MAT分析DUMP文件知道吗?

5、堆里面的分区：Eden，Survival from to，老年代，各自的特点?

6、GC的三种收集方法：标记清除，标记整理，复制算法的原理与特点，分别用在什么地方?

> 唠叨几句

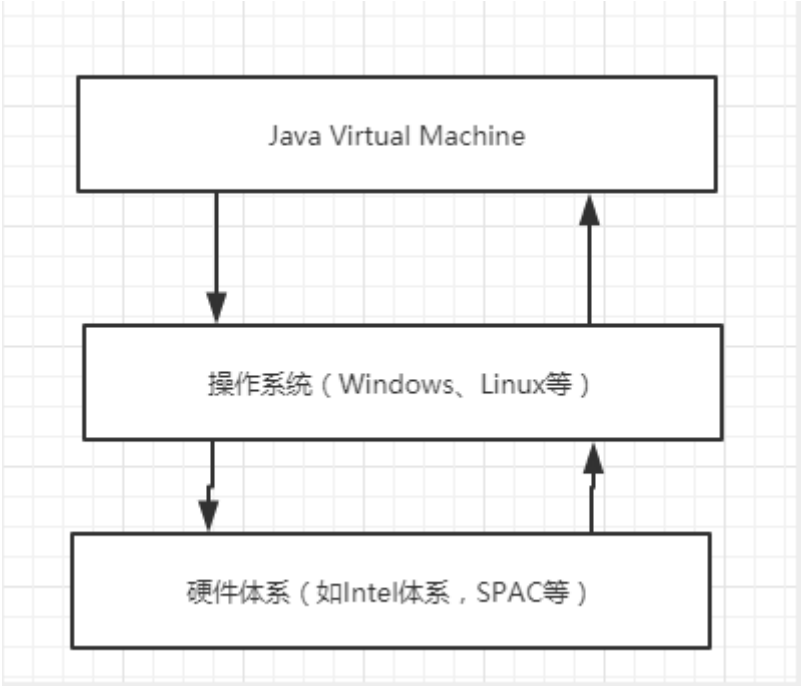
每一个学习JVM的人，都渴望成功。每一个Java开发人员的终极目标都是在日常生活中深入理解JVM的运行原理。JVM和平时的应用框架明显的区别，应用框架学习之后，可以直接拿来写项目了，就可以运行起来看到helloworld。然而对于JVM，是一个特别枯燥的事情，还看不到直接的效果，必须要写笔记，因为一扭头就会忘记。

JVM是一个令人望而却步的领域，因为它博大精深，涉及到的内容与知识点非常之多。虽然Java开发者每天都在使用JVM，但对其有所研究并且研究深入的人却少之又少。然而，JVM的重要性却又是不言而喻的。基于JVM的各种动态与静态语言生态圈已经异常繁荣了，对JVM的运行机制有一定的了解不但可以提升我们的竞争力，还可以让我们在面对问题时能够沉着应对，加速问题的解决速度；同时还能够增强我们的自信心，让我们更加游刃有余。

而且，如果我们想要进阶到技术专家或者更高等级，就必须要学习 JVM；

JVM的位置

首先我们来看看 JVM 在我们整个系统的位置：

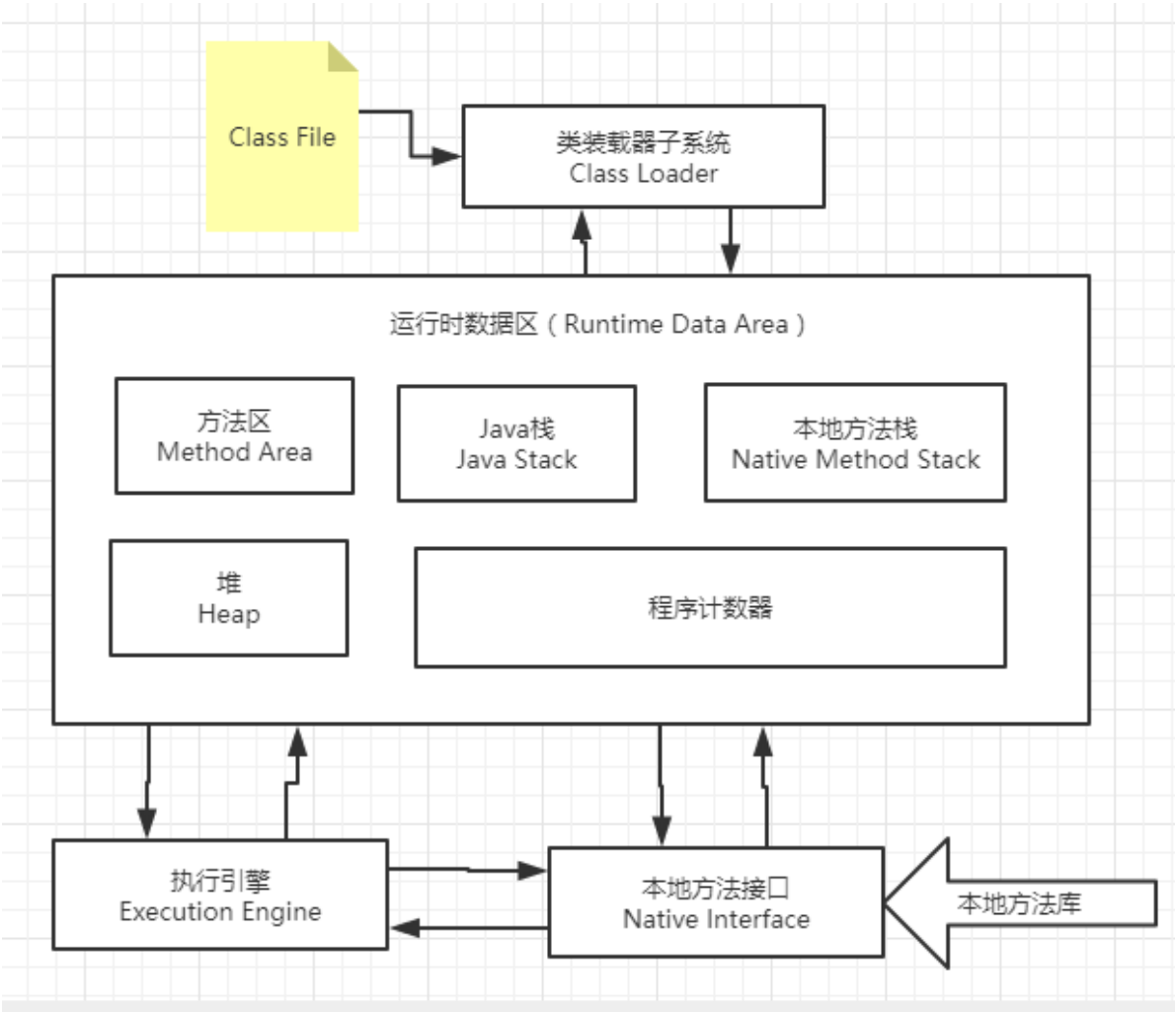


所以要理解一个问题：JVM是运行在操作系统之上的，它与硬件没有直接的交互

假如你的电脑刚买来就有Java环境，那么说明这个电脑已经被人用过了！2333

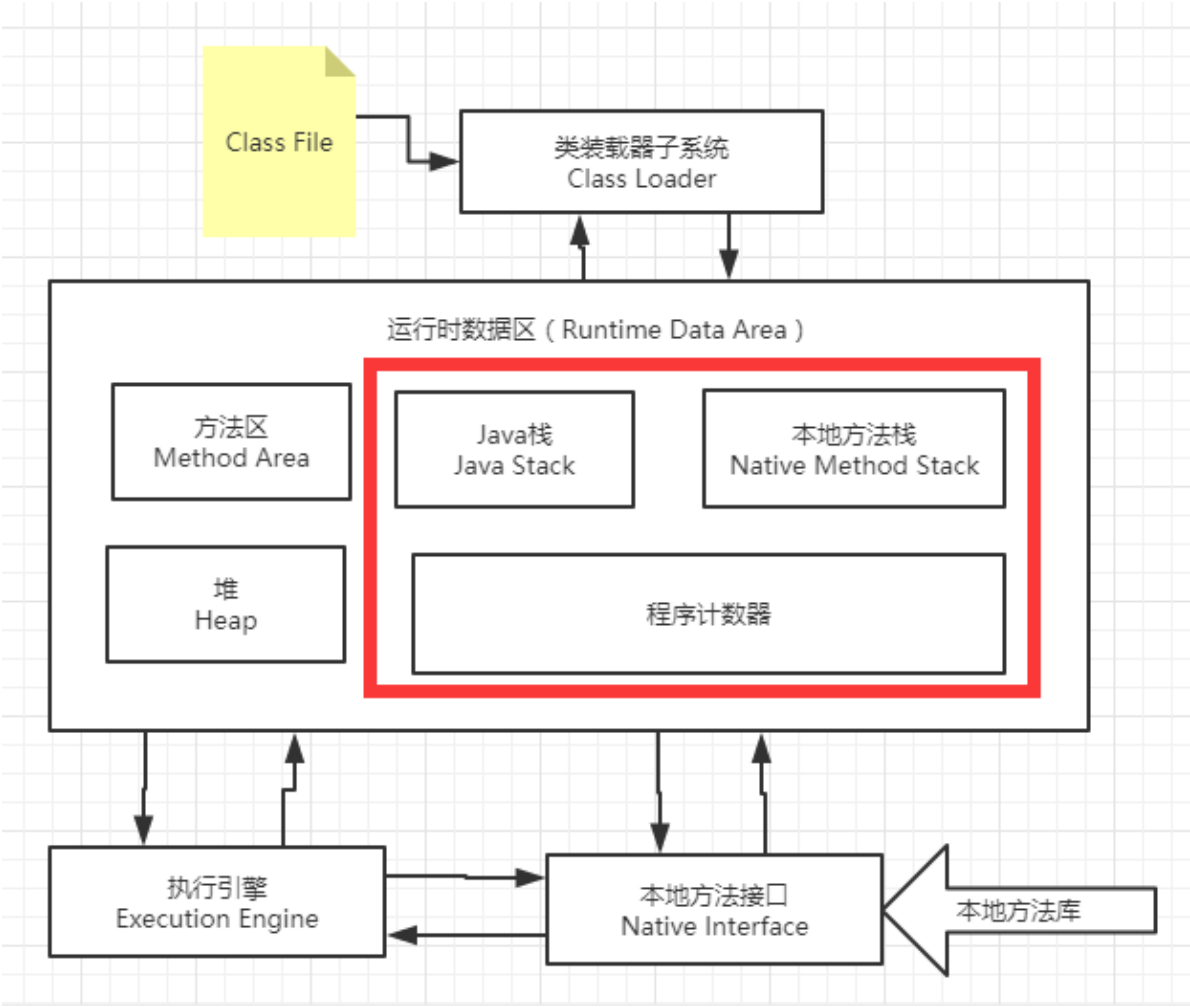
JVM体系结构图

如果你不能够闭着眼睛画出 JVM 的体系结构图，说明你还没有入门 JVM：

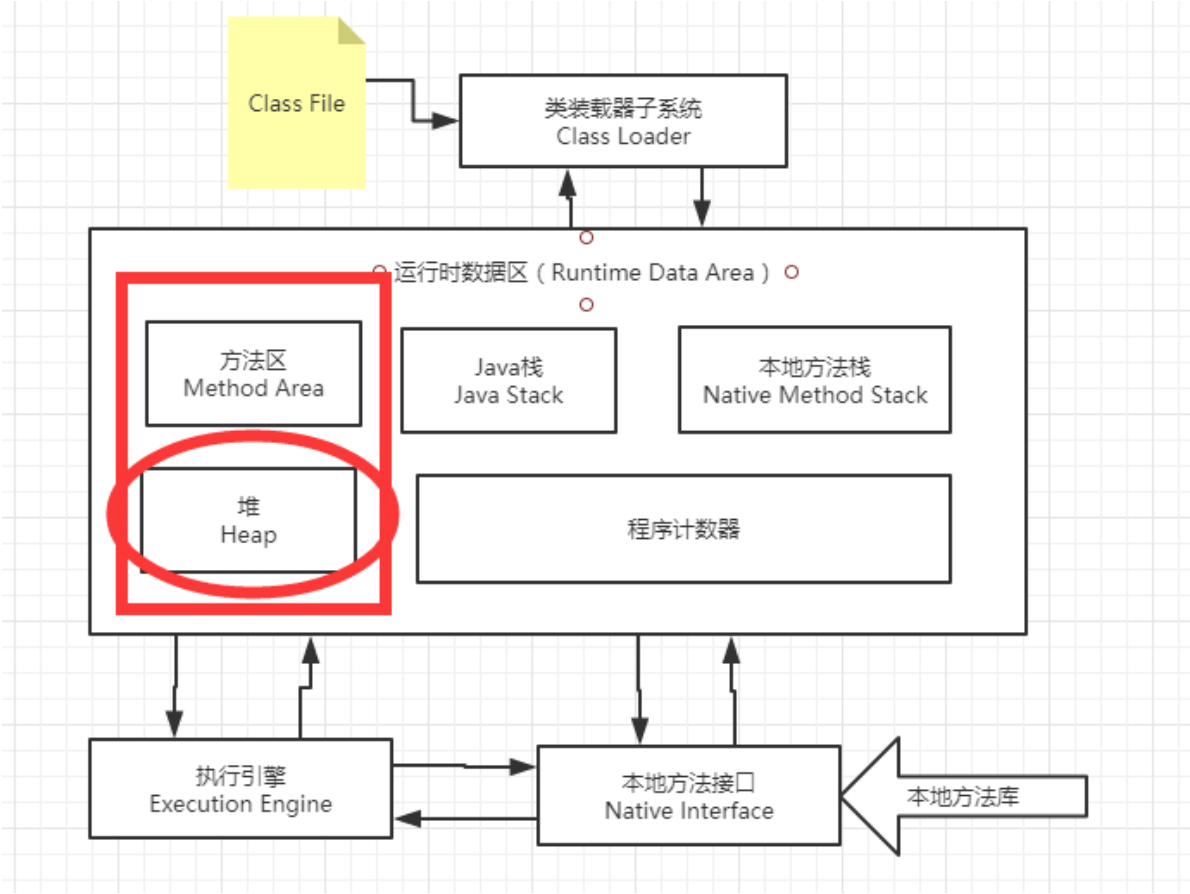


课堂要求：在现场自己画出来这个图，能记下来多少记多少！

分析：这个区域一定不会有垃圾回收



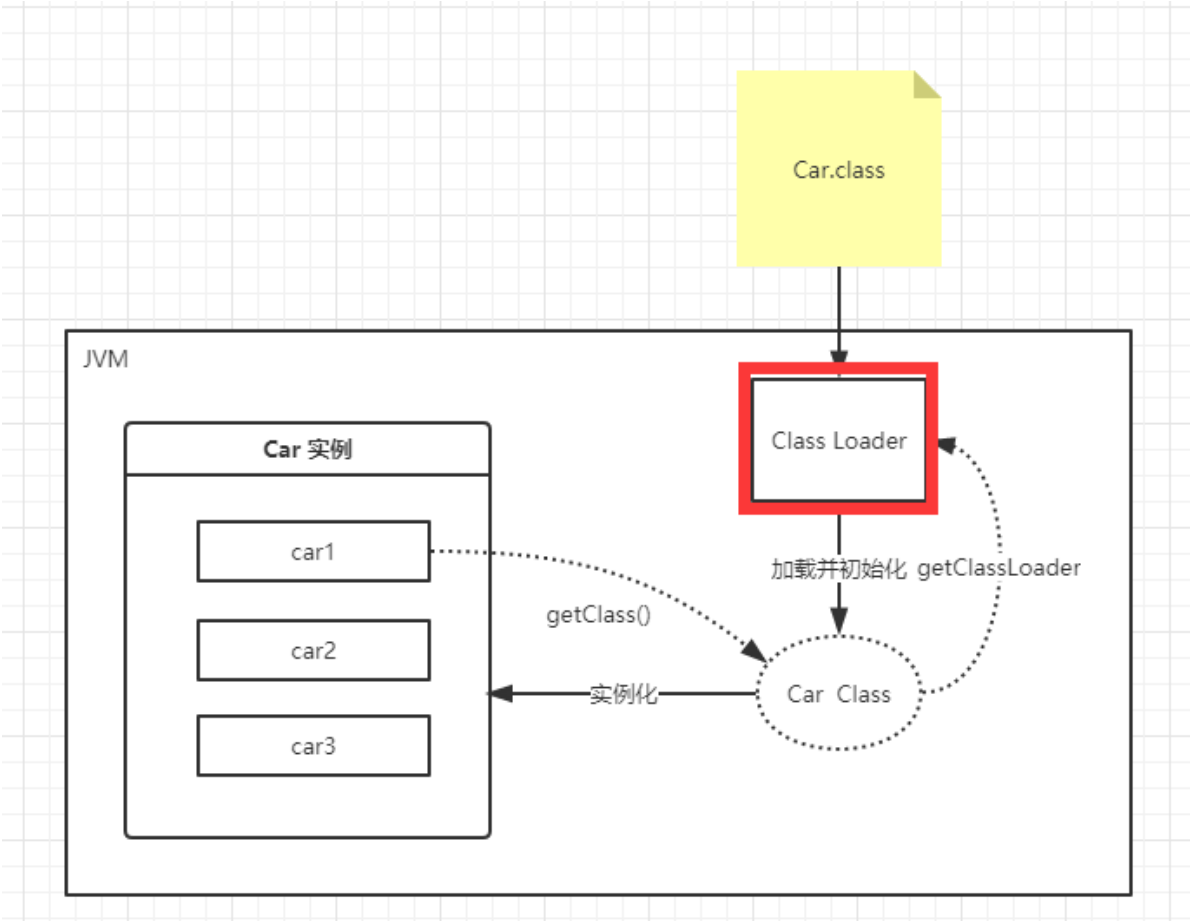
所谓JVM的调优，其实就是在调这个区域，而且99%情况下都在调堆！



在整个 JVM 的学习过程当中，希望大家可以在大脑中一直留着这幅图的印象！

类加载器ClassLoader

我们先来看看一个类加载到 JVM 的一个基本结构：



在如下几种情况下，Java虚拟机将结束生命周期：

1. 执行了System.exit()方法
2. 程序正常执行结束
3. 程序在执行过程中遇到了异常或者错误而异常终止
4. 由于操作系统出现错误而导致Java虚拟机进行终止

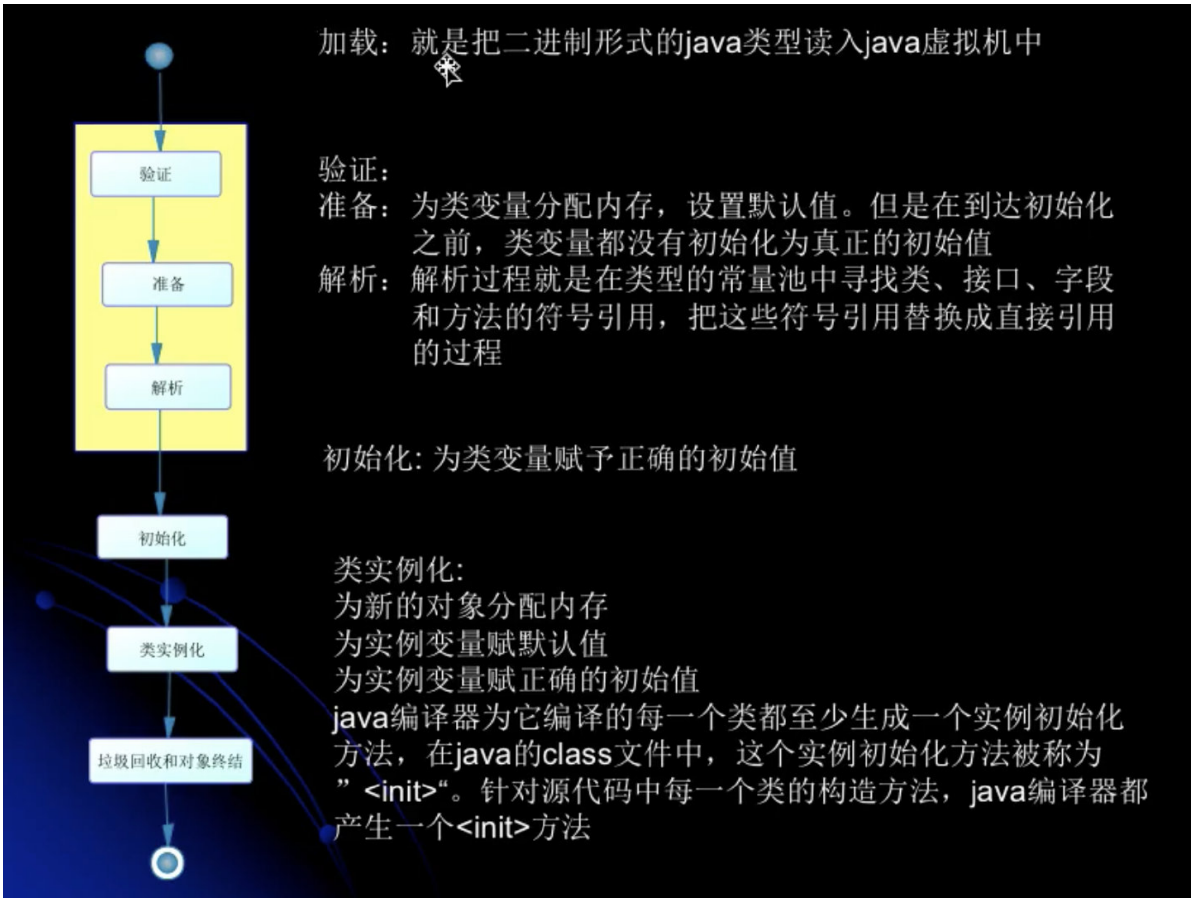
> 类的加载、连接与初始化

在Java代码中，**Class的加载、连接与初始化**过程都是在程序运行期间完成的。Runtime！

- 加载： 查找并加载类的二进制数据
- 连接
  - 验证：确保被加载的类的正确性
  - 准备：为类的静态变量分配内存，并将其初始化为默认值
  - 解析：把类中的符号引用转换为直接引用

    ```
    1 在编译的时候一个每个java类都会被编译成一个class文件，但在编译的时候虚拟机并不知道所引用类的地址，多以就用符号引用来代替，而在这个解析阶段就是为了把这个符号引用转化成为真正的地址的阶段。
    ```

- 初始化：为类的静态变量赋予正确的初始值



从代码来理解：

```
class Test{
  public static int a = 1;
}
//我们程序中给定的是  public static int a = 1;
//但是在加载过程中的步骤如下：

1. 加载阶段
  编译文件为 .class文件，然后通过类加载，加载到JVM

2. 连接阶段
  第一步（验证）：确保Class类文件没问题
  第二步（准备）：先初始化为 a=0。（因为你int类型的初始值为0）
  第三步（解析）：将引用转换为直接引用

3. 初始化阶段：
  通过此解析阶段，把1赋值为变量a
```

> 类的加载

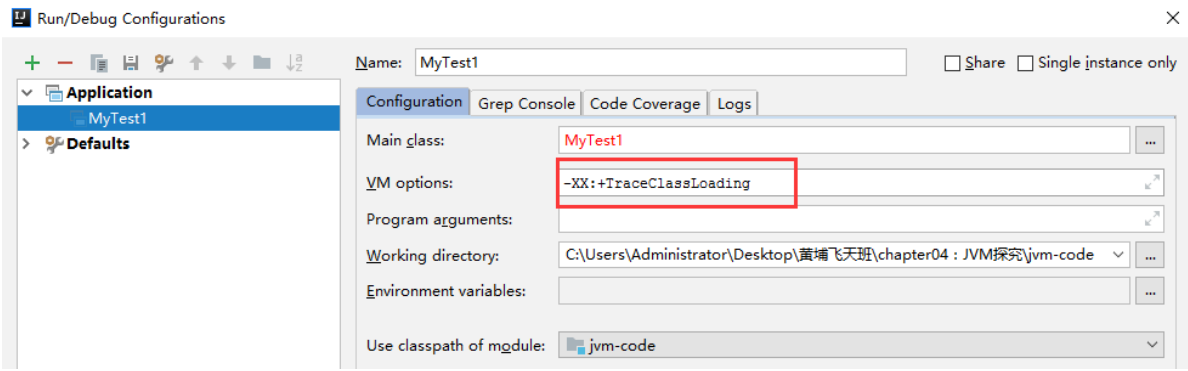
类的加载指的是将类的.class文件中二进制数据读入到内存中，将其放在运行时数据区内的方法区内，然后再内存中创建一个`java.lang.Class`对象用来封装类在方法区内的数据结构。

```
//对于静态字段来说，只有直接定义了该字段的类才会被初始化；
//当一个类在初始化时，要求其父类全部都已经初始化完毕了；
//所有Java虚拟机实现必须在每个类或者接口被Java程序“首次主动使用”时才初始化他们
public class MyTest1 {
    public static void main (String[] args){
        System.out.println(MyChild1.str2);
```

```
    }
}

class MyParent1{
    public static String str = "hello world";
    static {
        System.out.println("MyParent1 static");
    }
}

class MyChild1 extends MyParent1{
    public static String str2 = "welcome";
    static{
        System.out.println("MyChild1 static");
    }
}
// 输出结果:
MyParent1 static block
MyChild1 static block
welcome
```

查看类的加载信息，并打印出来:

```
jvm 参数介绍:
    -XX:+TraceClassLoading,用于追踪类的加载信息并打印出来。

所有的参数都是:
    -XX:+<option> ，  表示开启option选项
    -XX:-<option> ，  表示关闭option选项
    -XX:+<option>=<value>   表示将option选项的值设置为value
```



> 常量池的概念

我们先来看一道题:

```
public class MyTest2{
  public static void main(String[] args){
    System.out.println(MyParent2.str);
  }
}

class MyParent2{
   public static final String str = "hello world";
   static {
```

```
    System.out.println("Myparent2 static block");// 这一行能输出吗？不会
  }
}
/*
常量在编译阶段会存入到调用这个常量的方法所在的类的常量池中
本质上，调用类并没有直接用用到定义常量的类，因此并不会触发定义常量的类的初始化。
注意：这里指的是将常量存放到了MyTest2的常量池中，之后MyTest2与MyParent2就没有任何关系了。
*/
```

再看一道题，类的初始化规则：

```
/*
* 当一个常量的值并非编译期间可以确定的，那么其值就不会被放到调用类的常量池中，
    这是在程序运行时，会导致主动使用这个常量所在的类，显然就会导致这个类被初始化。
*/
public class MyTest3{
  public static void main(String[] args){
    System.out.println(MyParent3.str);
  }
}

class MyParent3{
   public static final String str = UUID.randomUUID().toString();

   static {
    System.out.println("Myparent3 static block");  // 这一行能输出吗？会
  }
}

为什么第二个例子不会输出，第三个例子就输出了呢？
因为第三个例子的值，是只有当运行期才会被确定的值。而第二个例子的值，是编译时就能被确定的值。
```

> ClassLoader分类

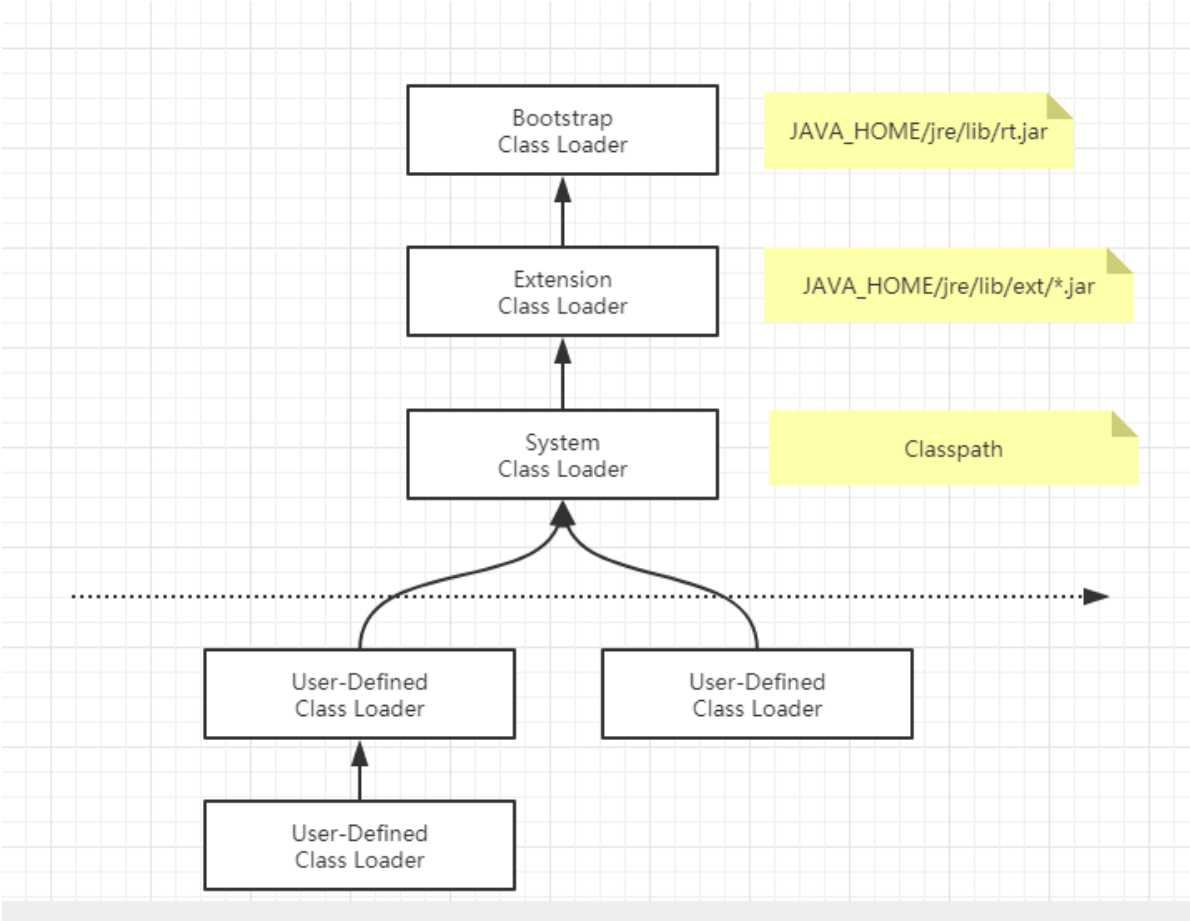
有两种类型的类加载器

1、Java虚拟机自带的加载器

- 根类加载器（BootStrap）(BootClassLoader) sun.boot.class.path （加载系统的包，包含jdk核心库里的类)
- 扩展类加载器（Extension） （ExtClassLoader） java.ext.dirs（加载扩展jar包中的类）
- 系统（应用）类加载器（System）(AppClassLoader) java.class.path（加载你编写的类，编译后的类）

2、用户自定义的类加载器

- Java.long.ClassLoader的子类（继承），用户可以定制类的加载方式



代码测试

```
public class ClassLoaderDemo01 {
    public static void main(String[] args) {
        Object object = new Object();
        ClassLoaderDemo01 demo01 = new ClassLoaderDemo01();

        System.out.println(object.getClass().getClassLoader());
        System.out.println(demo01.getClass().getClassLoader());
        System.out.println(demo01.getClass().getClassLoader().getParent());

 System.out.println(demo01.getClass().getClassLoader().getParent().getParent
());

        /*
        结果:
            null
            sun.misc.Launcher$AppClassLoader@18b4aac2
            sun.misc.Launcher$ExtClassLoader@1b6d3586
            null
         **/
    }
}
```

> 双亲委派机制

双亲委派机制的工作原理：一层一层的 让父类去加载，最顶层父类不能加载往下数，依次类推。

1. 类加载器收到类加载的请求；

2. 把这个请求委托给父加载器去完成，一直向上委托，直到启动类加载器；
3. 启动器加载器检查能不能加载（使用findClass()方法），能就加载（结束）；否则，抛出异常，通知子加载器进行加载。
4. 重复步骤三；

代码测试：

```
package java.lang;

public class String {
    public static void main(String[] args) {
        System.out.println(1);
    }
}
```

大家所熟知的String 类，直接告诉大家，String 默认情况下是启动类加载器进行加载的。假设我也自定义一个String 。现在你会发现自定义的String 可以正常编译，但是永远无法被加载运行。

这是因为申请自定义String 加载时，总是启动类加载器，而不是自定义加载器，也不会是其他的加载器。

双亲委派机制可以确保Java核心类库所提供的类，不会被自定义的类所替代。

Native方法

编写一个多线程类启动

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

    },"your thread name").start();
}
```

点进去看start方法的源码

```
public synchronized void start() {

    if (threadStatus != 0)
        throw new IllegalThreadStateException();

    group.add(this);

    boolean started = false;
    try {
        start0(); //调用了一个start0方法
        started = true;
    } finally {
        try {
            if (!started) {
                group.threadStartFailed(this);
```

```
            }
        } catch (Throwable ignore) {
        }
    }
}

//这个Thread是一个类，这个方法定义在这里是不是很诡异！看这个关键字native：
private native void start0();
```

凡是带了native关键字的，说明 java的作用范围达不到，去调用底层C语言的库！

JNI：Java Native Interface （Java本地方法接口）

凡是带了native关键字的方法就会进入本地方法栈；

Native Method Stack 本地方法栈

本地接口的作用是融合不同的编程语言为Java所用，它的初衷是融合C/C++程序，Java在诞生的时候是C/C++横行的时候，想要立足，必须有调用C、C++的程序，于是就在内存中专门开辟了一块区域处理标记为native的代码，它的具体做法是 在 Native Method Stack 中登记native方法，在 (Execution Engine) 执行引擎执行的时候加载Native Libraies。

目前该方法使用的越来越少了，除非是与硬件有关的应用，比如通过Java程序驱动打印机或者Java系统管理生产设备，在企业级应用中已经比较少见。因为现在的异构领域间通信很发达，比如可以使用Socket通信，也可以使用Web Service等等，不多做介绍！

程序计数器

程序计数器：Program Counter Register

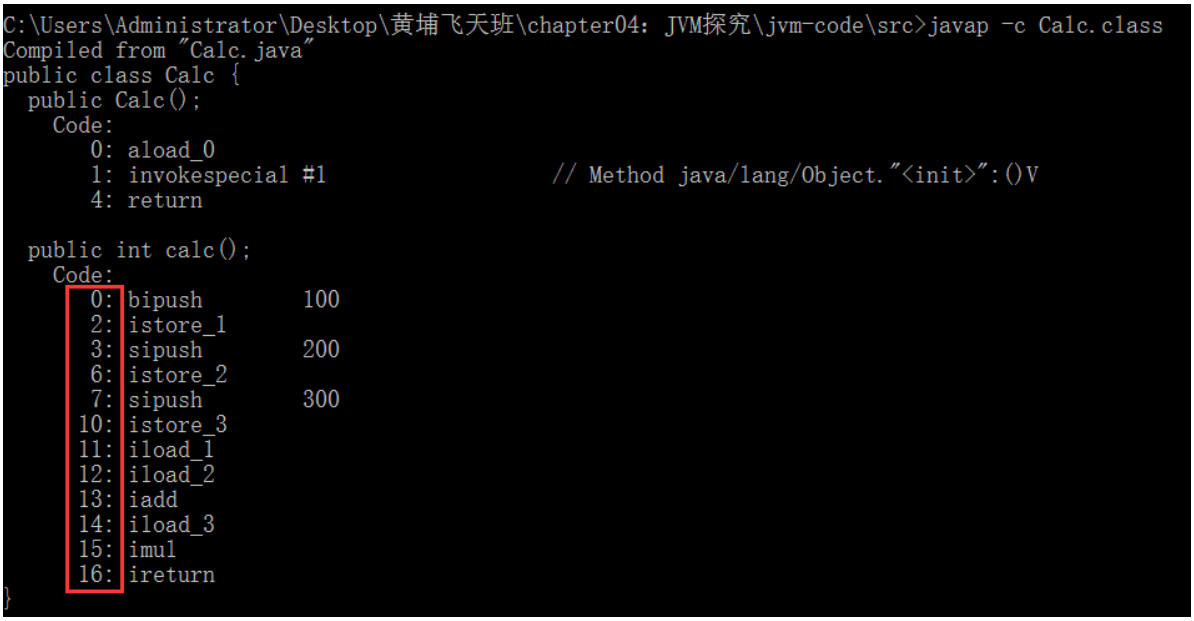
每个线程都有一个程序计数器，是线程私有的。

程序计数器是一块较小的内存空间，它的作用可以看作是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。是一个非常小的内存空间，几乎可以忽略不计

```
public class Calc {
    public int calc(){
        int a = 100;
        int b = 200;
        int c = 300;
        return ( a + b ) * c;
    }
}
```

反编译： `Javap -c xx.class` 反编译之后会有助记符。

- `ldc` 表示将int、float或是String类型的常量值从常量池中推送至栈顶。
- `bipush` 表示将单字节（-128~127）的常量值推送至栈顶。
- `sipush` 表示将短整型(-32767~32768)的常量值推送至栈顶。
- `istore_1` 将一个数值从操作数栈存储到局部变量表
- `iadd` 加
- `imul` 乘



图中使用红框框起来的就是字节码指令的**偏移地址**，偏移地址对应的**bipush** 等等是jvm 中的操作指令，这是入栈指令。 当执行到方法**calc()**时在当前的线程中会创建相应的程序计数器，在计数器中为存放执行地址 （红框中的）0 2 3...等等

方法区

Method Area 方法区 是 Java虚拟机规范 中定义的运行时数据区域之一，它与堆(heap)一样在线程之间共享。

Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区分开来。

JDK7 之前（永久代）用于存储已被虚拟机加载的类信息、常量、字符串常量、类静态变量、即时编译器编译后的代码等数据。每当一个类初次被加载的时候，它的元数据都会被放到永久代中。永久代大小有限制，如果加载的类太多，很可能导致永久代内存溢出，即`java.lang.OutOfMemoryError: PermGen。`

JDK8 彻底将永久代移除出 HotSpot JVM，将其原有的数据迁移至 Java Heap 或 Native Heap（Metaspace），取代它的是另一个内存区域被称为元空间（Metaspace）。

元空间（Metaspace）：元空间是方法区的在 HotSpot JVM 中的实现，方法区主要用于存储类信息、常量池、方法数据、方法代码、符号引用等。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。

可以通过`-XX:MetaspaceSize`和`-XX:MaxMetaspaceSize`配置内存大小。

如果Metaspace的空间占用达到了设定的最大值，那么就会触发GC来收集死亡对象和类的加载器。

栈（Stack）

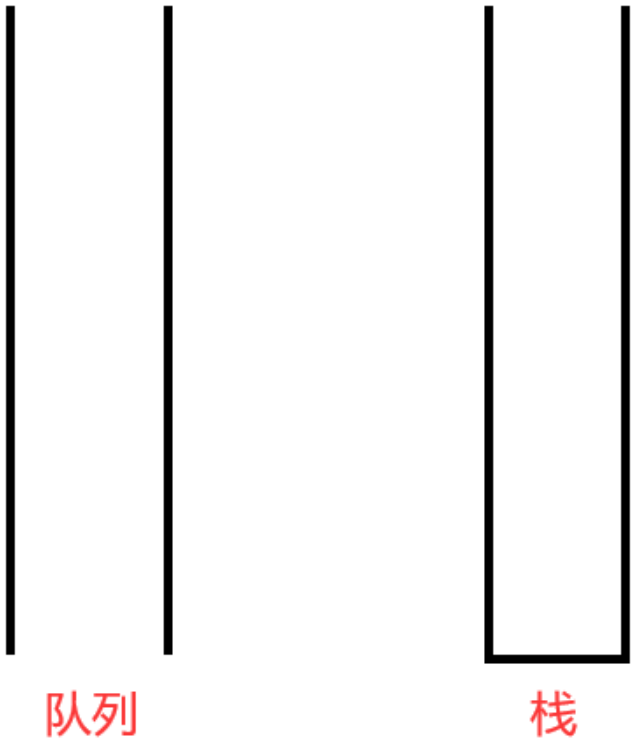
> 栈和队列

在计算机流传有一句废话： 程序 = 算法 + 数据结构

但是对于大部分同学都是： 程序 = 框架 + 业务逻辑

栈：后进先出 / 先进后出

队列：先进先出（FIFO : First Input First Output）



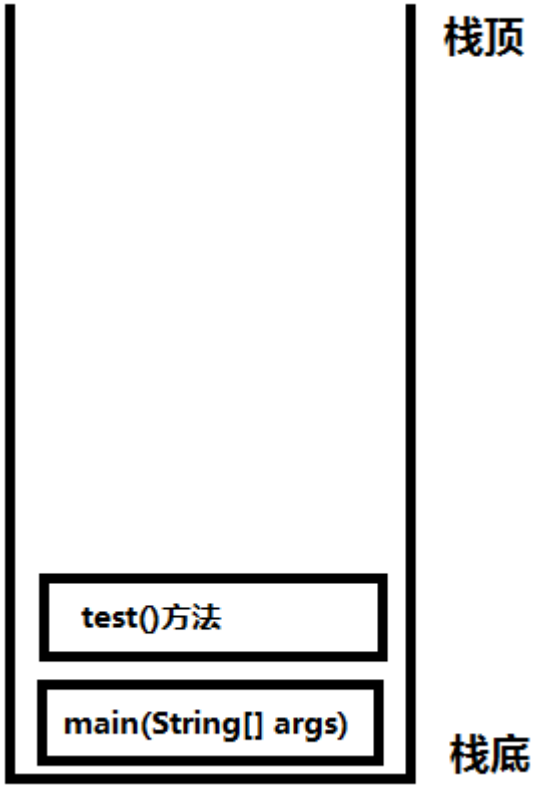
> Stack 栈是什么

栈管理程序运行

存储一些基本类型的值、对象的引用、方法等。

栈的优势是，存取速度比堆要快，仅次于寄存器，栈数据可以共享。

思考：为什么main方法最后执行！为什么一个test() 方法执行完了，才会继续走main方法！



喝多了吐就是栈，吃多了拉就是队列

说明：

1、栈也叫栈内存，主管Java程序的运行，是在线程创建时创建，它的生命期是跟随线程的生命期，线程结束栈内存也就释放。

2、**对于栈来说不存在垃圾回收问题**，只要线程一旦结束，该栈就Over，生命周期和线程一致，是线程私有的。

3、方法自己调自己就会导致栈溢出（递归死循环测试）

```
public class StackDemo {
    public static void main(String[] args) {
        a();
    }
    public static void a(){
        b();
    }
    public static void b(){
        a();
    }
}
```

栈运行原理

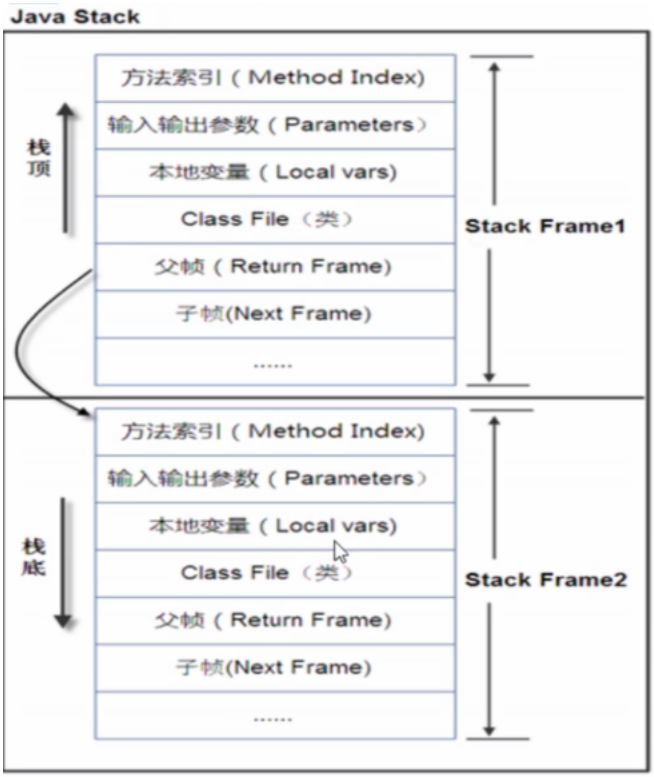
Java栈的组成元素—栈帧

栈帧是一种用于帮助虚拟机执行方法调用与方法执行的数据结构。他是独立于线程的，一个线程有自己的一个栈帧。封装了方法的局部变量表、动态链接信息、方法的返回地址以及操作数栈等信息。

第一个方法从调用开始到执行完成，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

当一个方法A被调用时就产生了一个栈帧F1，并被压入到栈中，A方法又调用了B方法，于是产生了栈帧F2也被压入栈中，B方法又调用了C方法，于是产生栈帧F3也被压入栈中.....执行完毕后，先弹出F3，然后弹出F2，在弹出F1.....

遵循 “先进后出” / "后进先出" 的原则。



左图中存在两个栈帧

栈帧 2是最先被调用的方法，先入栈，

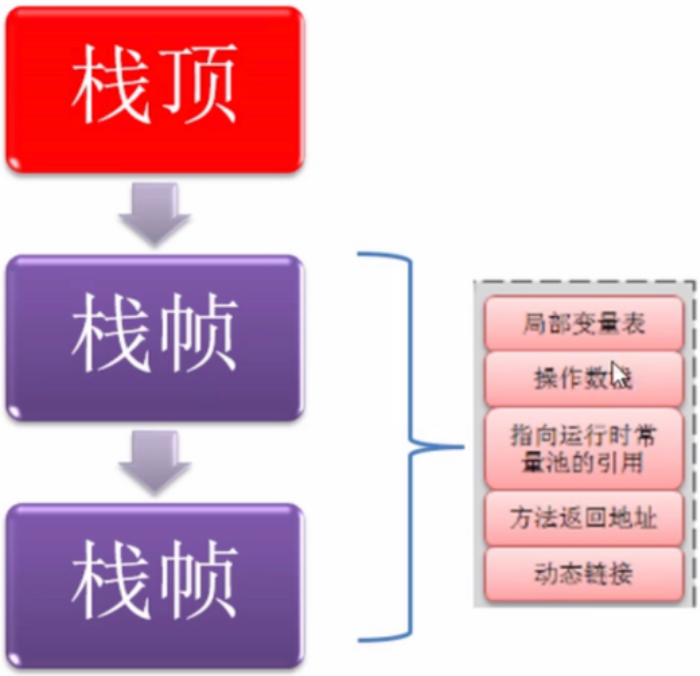
然后方法 2 又调用了方法1，栈帧 1处于栈顶的位置，

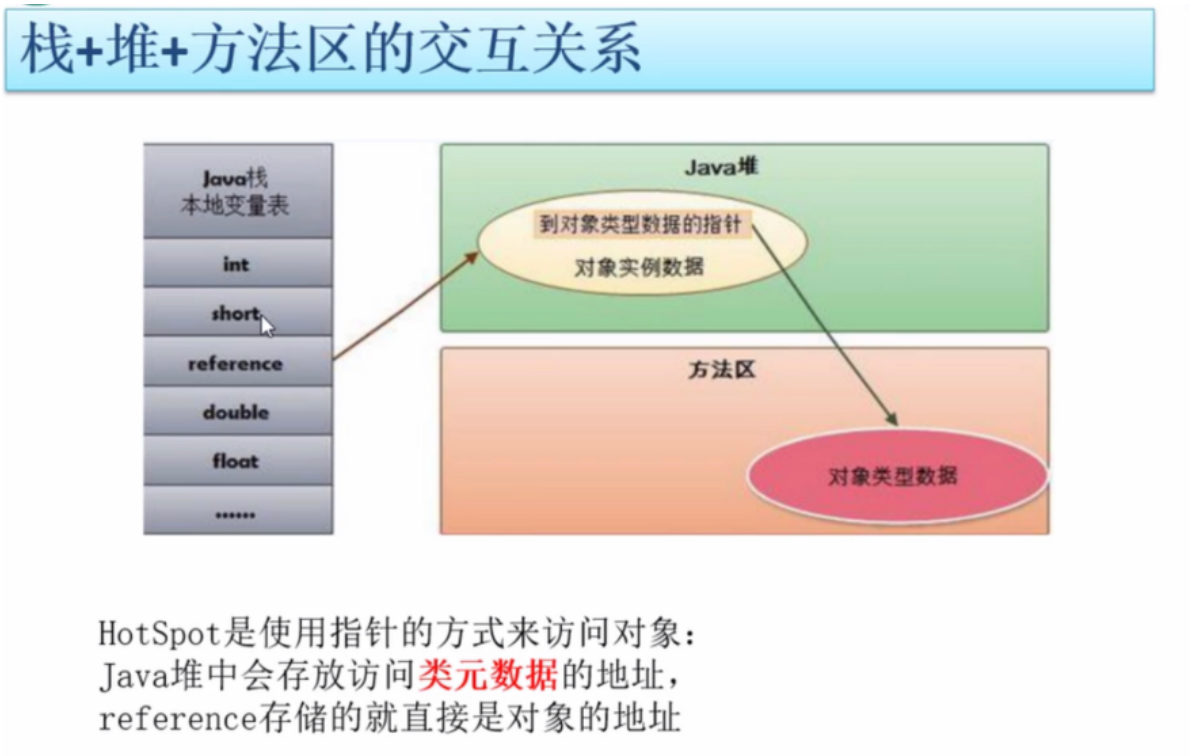
栈帧 2 处于栈底，执行完毕后，依次弹出栈帧 1和栈帧 2，

线程结束，栈释放。

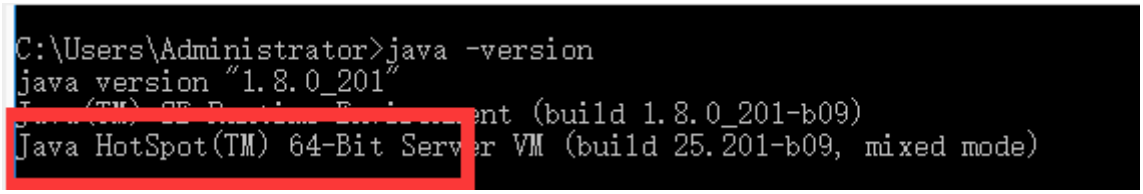
每执行一个方法都会产生一个栈帧，保存到栈(后进先出)的**项部，顶部栈就是当前的方法，该方法执行完毕 后会自动将此栈帧出栈。**

Exception in thread "main" java.lang.StackOverflowError





什么是HotSpot?



了解三种JVM：

- Sun公司的 HotSpot
- BEA公司的 JRockit
- IBM公司的 J9VM

堆（Heap）

Java7之前

Heap 堆，一个JVM实例只存在一个堆内存，堆内存的大小是可以调节的，类加载器读取了类文件后，需要把类，方法，常变量放到堆内存中，保存所有引用类型的真实信息，以方便执行器执行，堆内存分为三部分：

- 新生区 Young Generation Space Young/New
- 养老区 Tenure generation space Old/Tenure
- 永久区 Permanent Space Perm

堆内存逻辑上分为三部分：新生，养老，永久（元空间 : JDK8 以后名称）

GC垃圾回收主要是在 新生区和养老区，又分为 轻GC 和 重GC，如果内存不够，或者存在死循环，就会导致`java.lang.OutOfMemoryError: Java heap space`

> 新生区

新生区是类诞生，成长，消亡的区域，一个类在这里产生，应用，最后被垃圾回收器收集，结束生命。

新生区又分为两部分：伊甸区（Eden Space）和幸存者区（Survivor Space），所有的类都是在伊甸区被new出来的，幸存区有两个：0区 和 1区，当伊甸园的空间用完时，程序又需要创建对象，JVM的垃圾回收器将对伊甸园区进行垃圾回收（Minor GC）。将伊甸园中的剩余对象移动到幸存0区，若幸存0区也满了，再对该区进行垃圾回收，然后移动到1区，那如果1区也满了呢？（这里幸存0区和1区是一个互相交替的过程）再移动到养老区，若养老区也满了，那么这个时候将产生MajorGC（Full GC），进行养老区的内存清理，若养老区执行了Full GC后发现依然无法进行对象的保存，就会产生OOM异常“OutOfMemoryError”。

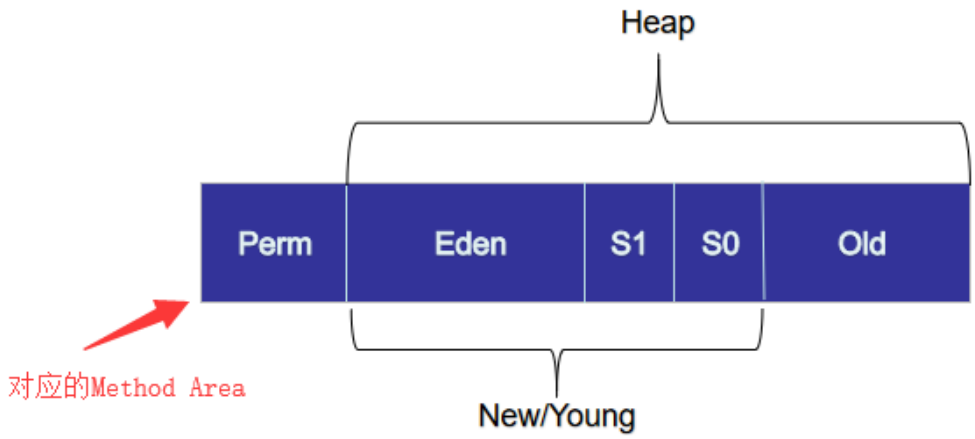
如果出现 java.lang.OutOfMemoryError：java heap space异常，说明Java虚拟机的堆内存不够，原因如下：

1、Java虚拟机的堆内存设置不够，可以通过参数 -Xms（初始值大小），-Xmx（最大大小）来调整。

2、代码中创建了大量大对象，并且长时间不能被垃圾收集器收集（存在被引用）或者死循环

> Sun HotSpot内存管理

分代管理，不同的区域使用不同的算法：



Why？真相：经过研究，不同对象的生命周期不同，在Java中98%的对象都是临时对象。

> 永久区（Perm）

永久存储区是一个常驻内存区域，用于存放JDK自身所携带的Class，Interface的元数据，也就是说它存储的是运行环境必须的类信息，被装载进此区域的数据是不会被垃圾回收器回收掉的，关闭JVM才会释放此区域所占用的内存。

如果出现 java.lang.OutOfMemoryError：PermGen space，说明是 Java虚拟机对永久代Perm内存设置不够。一般出现这种情况，都是程序启动需要加载大量的第三方jar包，例如：在一个Tomcat下部署了太多的应用。或者大量动态反射生成的类不断被加载，最终导致Perm区被占满。

注意：

Jdk1.6之前：　有永久代，常量池1.6在方法区

Jdk1.7：　　有永久代，但是已经逐步 “去永久代”，常量池1.7在堆

Jdk1.8及之后：无永久代，常量池1.8在元空间

熟悉三区结构后方可学习JVM垃圾回收机制

实际而言，方法区（Method Area）和堆一样，是各个线程共享的内存区域，它用于存储虚拟机加载的：类信息+普通常量+静态常量+编译器编译后的代码，**虽然JVM规范将方法区描述为堆的一个逻辑部分，但它却还有一个别名，叫做Non-Heap（非堆），目的就是要和堆分开。**

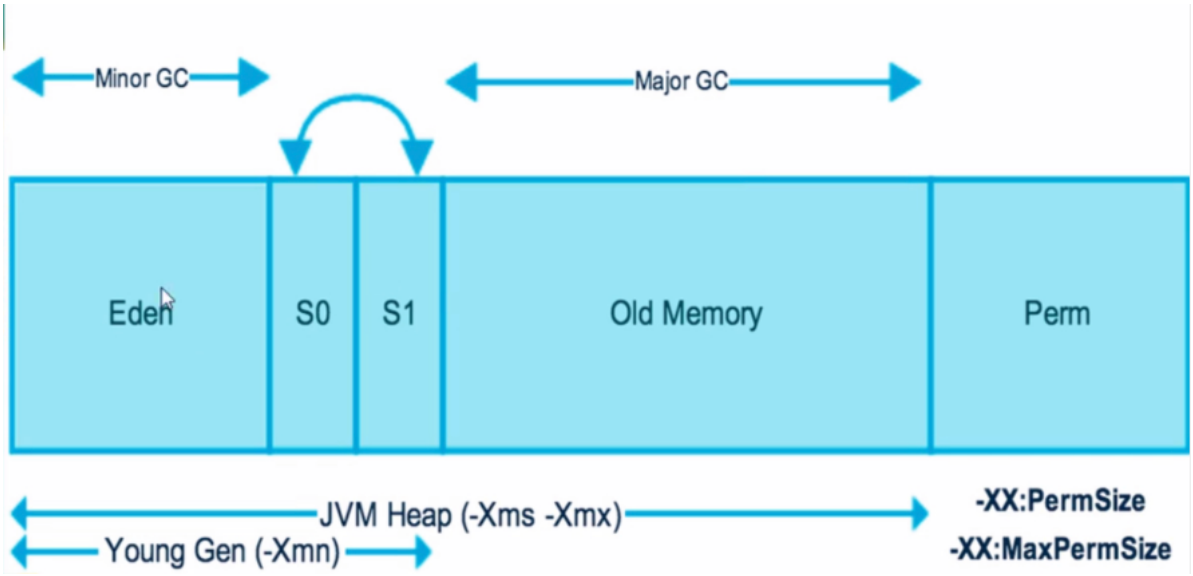
对于HotSpot虚拟机，很多开发者习惯将方法区称之为 “永久代（Parmanent Gen）”，但严格本质上说两者不同，或者说使用永久代实现方法区而已，永久代是方法区（相当于是一个接口interface）的一个实现，Jdk1.7的版本中，已经将原本放在永久代的字符串常量池移走。

常量池（Constant Pool）是方法区的一部分，Class文件除了有类的版本，字段，方法，接口描述信息外，还有一项信息就是常量池，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！

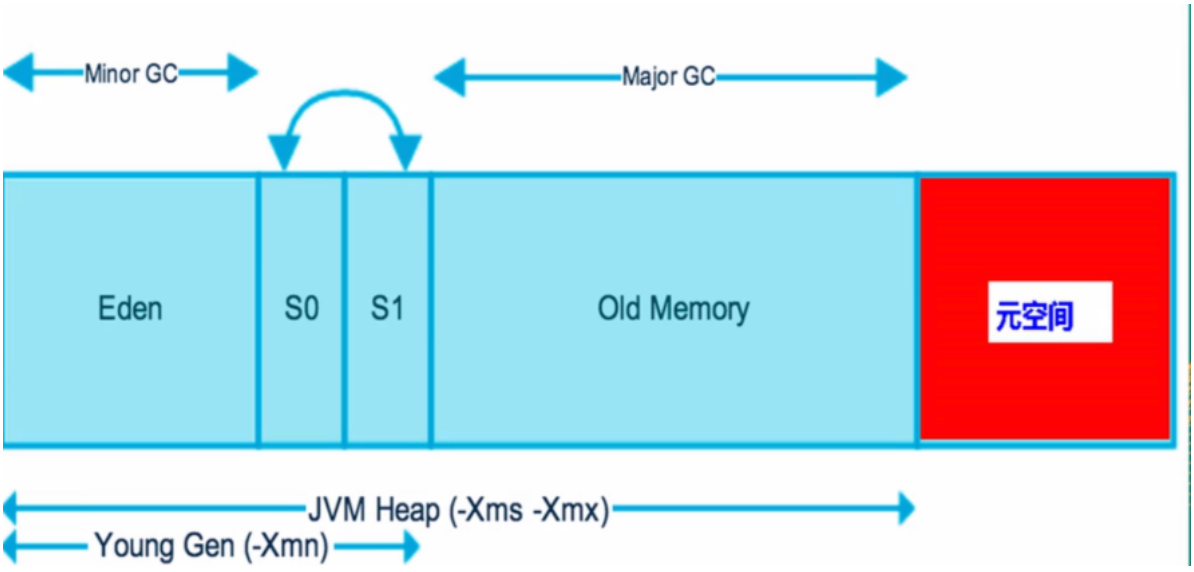
堆内存调优

了解完基本的堆的信息之后，我们就可以简单学习下关于堆内存调优的说明了！我们是基于 HotSpot 虚拟机的，JDK1.8；

> 先看下 JDK 1.7 的 和 1.8 的区别



> JDK 1.8 的



> 使用 IDEA 调整堆内存大小测试

堆内存调优

`-Xms`：设置初始分配大小，默认为物理内存的“1/64”

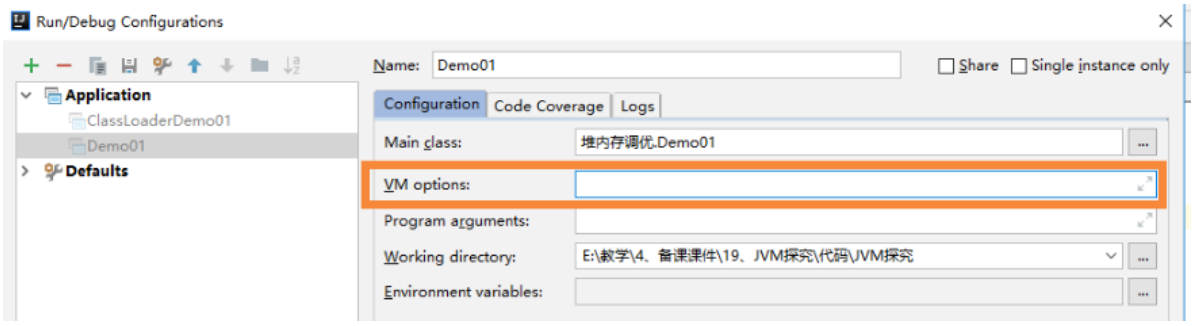
`-Xmx`：最大分配内存，默认为物理内存的“1/4”

`-XX:+PrintGCDetails`：输出详细的GC处理日志

代码测试

```java
public class Demo01 {
    public static void main(String[] args) {
        //返回Java虚拟机试图使用的最大内存量
        long maxMemory = Runtime.getRuntime().maxMemory();
        //返回Java虚拟机中的内存总量
        long totalMemory = Runtime.getRuntime().totalMemory();

        System.out.println("MAX_MEMORY="+maxMemory+"(字节)、"
                           +(maxMemory/(double)1024/1024)+"MB");
        System.out.println("TOTAL_MEMORY="+totalMemory+"(字节)、"
                           +(totalMemory/(double)1024/1024)+"MB");
    }
}
```

IDEA中进行JVM调优参数设置，然后启动



发现，默认的情况下分配的内存是总内存的 1/4，而初始化的内存为 1/64！

```
-Xms1024m -Xmx1024m -XX:+PrintGCDetails
```

VM参数调优：把初始内存，和总内存都调为 1024M，运行，查看结果！

```
D:\Environment\java\jdk1.8.0_201\bin\java.exe ...
MAX_MEMORY=1029177344(字节)、981.5MB
TOTAL_MEMORY=1029177344(字节)、981.5MB
Heap
 PSYoungGen      total 305664K, used 20971K [0x00000000eab00000, 0x0000000100000000, 0x0000000100000000)
  eden space 262144K, 8% used [0x00000000eab00000,0x00000000ebf7afb8,0x00000000fab00000)
  from space 43520K, 0% used [0x00000000fd580000,0x00000000fd580000,0x0000000100000000)
  to   space 43520K, 0% used [0x00000000fab00000,0x00000000fab00000,0x00000000fd580000)
 ParOldGen       total 699392K, used 0K [0x00000000c0000000, 0x00000000eab00000, 0x00000000eab00000)
  object space 699392K, 0% used [0x00000000c0000000,0x00000000c0000000,0x00000000eab00000)
 Metaspace       used 3242K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 351K, capacity 388K, committed 512K, reserved 1048576K
```

我们来大概计算分析一下！

```
MAX_MEMORY=1029177344(字节)、981.5MB
TOTAL_MEMORY=1029177344(字节)、981.5MB
Heap
 PSYoungGen      total 305664K, used 20971K [0x00000000eab00000, 0x0000000100000000, 0x0000000100000000)
  eden space 262144K, 8% used [0x00000000eab00000,0x00000000ebf7afb8,0x0000000fab00000)
  from space 43520K, 0% used [0x0000000fd580000,0x0000000fd580000,0x0000000100000000)
  to   space 43520K, 0% used [0x0000000fab00000,0x0000000fab00000,0x0000000fd580000)
 ParOldGen       total 699392K, used 0K [0x00000000c0000000, 0x00000000eab00000, 0x00000000eab00000)
  object space 699392K, 0% used [0x00000000c0000000,0x00000000c0000000,0x00000000eab00000)
 Metaspace       used 3242K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 351K, capacity 388K, committed 512K, reserved 1048576K
```

305664/1024 + 699392/1024 =981.5

再次证明：元空间并不在虚拟机中，而是使用本地内存。

> 测试二

代码：

```java
public class Demo02 {
    public static void main(String[] args) {
        String str = "kuangShenSayJava";
        while (true){
            str += str + new Random().nextInt(88888888)
                +new Random().nextInt(999999999);
        }
    }
}
```

vm参数：

```
-Xms8m -Xmx8m -XX:+PrintGCDetails
```

测试，查看结果！

```
D:\Environment\java\jdk1.8.0_201\bin\java.exe ...
[GC (Allocation Failure) [PSYoungGen: 1536K->488K(2048K)] 1536K->624K(7680K), 0.0037272 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
[GC (Allocation Failure) [PSYoungGen: 2018K->507K(2048K)] 2154K->935K(7680K), 0.0007673 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 1853K->503K(2048K)] 2281K->1346K(7680K), 0.0006728 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 1586K->320K(2048K)] 3483K->2744K(7680K), 0.0006516 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
[Full GC (Ergonomics) [PSYoungGen: 876K->0K(2048K)] [ParOldGen: 5586K->3258K(5632K)] 6462K->3258K(7680K), [Metaspace: 3229K->3229K(1056768K)], 0.0067
[GC (Allocation Failure) [PSYoungGen: 1114K->32K(2048K)] 5426K->5398K(7680K), 0.0005028 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Ergonomics) [PSYoungGen: 32K->0K(2048K)] [ParOldGen: 5366K->2730K(5632K)] 5398K->2730K(7680K), [Metaspace: 3230K->3230K(1056768K)], 0.0065
[GC (Allocation Failure) [PSYoungGen: 29K->0K(2048K)] 4868K->4838K(7680K), 0.0002976 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Ergonomics) [PSYoungGen: 0K->0K(2048K)] [ParOldGen: 4838K->3784K(5632K)] 4838K->3784K(7680K), [Metaspace: 3230K->3230K(1056768K)], 0.0023
[GC (Allocation Failure) [PSYoungGen: 0K->0K(2048K)] 3784K->3784K(7680K), 0.0002214 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Allocation Failure) [PSYoungGen: 0K->0K(2048K)] [ParOldGen: 3784K->3764K(5632K)] 3784K->3764K(7680K), [Metaspace: 3230K->3230K(1056768K)],
Heap
 PSYoungGen      total 2048K, used 61K [0x00000000ffd80000, 0x0000000100000000, 0x0000000100000000)
  eden space 1536K, 3% used [0x00000000ffd80000,0x00000000ffd8f418,0x00000000fff00000)
  from space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000)
  to   space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000)
 ParOldGen       total 5632K, used 3764K [0x00000000ff800000, 0x00000000ffd80000, 0x00000000ffd80000)
  object space 5632K, 66% used [0x00000000ff800000,0x00000000ffbad278,0x00000000ffd80000)
 Metaspace       used 3261K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 353K, capacity 388K, committed 512K, reserved 1048576K
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOf(Arrays.java:3332)
    at java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBuilder.java:124)
    at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:674)
    at java.lang.StringBuilder.append(StringBuilder.java:208)
    at 堆内存调优.Demo02.main(Demo02.java:10)

Process finished with exit code 1
```

这是一个young 区域撑爆的JAVA 内存日志，其中 PSYoungGen 表示 youngGen分区的变化

1536k 表示 GC 之前的大小。

488k 表示GC 之后的大小。

整个Young区域的大小从 1536K 到 624K , young代的总大小为 7680K。

```
[Times: user=0.02 sys=0.00, real=0.01 secs]
```

user – 总计本次 GC 总线程所占用的总 CPU 时间

sys – OS 调用 or 等待系统时间

real – 应用暂停时间

如果GC 线程是 Serial Garbage Collector 串行搜集器的方式的话（只有一条GC线程,）， real time 等于user 和 system 时间之和。
通过日志发现Young的区域到最后 GC 之前后都是0，old 区域 无法释放，最后报堆溢出错误。

Dump内存快照

在运行java程序的时候，有时候想测试运行时占用内存情况，这时候就需要使用测试工具查看了。在eclipse里面有 **Eclipse Memory Analyzer tool(MAT)**插件可以测试，而在idea中也有这么一个插件，就是**JProfiler**，一款性能瓶颈分析工具!

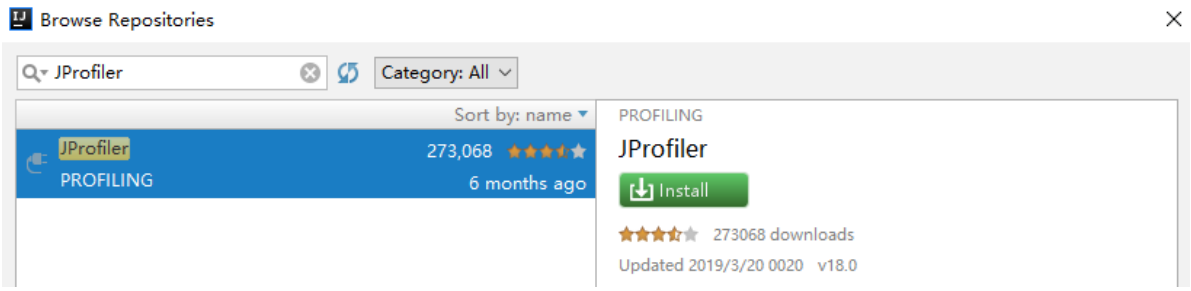
作用：

- 分析Dump文件，快速定位内存泄漏；
- 获得堆中对象的统计数据
- 获得对象相互引用的关系
- 采用树形展现对象间相互引用的情况
- ……

而且这个软件跨平台：

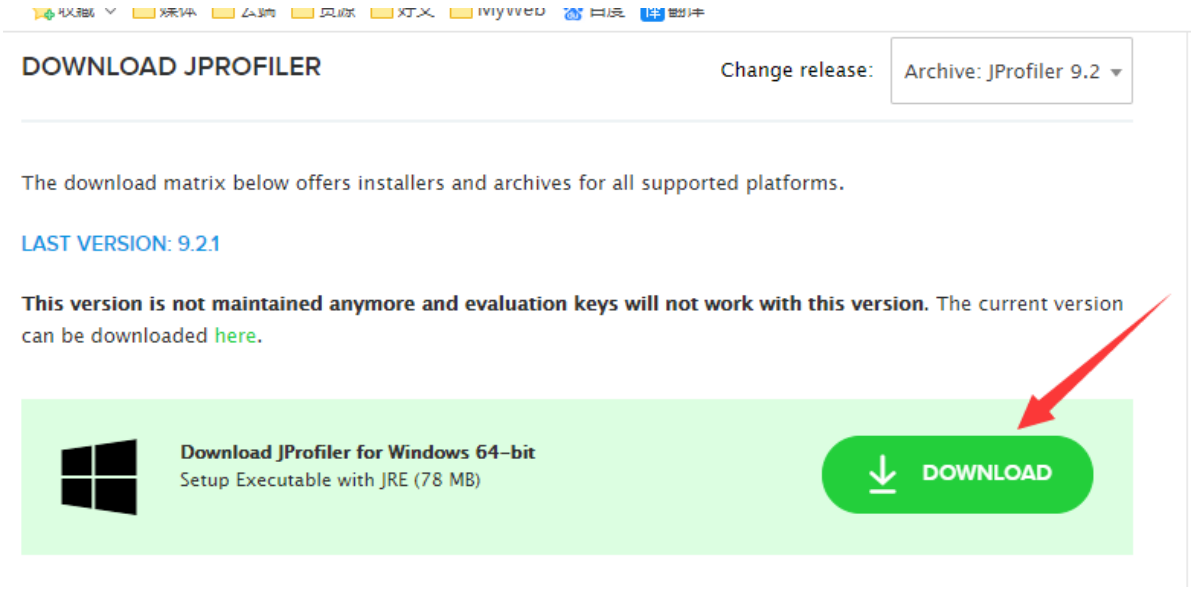
> 安装JProfiler

1、IDEA插件安装



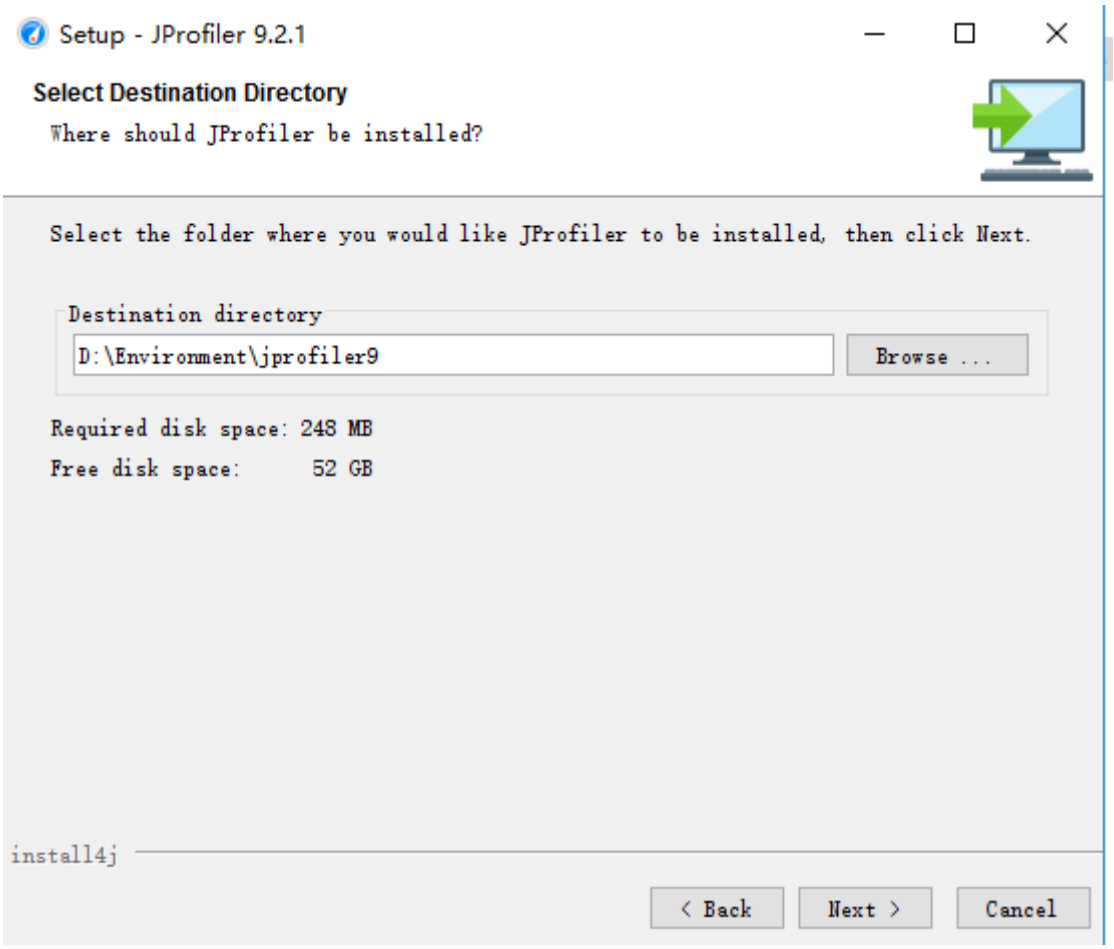
2、安装JProfiler监控软件

下载地址：https://www.ej-technologies.com/download/jprofiler/version_92



3、下载完双击运行，选择自定义目录安装，点击Next

注意：安装路径，**建议选择一个文件名中没有中文，没有空格的路径**，否则识别不了。然后一直点Next

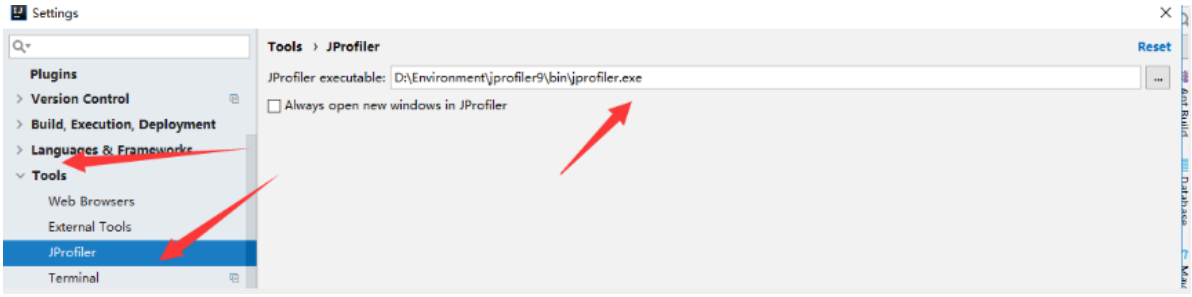


4、注册

```
// 注册码仅供大家参考
L-Larry_Lau@163.com#23874-hrwpdp1sh1wrn#0620
L-Larry_Lau@163.com#36573-fdkscp15axjj6#25257
L-Larry_Lau@163.com#5481-ucjn4a16rvd98#6038
L-Larry_Lau@163.com#99016-hli5ay1ylizjj#27215
L-Larry_Lau@163.com#40775-3wle0g1uin5c1#0674
```

5、配置IDEA运行环境

*Settings-Tools-JProfiler-JProfiler executable*选择JProfile安装可执行文件。（如果系统只装了一个版本，启动IDEA时会默认选择）保存



6、选择你要分析的项目，点击JProfiler图标启动， 启动完成会自动弹出JProfiler窗口，在里面就可以监控自己的代码性能了。

> 代码测试

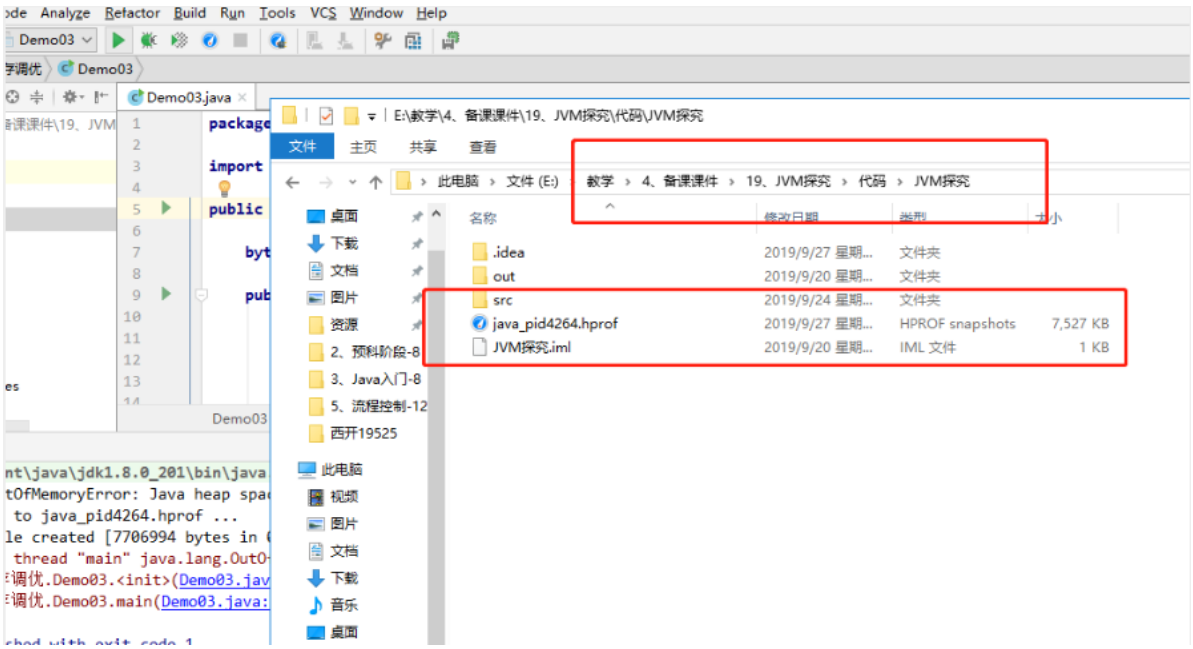
```java
public class Demo03 {

    byte[] byteArray = new byte[1*1024*1024]; //1M = 1024K

    public static void main(String[] args) {
        ArrayList<Demo03> list = new ArrayList<>();
        int count = 0;
        try {
            while (true){
                list.add(new Demo03());
                count = count + 1;
            }
        }catch (Error e){
            System.out.println("count:"+count);
            e.printStackTrace();
        }
    }
}
```

vm参数： `-Xms1m -Xmx8m -XX:+HeapDumpOnOutOfMemoryError`

运行结果：

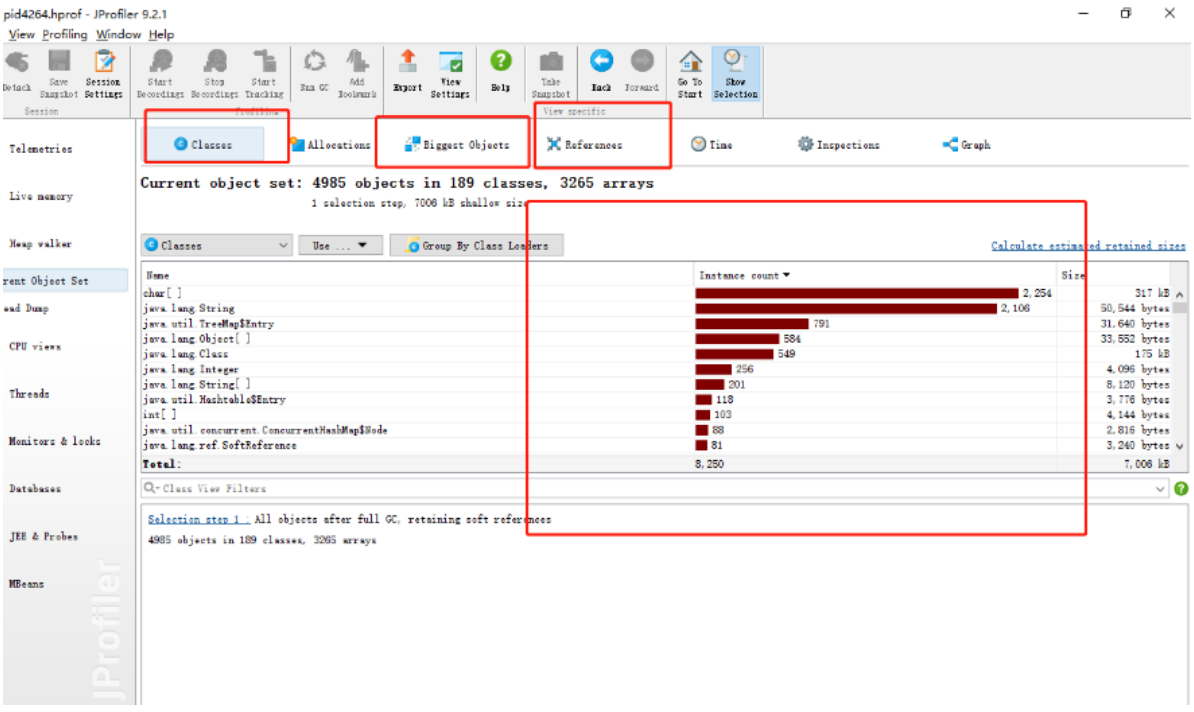
```
D:\Environment\java\jdk1.8.0_201\bin\java.exe ...
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid3476.hprof ...
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
Heap dump file created [7707272 bytes in 0.021 secs]
    at 堆内存调优.Demo03.<init>(Demo03.java:7)
    at 堆内存调优.Demo03.main(Demo03.java:14)

Process finished with exit code 1
```

寻找文件：

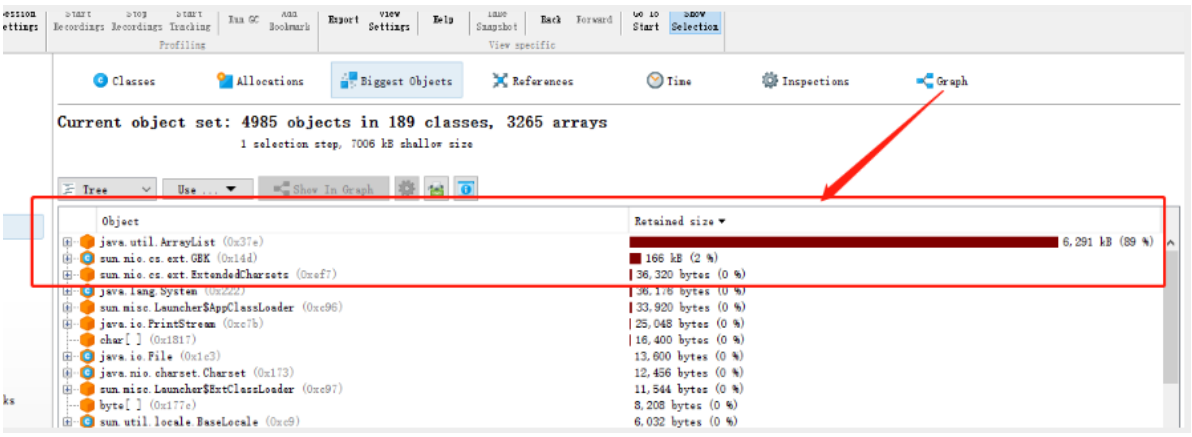


> 使用 Jprofiler 工具分析查看

双击这个文件默认使用 Jprofiler 进行 Open



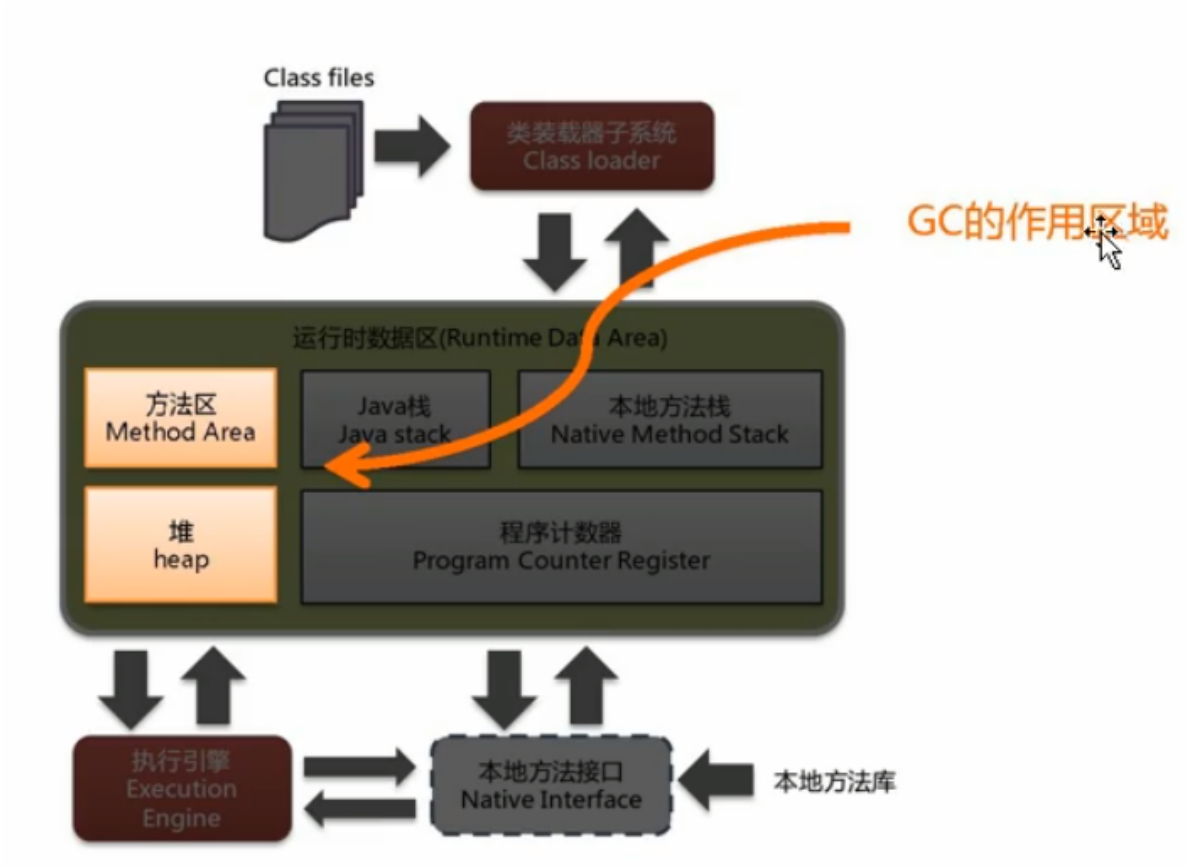
大的对象!



具体的 Jprofiler 使用参考：https://www.cnblogs.com/jpfss/p/8488111.html

GC详解

> 回顾一下 GC 的作用域

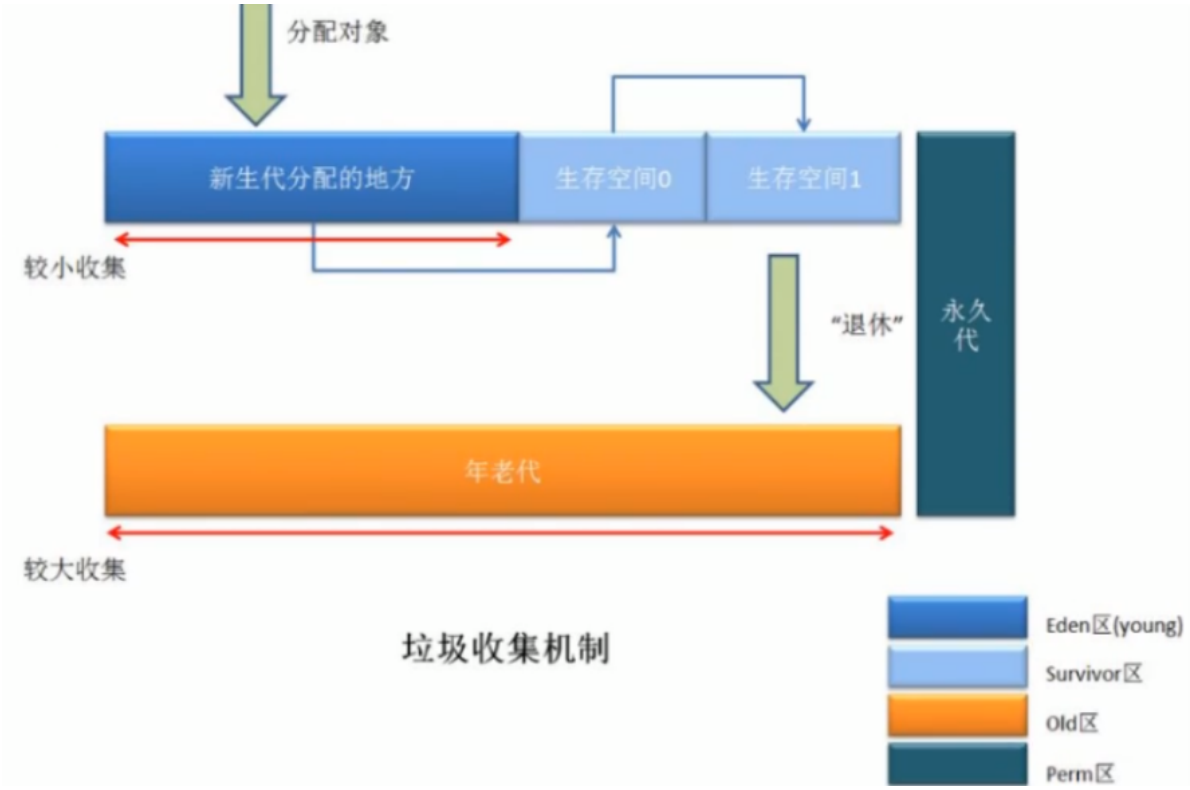


记住GC口诀：分代收集算法

次数频繁Young区，次数较少Old区，基本不动Perm（永久区）区

> GC算法总体概述

先看下一个对象的历程：



JVM 在进行GC时，并非每次都对上面三个内存区域一起回收的，大部分时候回收的都是指新生代

因此GC按照回收的区域又分了两种类型，一种是普通的GC（minor GC），一种是全局GC （major GC or Full GC)

普通GC：只针对新生代区域的GC

全局GC：针对老年代的GC，偶尔伴随对新生代的GC以及对永久代的GC

> GC面试题

1、JVM内存模型以及分区，需要详细到每个区放什么

2、堆里面的分区：Eden，Survival from to，老年代，各自的特点。

3、GC的三种收集方法：标记清除，标记整理，复制算法的原理与特点，分别用在什么地方？

4、Minor GC 与 Full GC 分别在什么时候发生?

很多的问题其实很简单，只是大家没有去研究而已，下面我们来聊聊几种垃圾回收方法！

GC四大算法

> 引用计数法　　说明：了解即可！

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的对象意味着是垃圾对象，可以被GC回收。

目前虚拟机基本都是采用可达性算法，从GC Roots 作为起点开始搜索，那么整个连通图中的对象边都是活对象，对于GC Roots 无法到达的对象变成了垃圾回收对象，随时可被GC回收。

> 复制算法（Copying）

年轻代中使用的是Minor GC，采用的就是复制算法（Copying）

什么是复制算法?

Minor GC 会把Eden中的所有活的对象都移到Survivor区域中，如果Survivor区中放不下，那么剩下的活的对象就被移动到Old generation中，也就是说，一旦收集后，Eden就是变成空的了

当对象在Eden（包括一个Survivor区域，这里假设是From区域）出生后，在经过一次Minor GC后，如果对象还存活，并且能够被另外一块Survivor区域所容纳 （上面已经假设为from区域，这里应为to区域，即to区域有足够的内存空间来存储Eden 和 From 区域中存活的对象），则使用复制算法将这些仍然还活着的对象复制到另外一块Survivor区域（即 to 区域）中，然后清理所使用过的Eden 以及Survivor区域（即form区域），并且将这些对象的年龄设置为1，以后对象在Survivor区，每熬过一次Minor

GC，就将这个对象的年龄 + 1，当这个对象的年龄达到某一个值的时候（默认是15岁，通过-XX:MaxTenuringThreshold 设定参数）这些对象就会成为老年代。

```
-XX:MaxTenuringThreshold     任期门槛=>设置对象在新生代中存活的次数
```

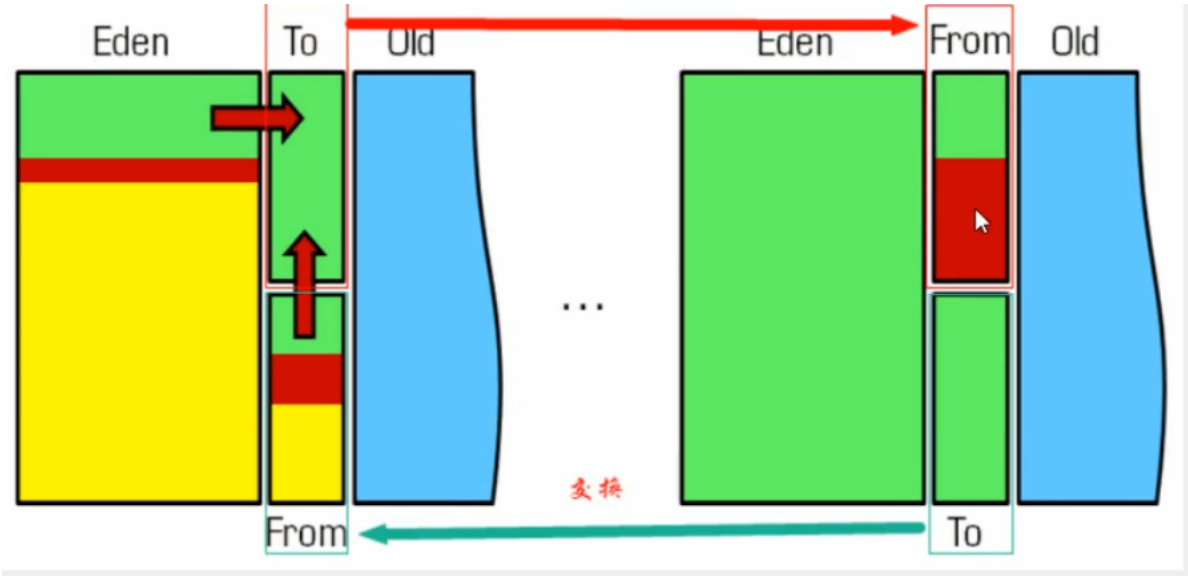
面试题：如何判断哪个是to区呢？一句话：**谁空谁是to**

原理解释：

年轻代中的GC，主要是复制算法（Copying）

HotSpot JVM 把年轻代分为了三部分：一个 Eden 区 和 2 个Survivor区（from区 和 to区）。默认比例为 8:1:1，一般情况下，新创建的对象都会被分配到Eden区（一些大对象特殊处理），这些对象经过第一次Minor GC后，如果仍然存活，将会被移到Survivor区，对象在Survivor中每熬过一次Minor GC，年龄就会增加1岁，当它的年龄增加到一定程度时，就会被移动到年老代中，因为年轻代中的对象基本上都是朝生夕死，所以在年轻代的垃圾回收算法使用的是复制算法！复制算法的思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片！

在GC开始的时候，对象只会在Eden区和名为 “From” 的Survivor区，Survivor区“TO” 是空的，紧接着进行GC，Eden区中所有存活的对象都会被复制到 “To”， 而在 “From” 区中，仍存活的对象会更具他们的年龄值来决定去向。年龄达到一定值 的对象会被移动到老年代中，没有达到阈值的对象会被复制到 “To 区域”，经过这次GC后，Eden区和From区已经被清空，这个时候， “From” 和 “To” 会交换他们的角色，也就是新的 “To“ 就是GC前的”From“， 新的 ”From“ 就是上次GC前的 ”To“。不管怎样，都会保证名为 To 的Survicor区域是空的。 Minor GC会一直重复这样的过程。直到 To 区 被填满， ”To “ 区被填满之后，会将所有的对象移动到老年代中。



因为Eden区对象一般存活率较低，一般的，使用两块10%的内存作为空闲和活动区域，而另外80%的内存，则是用来给新建对象分配内存的。一旦发生GC，将10%的from活动区间与另外80%中存活的Eden对象转移到10%的to空闲区域，接下来，将之前的90%的内存，全部释放，以此类推；

好处：没有内存碎片，坏处：浪费内存空间

劣势：

复制算法它的缺点也是相当明显的。

1、他浪费了一半的内存，这太要命了

2、如果对象的存活率很高，我们可以极端一点，假设是100%存活，那么我们需要将所有对象都复制一遍，并将所有引用地址重置一遍。复制这一工作所花费的时间，在对象存活率达到一定程度时，将会变的不可忽视，所以从以上描述不难看出。复制算法要想使用，最起码对象的存活率要非常低才行，而且最重要的是，我们必须要克服50%的内存浪费。

> 标记清除（Mark-Sweep）

说明：老年代一般是由标记清除或者是标记清除与标记整理的混合实现

什么是标记清除?

回收时，对需要存活的对象进行标记；

回收不是绿色的对象

当堆中的有效内存空间被耗尽的时候，就会停止整个程序（也被称为stop the world），然后进行两项工作，第一项则是标记，第二项则是清除。

标记：从引用根节点开始标记所有被引用的对象，标记的过程其实就是遍历所有的GC Roots，然后将所有GC Roots 可达的对象，标记为存活的对象。

清除： 遍历整个堆，把未标记的对象清除。

缺点：这个算法需要暂停整个应用，会产生内存碎片。

用通俗的话解释一下 标记/清除算法，就是当程序运行期间，若可以使用的内存被耗尽的时候，GC线程就会被触发并将程序暂停，随后将依旧存活的对象标记一遍，最终再将堆中所有没被标记的对象全部清除掉，接下来便让程序恢复运行。

劣势：

1. 首先、它的缺点就是效率比较低（递归与全堆对象遍历），而且在进行GC的时候，需要停止应用程序，这会导致用户体验非常差劲
2. 其次、主要的缺点则是这种方式清理出来的空闲内存是不连续的，这点不难理解，我们的死亡对象都是随机的出现在内存的各个角落，现在把他们清除之后，内存的布局自然乱七八糟，而为了应付这一点，JVM就不得不维持一个内存空间的空闲列表，这又是一种开销。而且在分配数组对象的时候，寻找连续的内存空间会不太好找。

> 标记压缩（Mark-Compact）

标记整理说明：老年代一般是由标记清除或者是标记清除与标记整理的混合实现。

什么是标记压缩?

在整理压缩阶段，不再对标记的对象作回收，而是通过所有存活对象都像一端移动，然后直接清除边界以外的内存。可以看到，标记的存活对象将会被整理，按照内存地址依次排列，而未被标记的内存会被清理掉，如此一来，当我们需要给新对象分配内存时，JVM只需要持有一个内存的起始地址即可，这比维护一个空闲列表显然少了许多开销。

标记、整理算法不仅可以弥补 标记、清除算法当中，内存区域分散的缺点，也消除了复制算法当中，内存减半的高额代价；

> 标记清除压缩（Mark-Sweep-Compact）

> 小总结

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：标记整理算法 = 标记清除算法 > 复制算法

可以看出，效率上来说，复制算法是当之无愧的老大，但是却浪费了太多内存，而为了尽量兼顾上面所提到的三个指标，标记整理算法相对来说更平滑一些， 但是效率上依然不尽如人意，它比复制算法多了一个标记的阶段，又比标记清除多了一个整理内存的过程。

难道就没有一种最优算法吗？猜猜看，下面还有

答案： 无，没有最好的算法，只有最合适的算法 。 ----------> 分代收集算法

年轻代： （Young Gen）

年轻代特点是区域相对老年代较小，对象存活低。

这种情况复制算法的回收整理，速度是最快的。复制算法的效率只和当前存活对象大小有关，因而很适用于年轻代的回收。而复制算法内存利用率不高的问题，通过hotspot中的两个survivor的设计得到缓解。

老年代： （Tenure Gen）

老年代的特点是区域较大，对象存活率高！

这种情况，存在大量存活率高的对象，复制算法明显变得不合适。一般是由标记清除或者是标记清除与标记整理的混合实现。Mark阶段的开销与存活对象的数量成正比，这点来说，对于老年代，标记清除或者标记整理有一些不符，但可以通过多核多线程利用，对并发，并行的形式提标记效率。Sweep阶段的开销与所管理里区域的大小相关，但Sweep “就地处决” 的 特点，回收的过程没有对象的移动。使其相对其他有对象移动步骤的回收算法，仍然是是效率最好的，但是需要解决内存碎片的问题。

常见面试题

1、JVM 垃圾回收的时候如何确定垃圾？是否知道什么是 GC Roots

什么是垃圾：简单的说就是内存中已经不再被使用到的空间就是垃圾。

```
1  Person person = null;
```

要进行垃圾回收，如何判断一个对象是否可以被回收？

> **方法一：引用计数法（了解即可）**

Java中，引用和对象是有关联的，如果要操作对象则必须用引用进行。

因此，很显然一个简单的办法是通过引用计数来判断一个对象是否可以进行回收。简单说，给对象中添加一个引用计数器，每当有一个地方引用它，计数器值加1，每当有一个引用失效时，计数器减1。

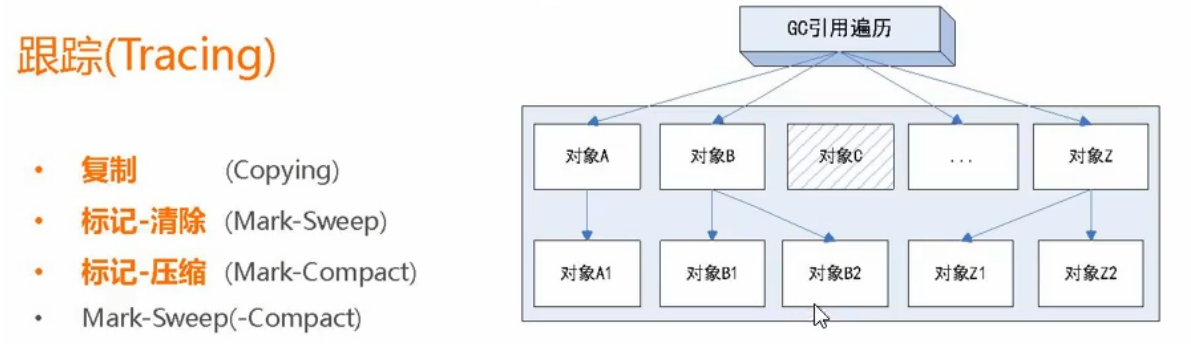
任何时刻计数器值为零的对象就是不可能再被使用的，那么这个对象就是可回收对象。

那么为什么主流的 Java虚拟机里面没有选用这种算法呢？其中最主要的原因是它很难解决对象之间相互循环引用的问题。

该算法存在但目前无人用了，解决不掉循环引用的问题，了解即可。

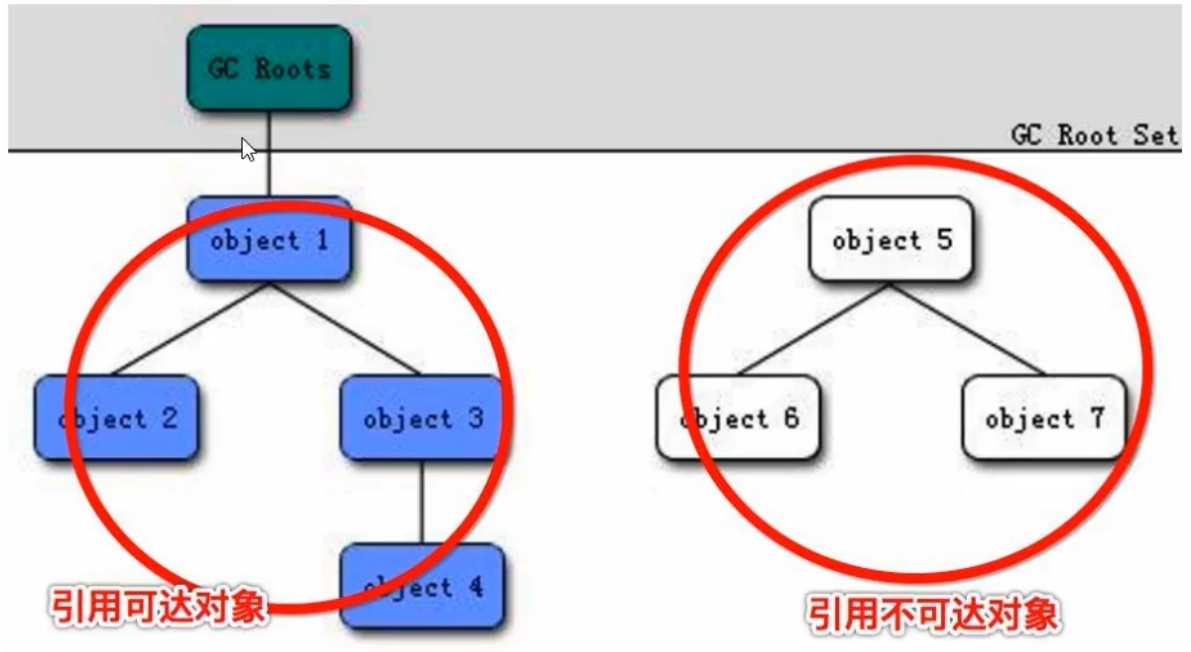
方法二：可达性分析算法

为了解决引用计数法的循环引用问题，Java 使用了可达性分析的方法。



所谓 `GC roots` 或者说 `tracing GC` 的“根集合" 就是一组必须活跃的引用。

基本思路就是通过一系列名为 `GC Roots` 的对象作为起始点，从这个被称为 GC Roots的对象开始向下搜索，如果一个对象到 GC Roots 没有任何引用链相连时，则说明此对象不可用。也即给定一个集合的引用作为根出发，通过引用关系遍历对象图，能被遍历到的（可到达的）对象就被判定为存活，没有被遍历到的就自然被判定为死亡。



Java中可以作为 GC Roots的对象：（共4种）

1、虚拟机栈（栈帧中的局部变量表）中引用的对象；

2、方法区中的类静态属性引用的对象。

3、方法区中常量引用的对象。

4、本地方法栈中 JNI （Native方法）引用的对象。

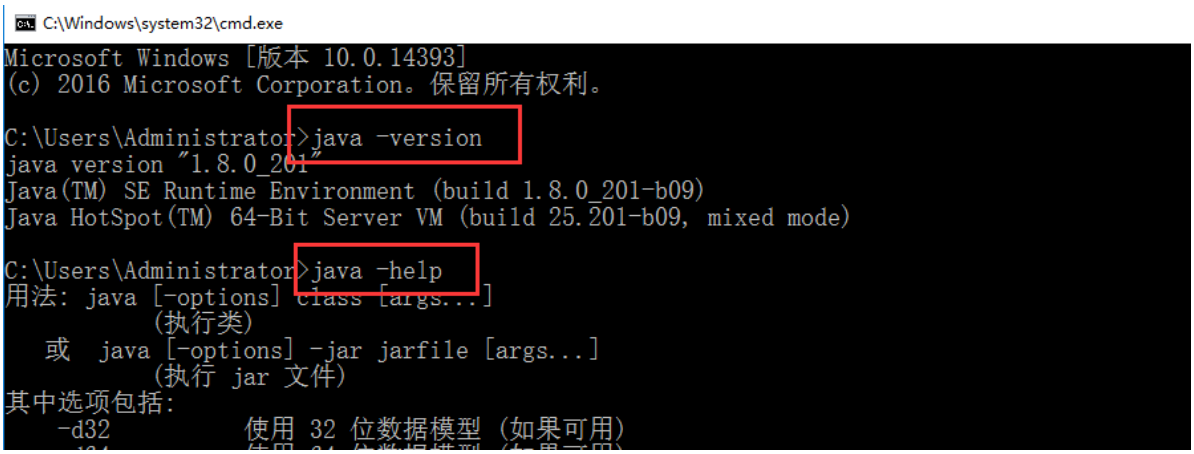
```
public class GCrootDemo{
    private byte[] byteArray = new byte[100*1024*1024];

    //private static GCrootDemo2 t2;
    //private static final GCrootDemo3 t3 = new GCrootDemo3();

    public static void m1(){
        GCrootDemo t1 = new GCrootDemo();
        System.gc();
        System.out.println("第一次GC完成");
    }

    public static void main(String[] args){
        m1();
    }

}
```

2、你说你做过 JVM 调优和参数配置，请问如何盘点查看 JVM 系统默认值

JVM的参数类型有三种：标配参数、X参数、XX参数；

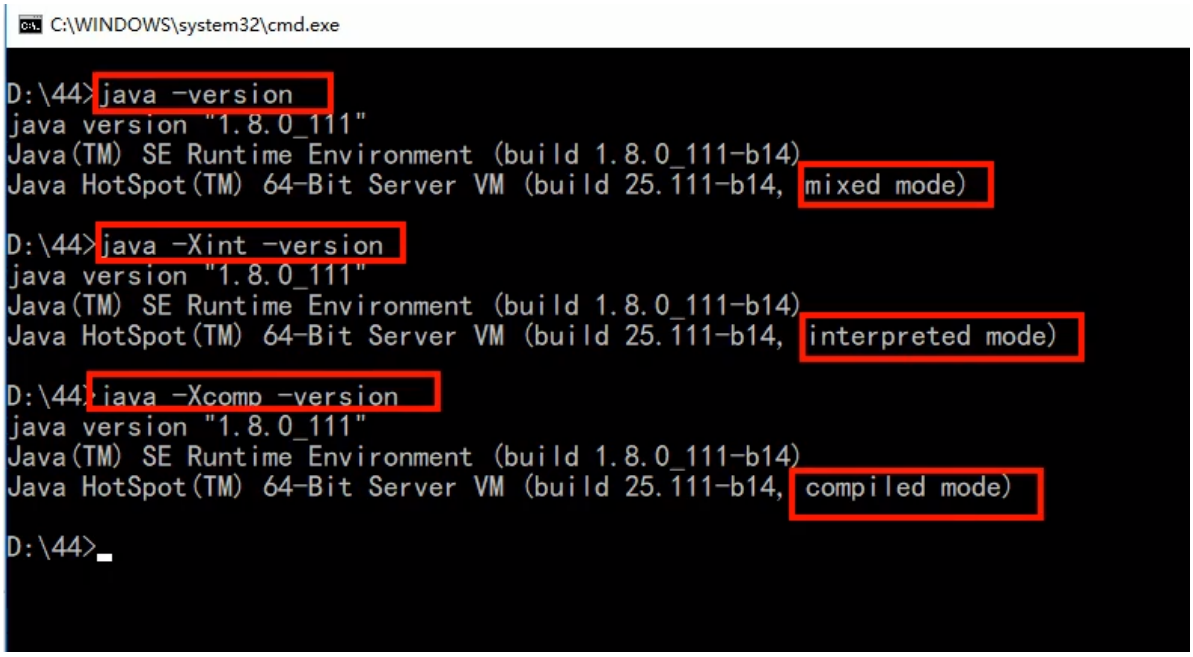
> **标配参数：**在 JDK 各个版本之间很稳定，很少有大的变化；

```
-version
-help
-showversion
```



> **X参数（了解）**

```
-Xint      # 解释执行
-Xcomp     # 第一次使用就编译成本地代码
-Xmixed    # 混合模式
```



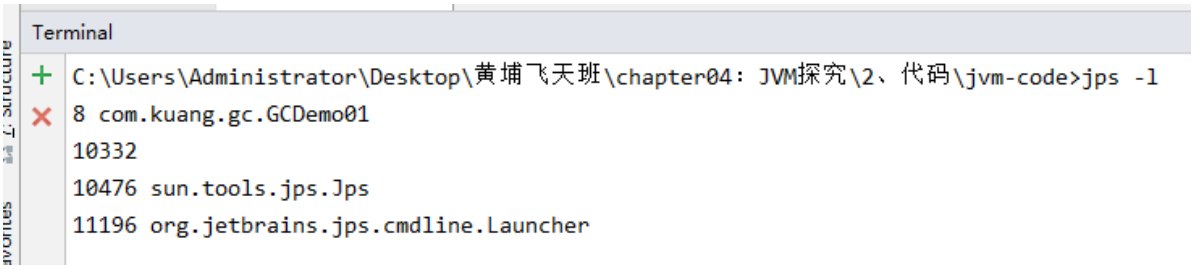
> **XX参数之Boolean类型** 公式：`-XX: + 或者 - 某个属性值` + 表示开启，- 表示关闭。

我们来启动代码测试：

```
package com.kuang.gc;

public class GCDemo01 {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("Hello");
        Thread.sleep(Integer.MAX_VALUE);
    }
}
```

如何查看一个正则运行中的 Java程序，它的某个 JVM 参数是否开启？具体值是多少？

`jps -l` 得到当前正在运行的进程编号



`jinfo -flag 进程号` 查看正则运行中的 Java程序，它的某个 JVM 参数是否开启

```
C:\Users\Administrator\Desktop\黄埔飞天班\chapter04: JVM探究\2、代码\jvm-code>jinfo -flag PrintGCDetails 8
-XX:-PrintGCDetails
```

我们停止程序，然后增加 VM 参数：

```
-XX:+PrintGCDetails
```

启动后再次测试，看是否开启！

```
C:\Users\Administrator\Desktop\黄埔飞天班\chapter04：JVM探究\2、代码\jvm-code>jinfo -flag PrintGCDetails 7476
-XX:+PrintGCDetails
```

发现已开启

小结：

1、是否打印GC收集细节：PrintGCDetails

2、是否使用串行垃圾回收器：UseSerialGC

XX参数之KV设值类型 公式：`-XX：属性key=属性值value`

1、`-XX:MetaspaceSize=128m` 元空间大小

2、`-XX:MaxTenuringThreshold=15` 进老年区存活次数判定

```
D:\devSoft\JetBrains\IdeaProjects\demo01>jinfo -flag MetaspaceSize 4300
-XX:MetaspaceSize=21807104
```

默认的

```
D:\devSoft\JetBrains\IdeaProjects\demo01>jinfo -flag MaxTenuringThreshold 5988
-XX:MaxTenuringThreshold=15
```

默认的

```
D:\devSoft\JetBrains\IdeaProjects\demo01>jinfo -flags 5988
Attaching to process ID 5988, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.111-b14
Non-default VM flags: -XX:CICompilerCount=4 -XX:InitialHeapSize=266338304 -XX:MaxHeapSize=4253024256 -XX:MaxNewSize=1417674752 -
XX:MetaspaceSize=1262485504 -XX:MinHeapDeltaBytes=524288 -XX:NewSize=88604672 -XX:OldSize=177733632 -XX:+UseCompressedClassPoint
ers -XX:+UseCompressedOops -XX:+UseFastUnorderedTimeStamps -XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC
Command line:  -XX:MetaspaceSize=1204m -javaagent:D:\devSoft\JetBrains\IntelliJ IDEA 2018.1.6\lib\idea_rt.jar=61077:D:\devSoft\J
etBrains\IntelliJ IDEA 2018.1.6\bin -Dfile.encoding=UTF-8
```

查看所有信息

默认

命令行修改

美团面试题：两个经典参数：`-Xms 和 -Xmx`，请问这个怎么解释呢？考察你有没有去研究过！

1、-Xms 等价于 -XX:InitialHeapSize 初始堆大小

2、-Xmx 等价于 -XX:MaxHeapSize 最大对大小

查看初始默认值

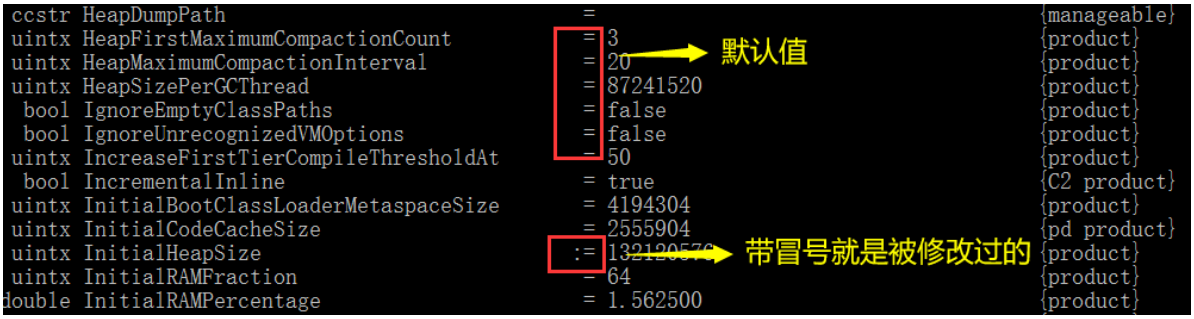
`-XX:+PrintFlagsInitial` 查看Java环境初始默认值

```
1  java -XX:+PrintFlagsInitial
```

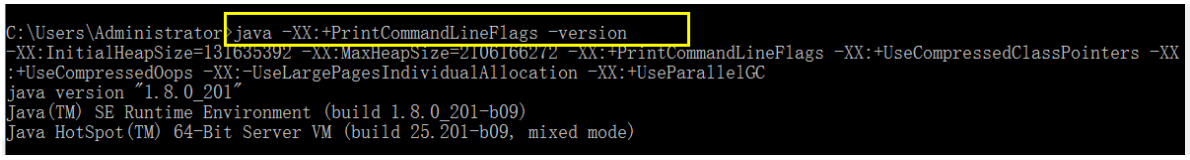
```
C:\Windows\system32\cmd.exe
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>java -XX:+PrintFlagsInitial
[Global flags]
     intx ActiveProcessorCount                      = -1                                  {product}
    uintx AdaptiveSizeDecrementScaleFactor          = 4                                   {product}
    uintx AdaptiveSizeMajorGCDecayTimeScale         = 10                                  {product}
    uintx AdaptiveSizePausePolicy                   = 0                                   {product}
    uintx AdaptiveSizePolicyCollectionCostMargin    = 50                                  {product}
    uintx AdaptiveSizePolicyInitializingSteps       = 20                                  {product}
    uintx AdaptiveSizePolicyOutputInterval          = 0                                   {product}
    uintx AdaptiveSizePolicyWeight                  = 10                                  {product}
    uintx AdaptiveSizeThroughPutPolicy              = 0                                   {product}
```

`-XX:+PrintFlagsFinal` 查看修改更新

```
java -XX:+PrintFlagsFinal -version
// 具体执行 后面是要修改的参数， Test要运行的 Java 类
java -XX:+PrintFlagsFinal -Xss128K Test
```



`java -XX:+PrintCommandLineFlags -version` 程序运行前打印出用户手动设置或者JVM自动设置的XX选项

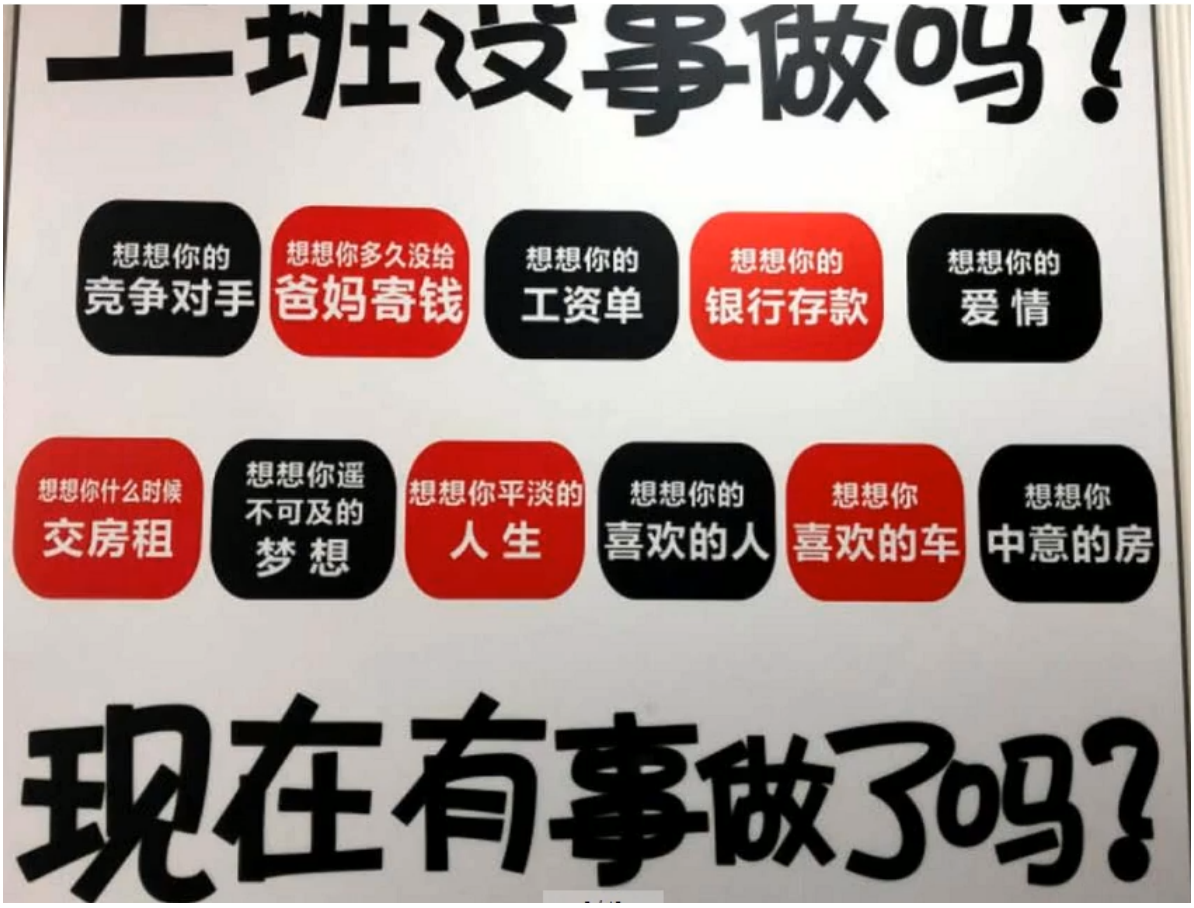


聊天鸡汤：

1、要想人前显贵，必定人后遭罪

2、今天的最好表现是对你们明天的最低要求

3、大家出去的时候以后一定要坐飞机，有的人还没坐过飞机，你以前出去，都是坐客车，坐火车出去的，无论如何今年对自己好点，坐一次飞机，先投资自己的脑袋，在丰富自己的口袋。你去看看汽车站，火车站，飞机站的旅客的区别，你看看你想过哪一种的生活，客车拥挤，你听过飞机卖站票的吗？你现在的努力，你现在的辛苦就是为了以后更好的生活！



3、你平时工作用过的 JVM 常用基本配置参数有哪些?

> -Xms

初始内存大小，默认为物理内存的 1/64，等价于 `-XX:InitialHeapSize`

> -Xmx

最大内存大小，默认为物理内存的1/4，等价于 `-XX:MaxHeapSize`

> -Xss

设置单个线程栈的大小，一般默认为 512k ~ 1024k，等价于 `-XX:TheadStackSize`

> -Xmn

设置年轻代大小，一般不用动

> -XX:MetaspaceSize

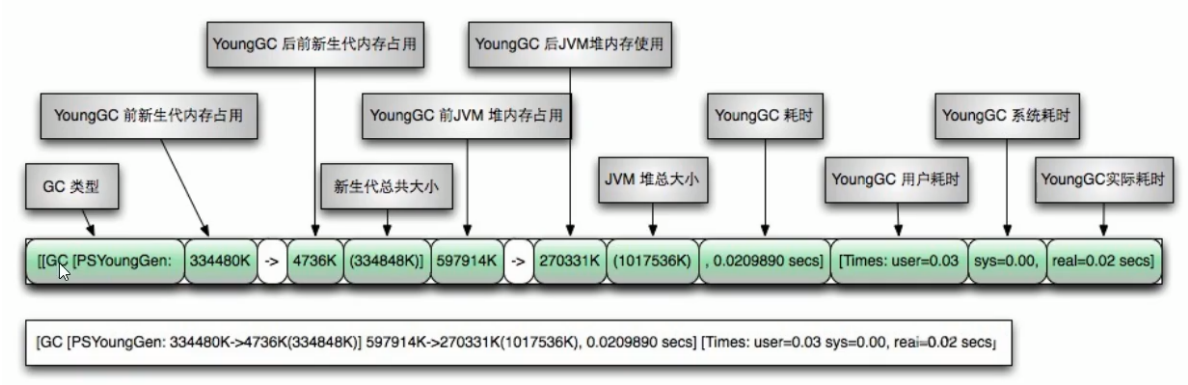
设置元空间大小：

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。

使用：`-XX:MetaspaceSize=512m`

> -XX:+PrintGCDetails

输出详细GC收集日志信息；输出参数说明如下：



> -XX:SurvivorRatio

设置新生代中 eden 和 s0/s1 空间的比例；

默认 `-XX:SurvivorRatio=8，Eden:S0:S1=8:1:1`

假如 `-XX:SurvivorRatio=4，Eden:S0:S1=4:1:1`

SurvivorRatio 值就是设置 eden区的比例占多少，s0/s1 相同

> -XX:NewRatio

配置年轻代与老年代在堆结构的占比；

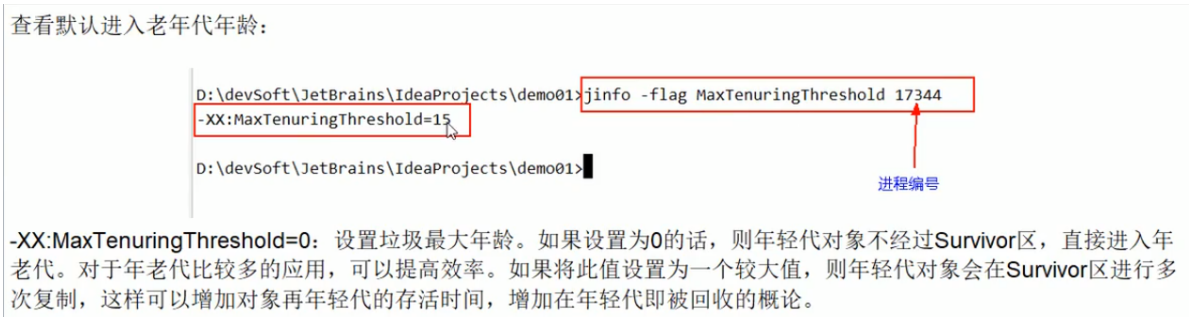
默认 `-XX:NewRatio=2` 新生代占1，老年代2，年轻代占整个堆的 1/3

假如 `-XX:NewRatio=4` 新生代占1，老年代4，年轻代占整个堆的 1/5

NewRatio 值就是这只老年代的占比，剩下的1给新生代

> -XX:MaxTenuringThreshold

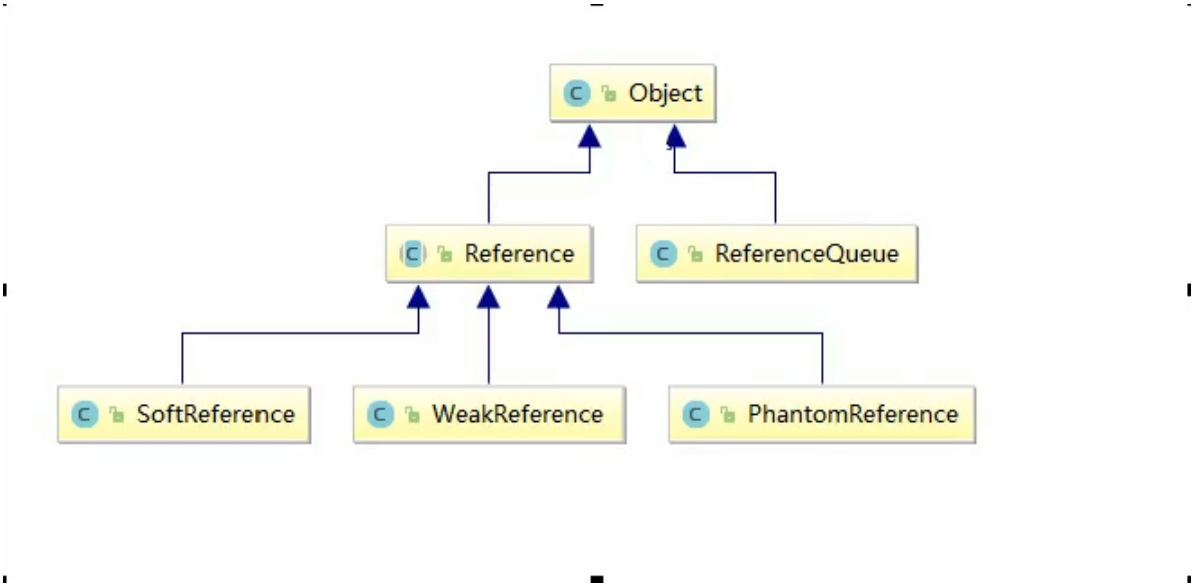
设置进入老年区的年龄限制，必须在 0~15 之间，默认 15；



-XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概论。

4、强引用、软引用、弱引用、虚引用分别是什么?

> 整体架构



强引用（默认支持模式）

当内存不足，JVM开始垃圾回收，对于强引用的对象，就是出现了OOM也不会对该对象进行回收，死都不收。

强引用是我们最常见的普通引用，只要还有强引用指向一个对象，就能表明对象还活着，垃圾收集器不会碰这种对象。在 Java中最常见的就是强引用，把一个对象赋值给一个引用变量，这个引用变量就是一个强引用，当一个对象被强引用时，它处在可达状态，它是不可能被垃圾回收机制回收的，即使该对象以后永远都不会被用到，JVM也不会回收。因此强引用是造成 Java 内存泄漏的主要原因之一。

对于一个普通的对象，如果没有其他的引用关系，只要超过了引用的作用域或者显示的将相应（强）引用赋值为null，一般认为就是可以被垃圾收集的了（当然具体回收时机还是要看垃圾收集策略）。

```
Object obj1 = new Object(); // 这样定义的默认就是强引用
Object obj2 = obj1; // obj2引用赋值
obj1 = null; // 置空
System.gc();
System.out.println(obj2); // 正常输出
```

软引用（SoftReference）

软引用是一种相对强引用弱化了一些的引用，需要调用 java.lang.ref.SoftReference类来实现，可以让对象豁免一些垃圾收集。

对于只有软引用的对象来说：

当系统内存充足时它 不会 被回收，当系统内存不足时它 会 被回收。

软引用通常在对应内存敏感的程序中，比如高速缓存就有用到软引用，内存够用的时候就保留，不够用就回收！

```
package com.kuang.gc;

import java.lang.ref.SoftReference;

public class SoftReferenceDemo {

```

```
    public static void softRef_Memory_Enough(){
        Object o1 = new Object();
        SoftReference<Object> softReference = new SoftReference<>(o1);
        System.out.println(o1);
        System.out.println(softReference.get());

        o1 = null;
        System.gc();

        System.out.println(o1);
        System.out.println(softReference.get());
    }

    /*
     JVM 配置，让它内存不够
     -Xms5m -Xmx5m -XX:+PrintGCDetails
     */
    public static void softRef_Memory_NotEnough(){
        Object o1 = new Object();
        SoftReference<Object> softReference = new SoftReference<>(o1);
        System.out.println(o1);
        System.out.println(softReference.get());

        o1 = null;
        //System.gc();

        try {
            byte[] bytes = new byte[30*1024*1024];
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            System.out.println(o1);
            System.out.println(softReference.get());
        }
    }

    public static void main(String[] args) {
        softRef_Memory_NotEnough();
    }
}
```

> 弱引用 WeakReference

弱引用需要用 java.lang.ref.WeakReference 类来实现，它比软引用的生存周期更短，对于只有弱引用的对象来说，只要垃圾回收机制一运行，不管 JVM 的内容空间是否足够，都会回收该对象占用的内存；

```
public static void main(String[] args) {
    Object o1 = new Object();
    WeakReference<Object> weakReference = new WeakReference<>(o1);
    System.out.println(o1);
    System.out.println(weakReference.get());

    o1 = null;
    System.gc();

    System.out.println(o1);
    System.out.println(weakReference.get());
}
```

> 软引用和弱引用的适用场景

假如有一个应用需要读取大量的本地图片：

1、如果每次读取图片都要从硬盘读取则会严重影响性能；

2、如果一次性全部加载到内存中又可能造成内存溢出。

此时适用软引用可以解决这类问题：

设计思路是：用一个HashMap来保存图片的路径和相应图片对象关联的软引用之间的映射关系，在内存不足时，JVM 会自动回收这些缓存图片对象所占用的空间，从而有效地避免了 OOM 的问题。

```
Map<String,SoftReference<Bitmap>> imageCache
    = new HashMap<String,SoftReference<Bitmap>>();
```

> 虚引用 PhantomReference

虚引用需要使用 java.lang.PhantomReference 类来实现。

顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。

如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收，它不能单独使用也不能通过它访问对象，虚引用必须和引用队列（ReferenceQueue）联合使用。

虚引用的主要作用是跟踪对象被垃圾回收的状态。仅仅是提供了一种确保对象被 finalize 以后，做某些事情的机制。PhantomReference 的 get方法总是返回 null，因此无法访问对应的引用对象。其意义在于说明一个对象已经进入 finalization 阶段，可以被 gc 回收，用来实现比 finalization 机制更灵活的回收操作。

换句话说，设置虚引用关联的唯一目的，就是在这个对象被收集器回收的时候收到一个系统通知或者后续添加进一步的处理。

Java技术允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。

```
public static void main(String[] args) throws InterruptedException {
    Object o1 = new Object();
    ReferenceQueue<Object> referenceQueue = new ReferenceQueue<>();
    PhantomReference<Object> phantomReference = new PhantomReference<>
(o1,referenceQueue);

    System.out.println(o1);  // java.lang.Object@1b6d3586
```

```
7       System.out.println(phantomReference.get()); // null
8       System.out.println(referenceQueue.poll());  // null
9
10      System.out.println("==================");
11      o1 = null;
12      System.gc();
13      TimeUnit.SECONDS.sleep(1);
14
15      System.out.println(o1);  // null
16      System.out.println(phantomReference.get());  // null
17      System.out.println(referenceQueue.poll());  // java.lang.Object@1b6d3586
18
19  }
```

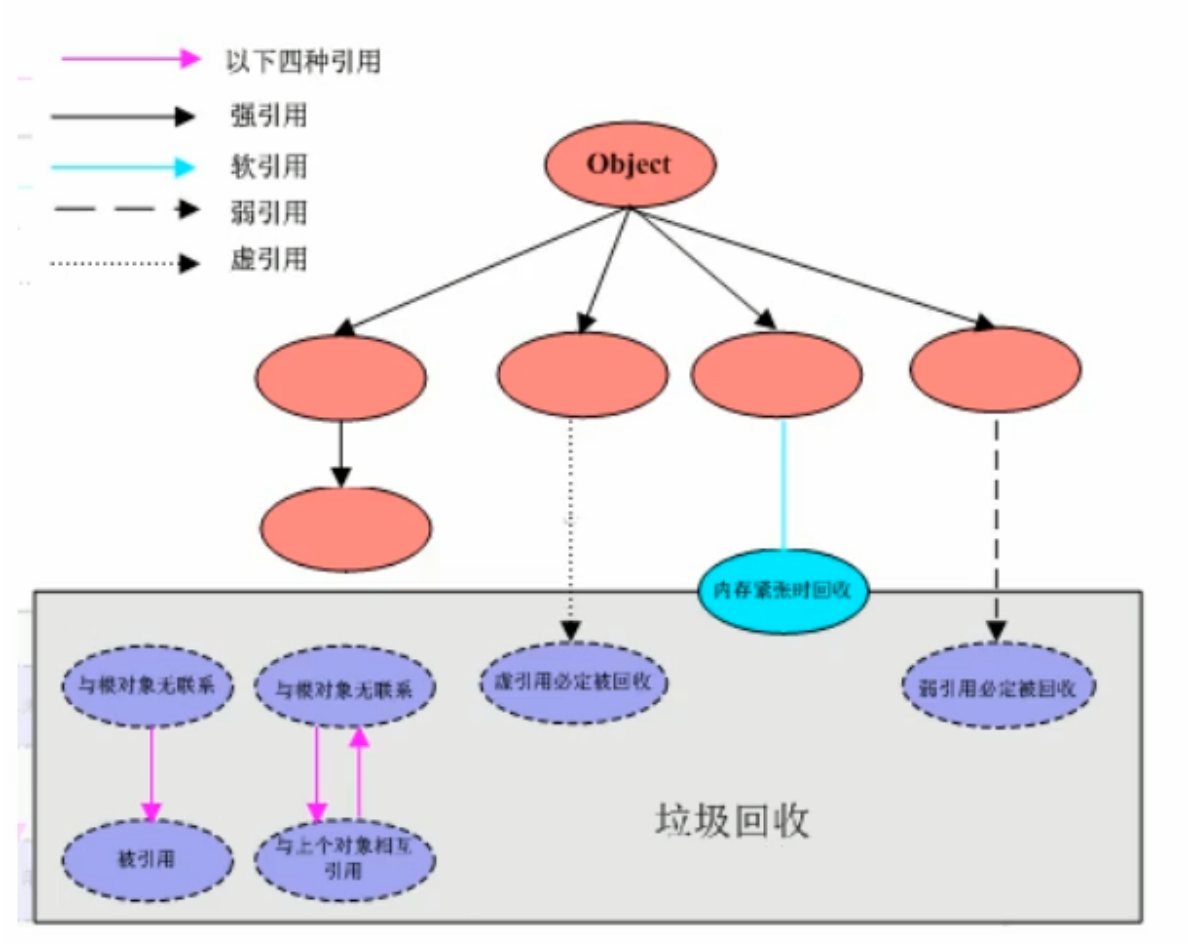
Java提供了4中引用类型，在垃圾回收的时候，都有自己各自的特点。

ReferenceQueue 是用来配合引用工作的，没有 ReferenceQueue一样可以运行。

创建引用的时候可以指定关联的队列，当GC 释放对象内存的时候，会将引用加入到引用队列，如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。这相当于是一种通知机制。

当关联的引用队列中有数据的时候，意味着引用指向的堆内存中的对象被回收。通过这种方式，JVM 允许我们在对象被销毁后，做一些我们自己想做的事情。

> 一幅图小总结



5、请你谈谈你对 OOM 的认识?

> java.lang.StackOverflowError

```
public static void main(String[] args){
    a();
}
public static void a(){
    a();
}
```

> java.lang.OutOfMemoryError：Java heap space

```
// -Xms10m -Xmx10m
public static void main(String[] args){
    String str = "kuangshen";
    while(true){
        str += str + new Random().nextInt(11111111) + new
Random().nextInt(11111111);
    }
}
```

> java.lang.OutOfMemoryError：GC overhead limit exceeded 超过GC开销限制

GC回收时间过长时会抛出 OutOfMemoryError。过长的定义是，超过98%的时间用来做GC并且回收了不到 2% 的堆内存，连续多次 GC 都只回收了不到 2% 的极端情况下才会抛出，假如不抛出 GC overhead limit exceeded 错误会发生什么情况呢？那就是 GC 清理的这么点内存很快会再次填满，迫使GC再次执行，这样就形成恶性循环，CPU使用率一直是100%，而GC却没有任何成果。

```
// -Xms10m -Xmx10m -XX:MaxDirectMemorySize=5m -XX:+PrintGCDetails
public static void main(String[] args) throws InterruptedException {
    int i = 0;
    List<String> list = new ArrayList<>();

    try {
        while (true){
            list.add(String.valueOf(++i).intern());
        }
    } catch (Throwable e) {
        System.out.println("i=>"+i);
        e.printStackTrace();
        throw e;
    }
    // java.lang.OutOfMemoryError: GC overhead limit exceeded
}
```

```
[Full GC (Ergonomics) [PSYoungGen: 2047K->2047K(2560K)] [ParOldGen: 7071K->7071K(7168K)] 9119K->9119K(9728K), [Metaspace: 3223K->3223
[Full GC (Ergonomics) [PSYoungGen: 2047K->2047K(2560K)] [ParOldGen: 7073K->7073K(7168K)] 9121K->9121K(9728K), [Metaspace: 3223K->3223
i=>145800
[Full GC (Ergonomics) [PSYoungGen: 2047K->0K(2560K)] [ParOldGen: 7089K->605K(7168K)] 9137K->605K(9728K), [Metaspace: 3247K->3247K(105
Heap
 PSYoungGen      total 2560K, used 56K [0x00000000ffd00000, 0x0000000100000000, 0x0000000100000000)
  eden space 2048K, 2% used [0x00000000ffd00000,0x00000000ffd0e090,0x00000000fff00000)
  from space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000)
```

> java.lang.OutOfMemoryError：Direct buffer memory 直接缓冲存储

写NIO程序经常使用 ByteBuffer 来读取或者写入数据，这是一种基于通道（channel）与缓冲区（Buffer）的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在 Java 堆里面的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆中来回复制数据。

`ByteBuffer.allocate(capability)` 第一种方式是分配JVM堆内存，属于GC管辖范围，由于需要拷贝所以速度相对较慢。

`ByteBuffer.allocateDirect(capability)` 第二种方式是分配 OS本地内存，不属于GC管辖范围，由于不需要拷贝所以速度相对较快。

但如果不断分配本地内存，堆内存很少使用，那么JVM就不需要执行 GC，DirectByteBuffer 对象们就不会被回收，这时候堆内存充足，但本地内存可能已经使用光了，再次尝试分配本地内存就会出现 OutOfMemoryError，那程序就直接崩溃了。

```
// -Xms10m -Xmx10m -XX:MaxDirectMemorySize=5m -XX:+PrintGCDetails
public static void main(String[] args) throws InterruptedException {
    System
        .out
        .println("配置的maxDirectMemory：" +
(sun.misc.VM.maxDirectMemory()/(double)1014/1024)+"MB");

    TimeUnit.SECONDS.sleep(2);
    // -Xms10m -Xmx10m -XX:MaxDirectMemorySize=5m -XX:+PrintGCDetails
    // 我们配置的5M，但是实际使用的6M，故意搞破坏
    ByteBuffer byteBuffer = ByteBuffer.allocateDirect(6 * 1024 * 1024);
    // java.lang.OutOfMemoryError: Direct buffer memory
}
```

> java.lang.OutOfMemoryError：unable to create new native thread 无法创建新的本地线程

高并发请求服务器时，经常出现如下异常：java.lang.OutOfMemoryError：unable to create new native thread

准确的讲，该native thread 异常与对应的平台有关；

导致原因：

1、你的应用创建了太多线程了，一个应用进程创建多个线程，超过系统承载极限

2、你的服务器并不允许你的应用程序创建这么多线程，Linux 系统默认允许单个进程可以创建的线程数是1024个，你的应用创建超过这个数量，就会报 java.lang.OutOfMemoryError：unable to create new native thread

解决办法：

1、想办法降低你的应用程序创建线程的数量，分析应用是否真的需要创建这么多线程，如果不是，改代码将线程数降到最低！

2、对于有的应用，确实需要创建很多线程，远超过Linux系统的默认 1024 个线程的限制，可以通过修改 Linux 服务器配置，扩大Linux默认限制！

```
//在 Linux 虚拟机下操作，使用非 root 用户测试，因为root用户是无上限创建线程的。
public static void main(String[] args) {
    for (int i = 1;  ; i++) {
        System.out.println("i=>"+i);
        new Thread(()->{
            try {
                Thread.sleep(Integer.MAX_VALUE);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        },""+i).start();
    }
}
```

> java.lang.OutOfMemoryError：Metaspace

Java 8 及之后的版本使用 Metaspcae 来替代永久代。

Metaspace 是方法区在 HotSpot中的实现，它与持久代最大的区别在于：Metaspace并不在虚拟机内存中而是使用本地内存。

永久代（java8后被原空间Metaspace取代了）存放了以下信息：

1、虚拟机加载的类信息

2、常量池

3、静态变量

4、即时编译后的代码

模拟Metaspace空间溢出，我们不断生成类往元空间灌，类占据的空间总是会超过Metaspace指定的空间大小的。

```
import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

// 注意要导入 spring 的核心包!
// -XX:MetaspaceSize=10m -XX:MaxMetaspaceSize=10m
public class Test{

    static class OOMTest{ }
    public static void main(final String[] args) {
        int i = 0; //模拟计数器，来查看多少次以后发生异常
        try{
            while(true){
                i++;
                // Spring的cglib动态代理技术
                Enhancer enhancer = new Enhancer();
                enhancer.setSuperclass(OOMTest.class);
                enhancer.setUseCache(false);
                enhancer.setCallback(new MethodInterceptor() {
                    public Object intercept(Object o, Method method,
Object[] objects, MethodProxy methodProxy) throws Throwable {
```

```
23                          return method.invoke(o,args);
24                      }
25                  });
26                  enhancer.create();
27              }
28          }catch(Throwable e){
29              System.out.println("i=>"+i);
30              e.printStackTrace();
31          }
32      }
33  }
```

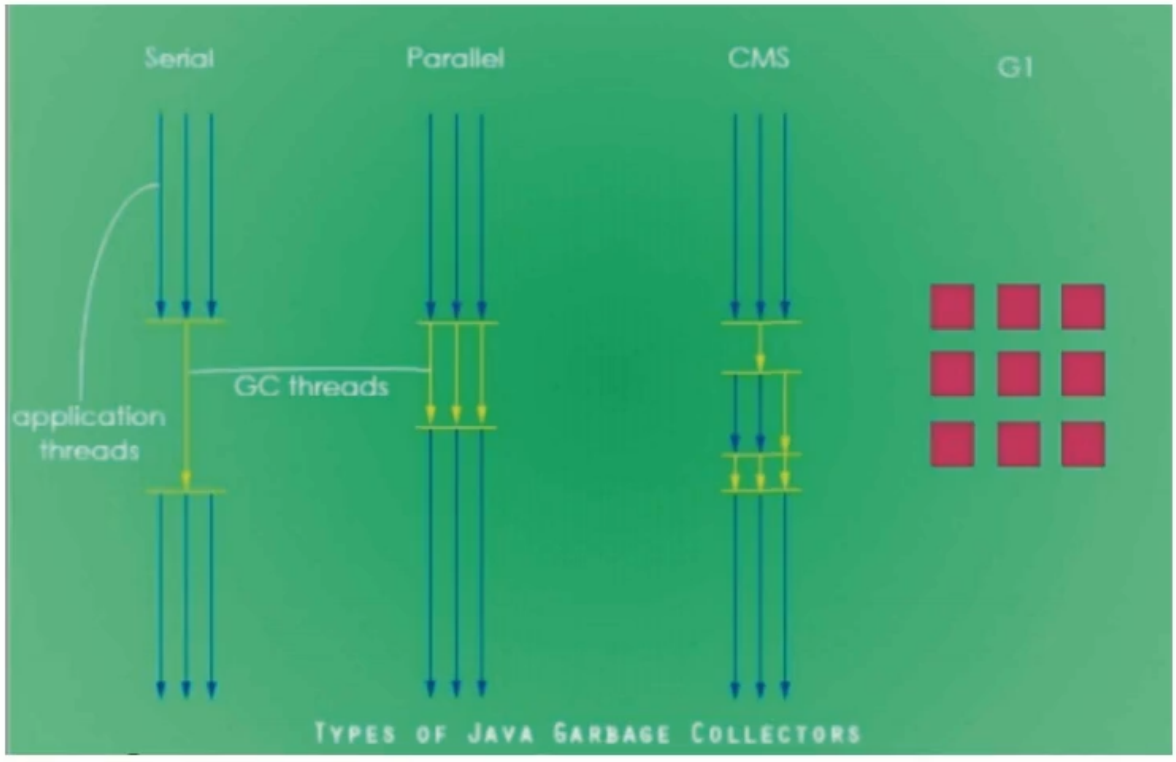
```
D:\Environment\java\jdk1.8.0_201\bin\java.exe ...
i=>568
org.springframework.cglib.core.CodeGenerationException: java.lang.OutOfMemoryError-->Metaspace
    at org.springframework.cglib.core.ReflectUtils.defineClass(ReflectUtils.java:530)
    at org.springframework.cglib.core.AbstractClassGenerator.generate(AbstractClassGenerator.java:363)
    at org.springframework.cglib.proxy.Enhancer.generate(Enhancer.java:582)
    at org.springframework.cglib.core.AbstractClassGenerator$ClassLoaderData.get(AbstractClassGenerator.java:131)
    at org.springframework.cglib.core.AbstractClassGenerator.create(AbstractClassGenerator.java:319)
    at org.springframework.cglib.proxy.Enhancer.createHelper(Enhancer.java:569)
    at org.springframework.cglib.proxy.Enhancer.create(Enhancer.java:384)
    at Test.main(Test.java:24)
Caused by: java.lang.OutOfMemoryError: Metaspace
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:763) <3 internal calls>
    at org.springframework.cglib.core.ReflectUtils.defineClass(ReflectUtils.java:527)
    ... 7 more
```

6、GC垃圾回收算法和垃圾收集器的关系？分别是什么？

GC算法（引用计数、复制、标记清除、标记整理）是内存回收的方法论，垃圾收集器就是算法的落地实现；

因为目前为止还没有完美的收集器出现，更加没有万能的收集器，只是针对具体应用最合适的收集器，进行分代收集；

> 4 种主要垃圾收集器



`串行垃圾回收器(Serial)`

它为单线程环境设计且只使用一个线程进行垃圾回收，会暂停所有的用户线程。所以不适合服务器环境。

`并行垃圾回收器(Parallel)`

多个垃圾收集线程并行工作，此时用户线程是暂停的，适用于科学计算、大数据处理首台处理等弱交互场景。

`并发垃圾回收器(CMS)`

用户线程和垃圾收集线程同时执行（不一定是并行，可能交替执行），不需要停顿用户线程，互联网公司多用它，适用对响应时间有要求的场景。

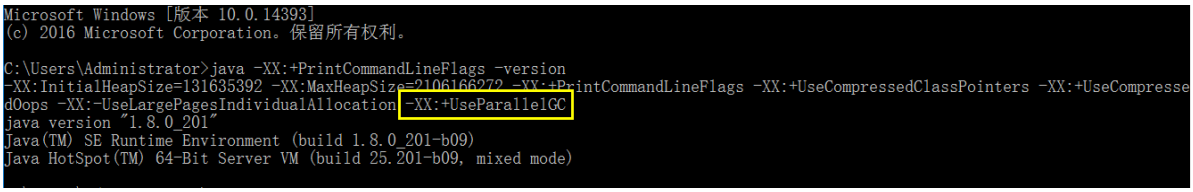
`G1垃圾回收器`

G1垃圾回收器将堆内存分割成不同的区域然后并发的对其进行垃圾回收

7、谈谈垃圾收集器

> 怎么查看默认的垃圾收集器是哪个?

命令行输入：`java -XX:+PrintCommandLineFlags -version`



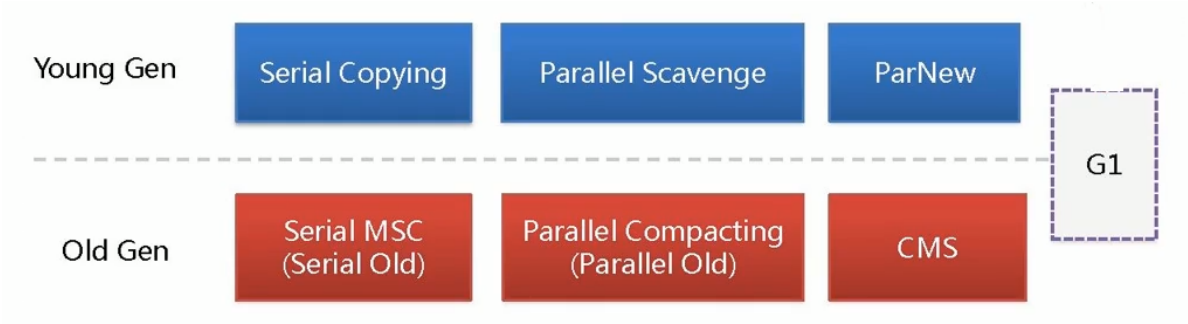
> 默认的垃圾收集器有哪些?

java 的 gc 回收的类型主要有几种：

`UseSerialGC`、`UseParallelGC`、`UseConcMarkSweepGC`、`UseParNewGC`、`UseParallelOldGC`、`UseG1GC`

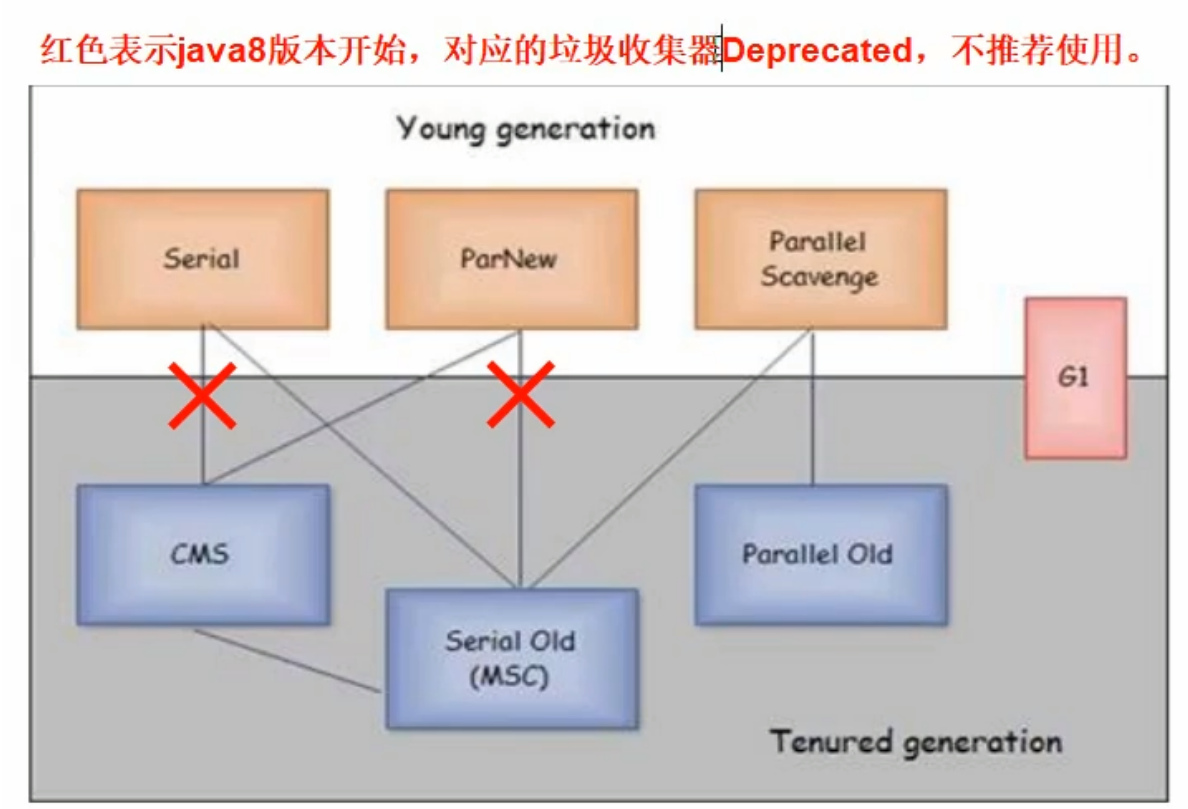
```
bool Arguments::check_gc_consistency() {
  bool status = true;
  // Ensure that the user has not selected conflicting sets
  // of collectors. [Note: this check is merely a user convenience;
  // collectors over-ride each other so that only a non-conflicting
  // set is selected; however what the user gets is not what they
  // may have expected from the combination they asked for. It's
  // better to reduce user confusion by not allowing them to
  // select conflicting combinations.
  uint i = 0;
  if (UseSerialGC)                       i++;
  if (UseConcMarkSweepGC || UseParNewGC) i++;
  if (UseParallelGC || UseParallelOldGC) i++;
  if (UseG1GC)                           i++;
  if (i > 1) {
    jio_fprintf(defaultStream::error_stream(),
                "Conflicting collector combinations in option list; "
                "please refer to the release notes for the combinations "
                "allowed\n");
    status = false;
  }

  return status;
}
```

垃圾收集器



垃圾收集器就来具体实现这些GC算法并实现内存回收。

不同厂商、不同版本的虚拟机实现差别很大，HotSpot 中包含的收集器如下图所示：



> 部分参数说明

DefNew => Default New Generation 【默认新一代】

Tenured => Old 【老年代】

ParNew => Parallel New Generation 【并行新一代】

PSYoungGen => Parallel Scavenge 【并行清除年轻区】

ParOldGen => Parallel Old Generation 【并行老年区】

> Server / Client 模式分别是什么意思?

适用范围：只需要掌握 Server 模式即可，Client模式基本不会用

操作系统：

32位Window操作系统，不论硬件如何都默认使用 Client 的 JVM 模式。

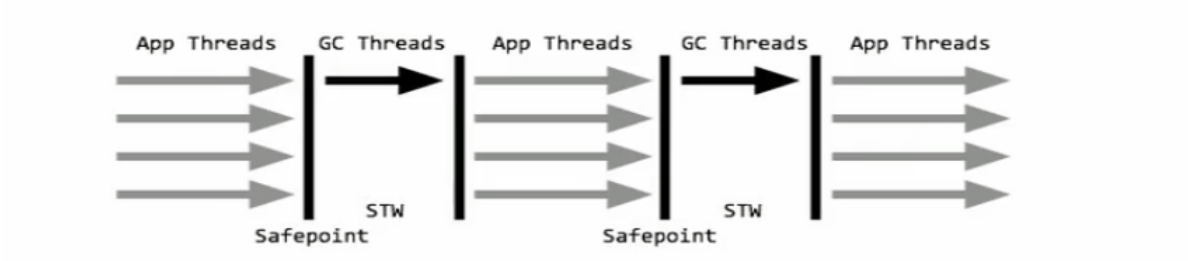
32位其他操作系统，2G 内存同时有 2个 cpu以上用Server模式，低于该配置还是 Client 模式。

64位都是 Server 模式。

> 新生代

串行GC（Serial收集器）/ （Serial Copying）

一句话：一个单线程的收集器，在进行垃圾收集时候，必须暂停其他所有的工作线程直到它收集结束。



串行收集器是最古老，最稳定以及效率高的收集器，只使用一个线程去回收但其在进行垃圾收集过程中可能会产生较长的停顿，Stop-The-World，虽然在手机垃圾过程中需要暂停所有其他的工作线程，但是它简单高效，对于限定单个 CPU 环境来说，没有线程交互的开销可以获得最高的单线程垃圾收集效率，因此Serial垃圾收集器依然是java虚拟机运行在 Client 模式下默认的新生代垃圾收集器。

对应 JVM 参数是：`-XX:+UseSerialGC`

开启后会使用：Serial（Young区用）+ Tenured（Old区用）的收集器组合

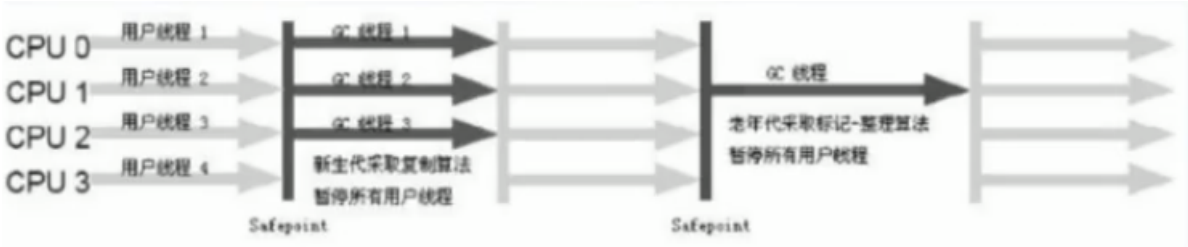
表示：新生代、老年代都会使用串行回收收集器，新生代使用复制算法，老年代使用标记-整理算法；

```
-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -
XX:+UseSerialGC
```

```
[GC (Allocation Failure) [DefNew: 2181K->2181K(3072K), 0.0000102 secs][Tenured: 5404K->2798K(6848K), 0.0024888 secs]
[GC (Allocation Failure) [DefNew: 2157K->0K(3072K), 0.0005035 secs][Tenured: 4902K->3850K(6848K), 0.0017765 secs] 49
[Full GC (Allocation Failure) [Tenured: 3850K->3760K(6848K), 0.0027737 secs] 3850K->3760K(9920K), [Metaspace: 3231K-
```

并行GC（ParNew）

一句话：使用多线程进行垃圾回收，在垃圾收集时，会 Stop-The-World 暂停其他所有的工作线程直到它收集结束。



ParNew 收集器其实就是 Serial 收集器新生代的并行多线程版本，最常见的应用场景是配合老年代的 CMG GC 工作，其余的行为和Serial 收集器完全一样，ParNew 垃圾收集器在垃圾收集过程中同样也要暂停所有其他的工作线程。它是很多 java 虚拟机运行在Server模式下新生代的默认垃圾收集器。

常用对应 JVM 参数：`-XX:+UseParNewGC` 启用 ParNew 收集器，只影响新生代的收集，不影响老年代。

开启上述参数后，会使用：ParNew（Young区用）+ Tenured 的收集器组合，新生代使用复制算法，老年代采用标记-整理算法。

但是 ParNew + Tenured 这样的搭配， Java8已经不再被推荐；

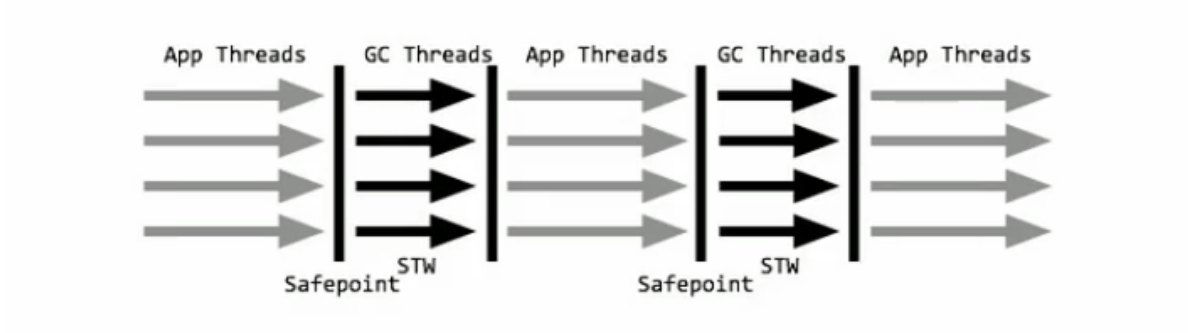
说明：

```
-XX:ParallelGCThreads=N 限制线程数量，默认开启和CPU数目相同的线程数。也可以通过N开启自定义的线程数
cpu>8    N = 5/8
cpu<8    N = 实际个数

-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+UseParNewGC
```

警告：

Java HotSpot(TM) 64-Bit Server VM warning: Using the ParNew young collector with the Serial old collector is deprecated and will likely be removed in a future release

并行回收GC（Parallel）/（Parallel Scavenge）



Parallel Scavenge 收集器类似 ParNew ，也是一个新生代垃圾收集器，使用复制算法，也是一个并行的多线程的垃圾收集器，俗称吞吐量优先收集器，一句话：串行收集器在新生代和老年代的并行化。

它重点关注的是：

可控制的吞吐量，比如程序运行100分钟，垃圾收集时间1分钟，吞吐量就是 99%。高吞吐量意味着高效利用CPU的时间，它多用于在后台运算而不需要太多交互的任务。

自适应调节策略也是 ParallelScavenge 收集器与 ParNew 收集器的一个重要区别。虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间（-XX：MaxGCPauseMillis）或最大的吞吐量。

常用JVM 参数：`-XX：+UseParallelGC` 或 `-XX：+UseParallelOldGC`（可互相激活）使用 ParallelScavenge 收集器。

开启该参数后：新生代使用复制算法，老年代使用标记-整理算法。

> 老年代

串行GC（Serial Old）/ （Serial MSC）

Serial Old 是 Serial 垃圾收集器老年代版本，它同样是单个单线程的收集器，使用标记-整理算法，这个收集器也主要是运行在Client默认的 java虚拟机默认的年老代垃圾收集器。

在Server模式下，主要有两个用途（了解，版本已经到8及以后）：

1、在 JDK 1.5 之前版本中与新生代的 Parallel Scavenge 收集器搭配使用。（Parallel Scavenge + Serial Old）

2、作为老年代版本中使用 CMS 收集器的后备垃圾收集方案。

并行GC（Parallel Old）/ （Parallel MSC）

Parallel Old 收集器是 Parallel Scavenge 的老年代版本，使用多线程的标记-整理算法，Parallel Old收集器在 JDK1.6才开始提供。

在 JDK1.6 之前，新生代使用 Parallel Scavenge 收集器只能搭配老年代的 Serial Old 收集器，只能保证新生代的吞吐量优先，无法保证整体的吞吐量。在 JDK 1.6 之前（ Parallel Scavenge + Serial Old）。

Parallel Old 正是为了在年老代同样提供吞吐量优先的垃圾收集器，如果系统对吞吐量要求比较高，JDK1.8后可以优先考虑新生代 Parallel Scavenge 和老年代 Parallel Old 收集器的搭配策略。在 JDK 1.8 及以后（ Parallel Scavenge + Parallel Old）

JVM 常用参数：

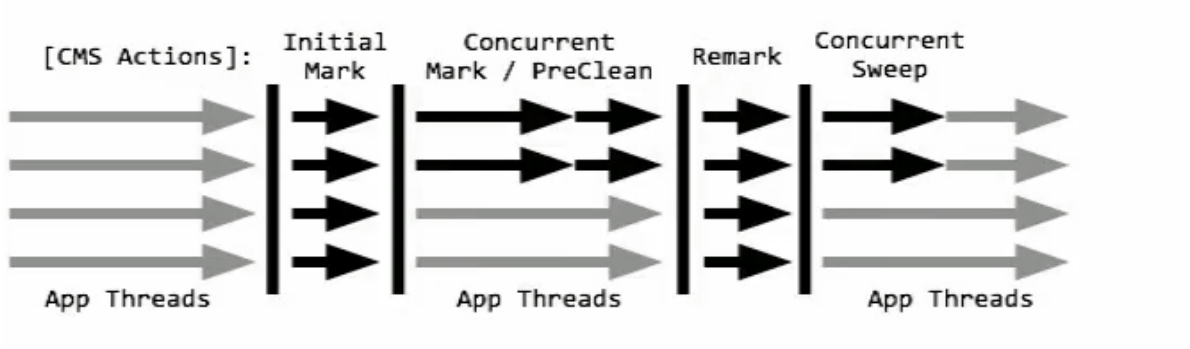
`-XX:+UseParallelOldGC` 使用Parallel Old 收集器，设置该参数后，新生代 Parallel Scavenge + 老年代 Parallel Old

并发标记清除GC（CMS）

CMS 收集器（Concurrent Mark Sweep：并发标记清除）是一种以获取最短回收停顿时间为目标的收集器。

适合应用在互联网站或者 B/S 系统的服务器上，这类应用尤其重视服务器的响应速度，希望系统停顿时间最短。

CMS 非常适合堆内存大、CPU核数多的服务器端应用，也是G1出现之前大型应用的首选收集器。



Concurrent Mark Sweep 并发标记清除，并发收集低停顿，并发指的是与用户线程一起执行。

开启该收集器的 JVM 参数：`-XX:+UseConcMarkSweepGC`，开启该参数后会自动将 -XX:+UseParNewGC 打开；

开启该参数后，使用 ParNew（Young区用）+CMS（Old区用）+ Serial Old 的收集器组合，Serial Old 将作为CMS出错的后备收集器。

```
-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:UseConcMarkSweepGC
```

优点：并发收集停顿低

缺点：

1、并发执行，对CPU资源压力大

```
由于并发进行，CMS在收集与应用线程会同时增加对内存的占用，也就是说，CMS必须要在老年代堆内存用尽之前完成垃圾回收
否则CMS回收失败时，将触发担保机制，串行老年代收集器将会以 STW 的方式进行一次 GC，从而造成较大停顿时间。
```

2、采用的标记清除算法会导致大量碎片

```
标记清除算法无法整理空间碎片，老年代空间会随着应用时长被逐步耗尽，最后将不得不通过担保机制对堆内存进行压缩。CMS也提供了参数
-XX:CMSFullGCsBeForeCompaction(默认0，即每次都进行内存整理) 来指定多少次 CMS 收集之后，进行一次压缩的 Full GC.
```

> GC 之如何选择垃圾收集器

1、单CPU或小内存，单机程序

`-XX:+UseSerialGC`

2、多CPU，需要最大吞吐量，如后台计算型应用

`-XX:+UseParallelGC` 或者 `-XX:+UseParakkekOldGC`

3、多CPU，追求低停顿时间，需要快速响应如互联网应用

`-XX:+UseConcMarkSweepGC` 或 `-XX:+ParNewGC`

参数	新生代垃级收集器	新生代算法	老年代垃圾收集器	老年代算法
-XX:+UseSerialGC	SerialGC	复制	SerialOldGC	标整
-XX:+UseParNewGC	ParNew	复制	SerialOldGC	标整
-XX:+UseParallelGC/-XX:+UseParallelOldGC	Parallel[Scavenge]	复制	Parallel Old	标整
-XX:+UseConcMarkSweepGC	ParNew	复制	CMS + Serial Old的收集器组合 (Serial Old作为CMS出错的后备收集器)	标清
-XX:+UseG1GC	G1整体上采用标记-整理算法	局部是通过复制算法，不会产生内存碎片。		

8、G1垃圾收集器

> 以前收集器特点

1、年轻代和老年代是各自独立且连续的内存块；

2、年轻代收集使用单 eden + s0 + s1 进行复制算法；

3、老年代收集必须扫描整个老年代区域；

4、都是以尽可能少而快速地执行 GC 为设计原则。

> G1 是什么?

G1（Garbage-First）收集器，是一款面向服务端应用的收集器；

The Garbage-First (G1) collector is a server-style garbage collector, targeted for multi-processor machines with large memories. It meets garbage collection (GC) pause time goals with a high probability, while achieving high throughput. The G1 garbage collector is fully supported in Oracle JDK 7 update 4 and later releases. The G1 collector is designed for applications that:

- Can operate concurrently with applications threads like the CMS collector.
- Compact free space without lengthy GC induced pause times.
- Need more predictable GC pause durations.
- Do not want to sacrifice a lot of throughput performance.
- Do not require a much larger Java heap.

从官网的描述中，我们知道 G1 是一种服务器端的垃圾收集器，应用在多处理器和大容量内存环境中，在实现高吞吐量的同时，尽可能的满足垃圾收集暂停时间的要求。另外，它还具有以下特性：

1、像CMS收集器一样，能与应用程序线程并发执行。

2、整理空闲空间更快

3、需要更多的时间来预测GC停顿时间。

4、不希望牺牲大量的吞吐性能。

5、不需要更大的 Java Heap。

G1 收集器的设计目标是取代 CMS 收集器，它同 CMS 相比，在以下方面表现的更出色：

G1 是一个有整理内存过程的垃圾收集器，不会产生很多内存碎片。

G1 的 Stop The World 更可控，G1在停顿时间上添加了预测机制，用户可以指定期望停顿时间。

CMS 垃圾收集器虽然减少了暂停应用程序的运行时间，但是它还存在着垃圾碎片问题。于是，为了去除内存碎片问题，同时又保留 CMS 垃圾收集器低暂停时间的优点，Java7发布了一个新的垃圾收集器-G1垃圾收集器。

G1 是在 2012 年才在JDK1.7u4 中可用，oralce 官方计划在 jdk9 中将G1变成默认的垃圾收集器以替代 CMS。它是一款面向服务端应用的收集器，主要应用在多CPU和大内存服务器环境下，极大的减少垃圾收集的停顿时间，全面提升服务器的性能，逐步替换 java8以前的 CMS 收集器。

主要改变是 Eden，Survivor 和 Tenured 等内存区域不再是连续的了，而是变成了一个个大小一样的 region，每个region从1M到32M不等。一个 region有可能属于Eden，Survivor 或者 Tenured 内存区域。

> 特点

1、G1能充分利用多CPU、多核环境硬件优势，尽量缩短 STW。

2、G1整体上采用标记-整理算法，局部是通过复制算法，不会产生内存碎片。

3、宏观上G1之中不再区分年轻代和老年代，把内存划分成多个独立的子区域（Region），可以近似理解为一个围棋的棋盘。

4、G1收集器里面将整个的内存区都混合在一起了，但其本身依然在小范围内要进行年轻代和老年代的区分，保留了新生代和老年代，但他们不再是物理隔离的，而是一部分Region的集合且不需要 Region 是连续的。也就是说依然会采用不同的GC方式来处理不同的区域。

5、G1虽然也是分代收集器，但整个内存分区不存在物理上的年轻代与老年代的区别，也不需要完全独立的 survivor（to space）堆做复制准备。G1只有逻辑上的分代概念，或者说每个分区都可能随G1的运行在不同代之间前后切换。

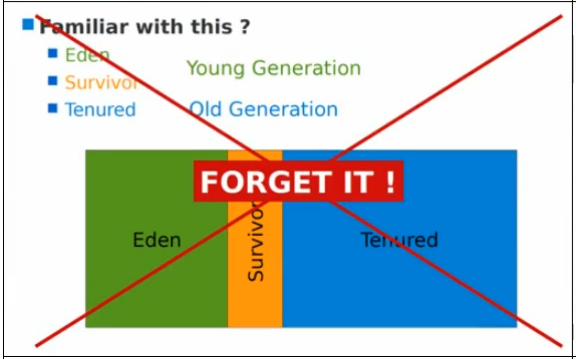
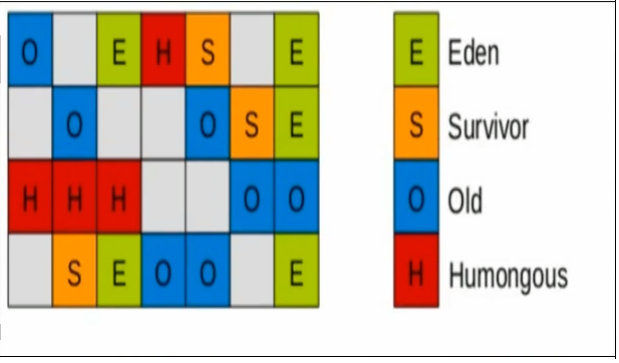
> 底层原理

Region区域化垃圾收集器：最大好处是化整为零，避免全内存扫描，只需要按照区域来进行扫描即可。

区域化内存划片 Region，整体编为了一系列不连续的内存区域，避免了全内存区的GC操作。

核心思想是将整个堆内存区域分成大小相同的子区域（Region），在 JVM 启动时会自动设置这些子区域的大小，在堆的使用上，G1并不要求对象的存储一定是物理上连续的，只要逻辑上连续即可。每个分区也不会固定地为某个代服务，可以按需在年轻代和老年代之间切换。启动时可以通过参数 -XX:G1HeapRegionSize=n可指定分区大小（1MB~32MB，且必须是2的幂），默认将整堆划分为2048个分区。

大小范围在 1MB ~ 32MB，最多能设置 2048 个区域，也即能够支持的最大内存为：32MB * 2048 = 64G内存！

<table>
<tr><td>G1将新生代，老年代的物理空间划分取消了。</td><td>G1算法将堆划分为若干个区域（Region）</td></tr>
<tr><td> 
</td><td> 
</td></tr>
</table>

G1算法将堆划分为若干个区域（Region），它仍然属于分代收集器

这些Region的一部分包含新生代，新生代的垃圾收集依然采用暂停所有应用线程的方式，将存活对象拷贝到老年代或者Survivor空间。

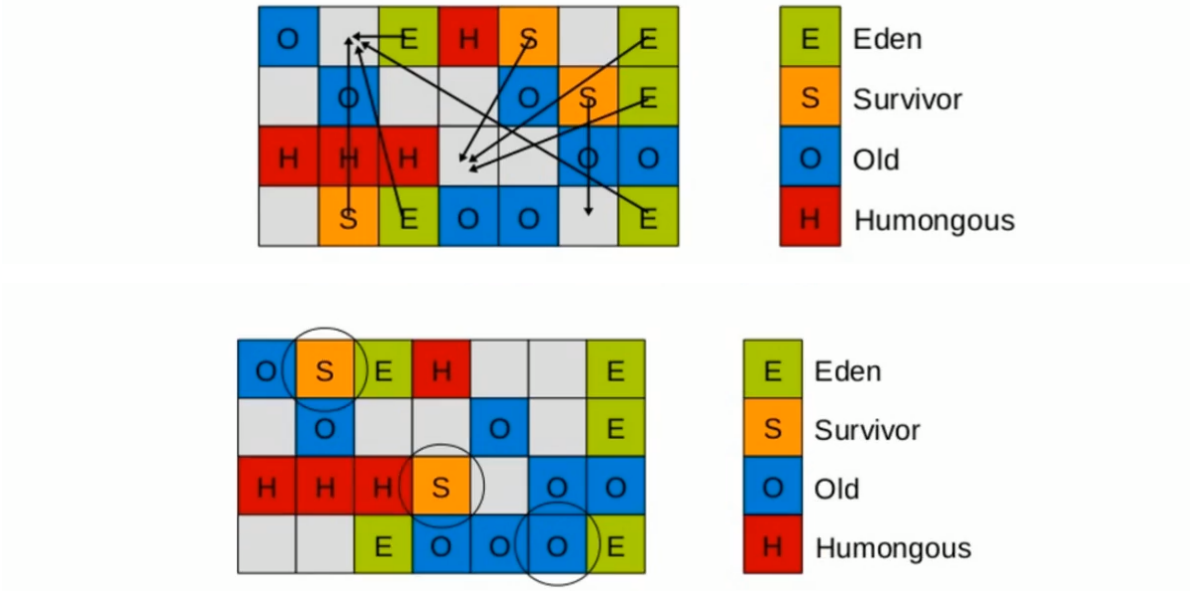
这些Region的一部分包含老年代，G1收集器通过将对象从一个区域复制到另外一个区域，完成了清理工作。这就意味着，在正常的处理过程中，G1完成了堆的压缩（至少是部分堆的压缩），这样也就不会有CMS内存碎片问题的存在了。

在G1中，还有一种特殊的区域，叫Humongous(巨大的)区域
如果一个对象占用的空间超过了分区容量50%以上，G1收集器就认为这是一个巨型对象。这些巨型对象默认直接会被分配在年老代，但是如果它是一个短期存在的巨型对象，就会对垃圾收集器造成负面影响。为了解决这个问题，G1划分了一个Humongous区，它用来专门存放巨型对象。如果一个H区装不下一个巨型对象，那么G1会寻找连续的H分区来存储。为了能找到连续的H区，有时候不得不启动Full GC。

> 回收步骤

针对Eden区进行收集，Eden区耗尽后会被触发，主要是小区域收集 + 形成连续的内存块，避免内存碎片

* Eden区的数据移动到Survivor区，假如出现Survivor区空间不够，Eden区数据会部会晋升到Old区
* Survivor区的数据移动到新的Survivor区，部会数据晋升到Old区
* 最后Eden区收拾干净了， GC结束，用户的应用程序继续执行。



> 常用配置参数

开发人员仅仅需要声明以下参数即可：

开启G1=>设置最大内存=>设置最大停顿时间

```
-XX:+UseG1GC -Xmx32g -XX:MaxGCPauseMillis=100
```

`-XX:MaxGCPauseMillis=n` 最大GC停顿时间单位毫秒，JVM将尽可能（不保证）停顿小于这个时间。

> 和 CMS 相比的优势

1、G1不会产生内存碎片

2、是可以精确控制停顿，该收集器是把整个堆（新生代、老年代）划分成多个固定大小的区域，每次根据允许停顿的时间去收集垃圾最多的区域。