一、前景提示

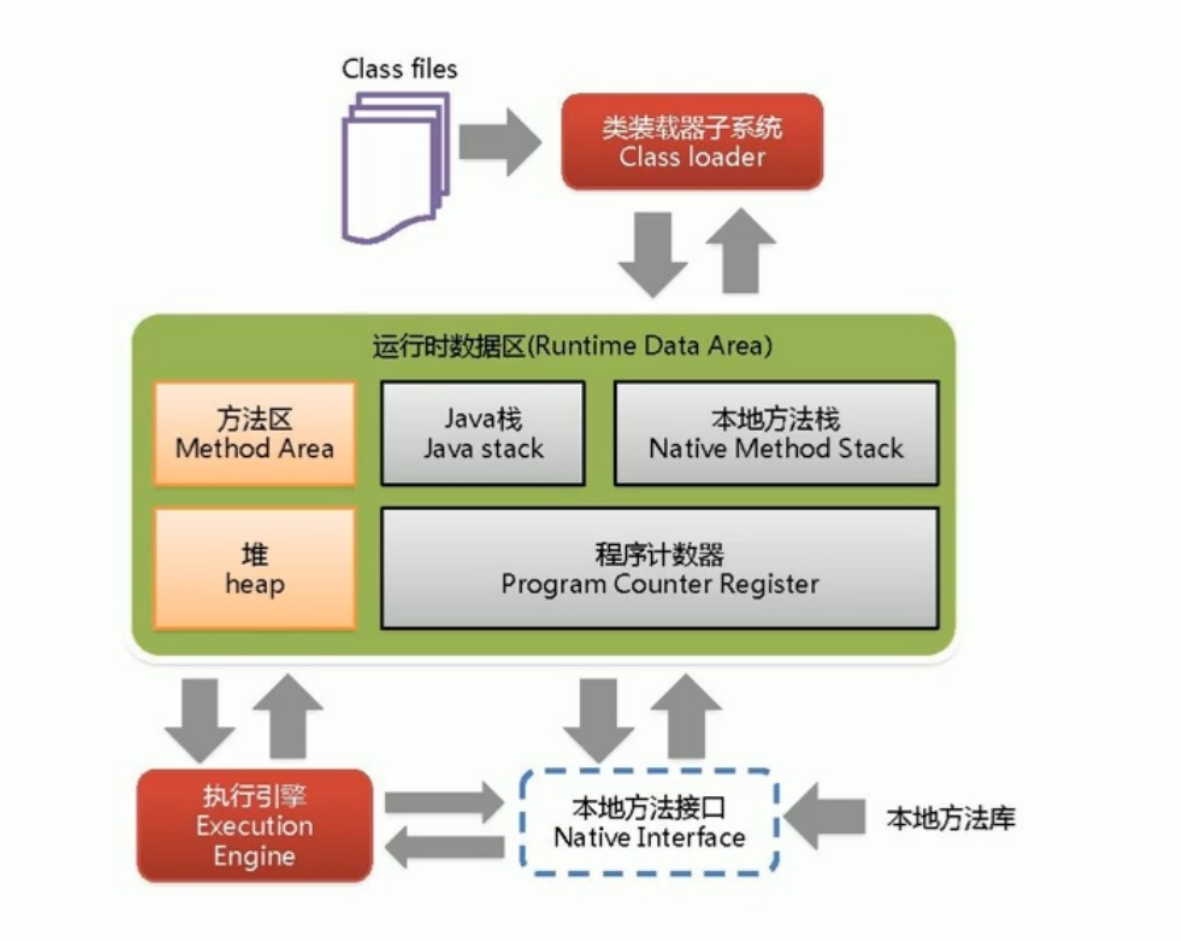
> 1. 请谈谈你对JVM的理解？java8的虚拟机有什么更新？
> 2. 什么是OOM？什么是StackOverFlowError？有哪些方法分析？
> 3. JVM的常用参数调优你知道哪些？
> 4. 谈谈JVM中，对类加载器你的认识？

二、JVM核心

1、JVM体系结构概述

> JVM是运行在操作系统之上的，它与硬件没有直接的交互

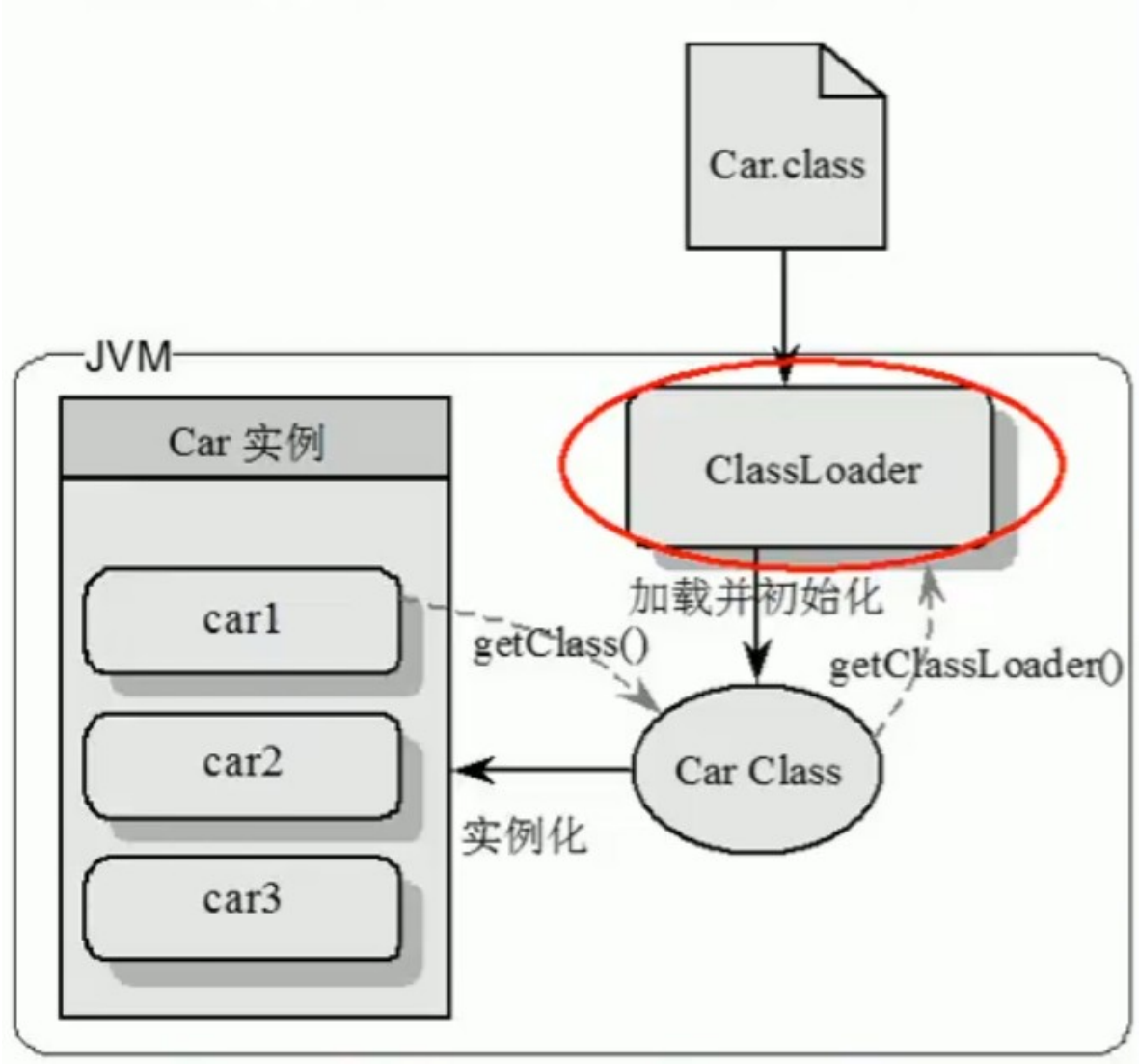
1.1 结构图



1.2 类装载器（ClassLoader）

1.2.1 介绍

> 负责加载class文件，**class文件再文件开头有特定的文件标识**，将class文件字节码内容加载到内存中，并将这些内容转换成非法区中的运行时数据结构并且ClassLoader只负责class文件的加载，至于它是否可以运行，则由Execution Engine决定。



> echo %JAVA_HOME%
>
> echo %PATH%
>
> echo %CLASSPATH%

1.2.2 类加载器的种类

- **虚拟机自带的加载器**
  1. **启动类加载器**（Bootstrap）C++

     > $JAVAHOME/jre/lib/rt.jar 包里面的类

     ```
     Object o = new Object();
     System.out.println(o.getClass().getClassLoader()); // null
     ```

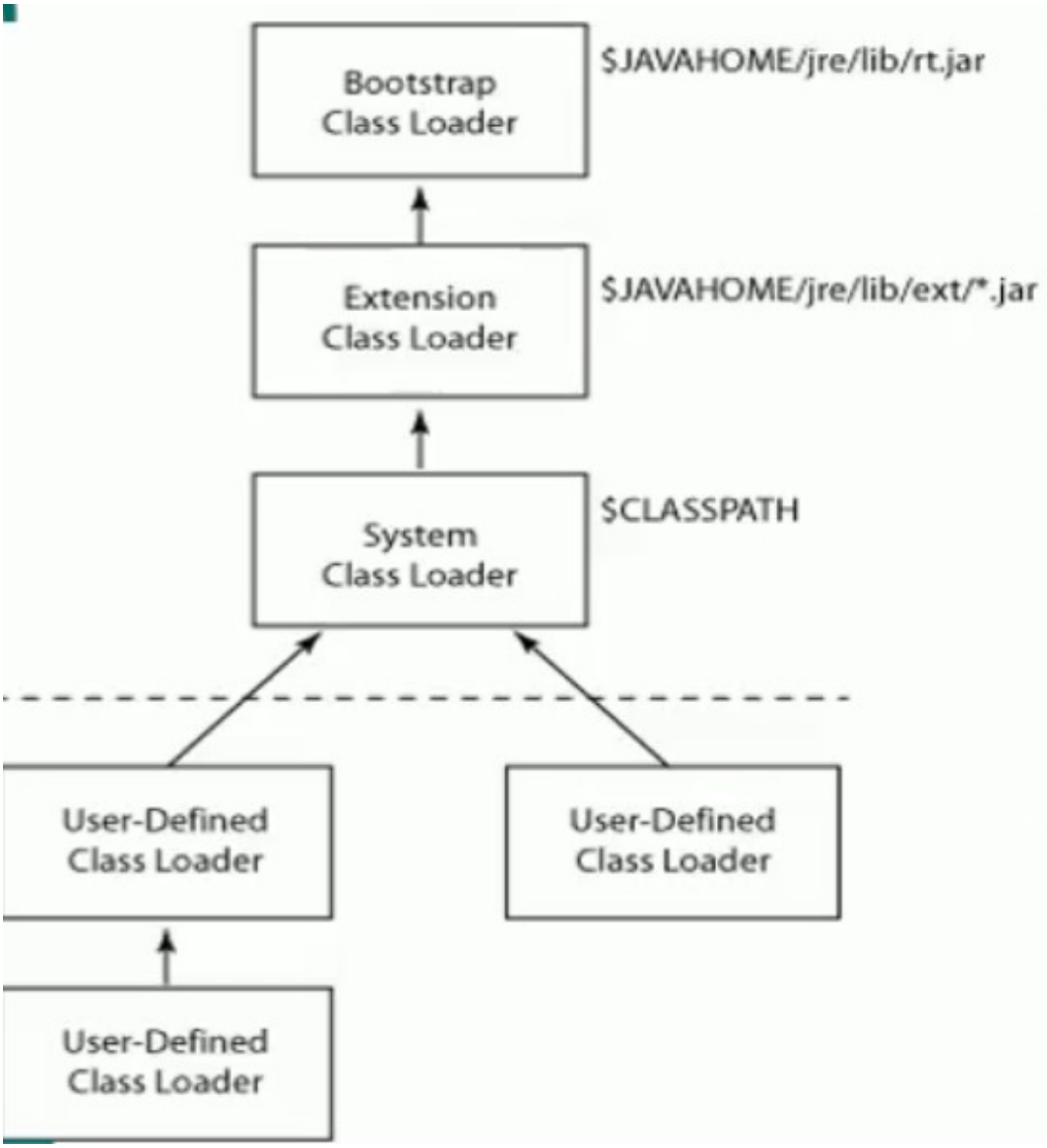
  2. **扩展类加载器**（Extension）Java

     > $JAVAHOME/jre/lib/ext/*.jar 里面的jar包

  3. **应用程序类加载器**（AppClassLoader）

     > $CLASSPATH

```
MyObject m = new MyObject();
// null
System.out.println(m.getClass().getClassLoader().getParent().getParent(
));
// sun.misc.Launcher$ExtClassLoader@2cdf8d8a
System.out.println(m.getClass().getClassLoader().getParent());
// sun.misc.Launcher$AppClassLoader@58644d46
System.out.println(m.getClass().getClassLoader());
```



- 用户自定义加载器（了解）
    1. java.lang.ClassLoader的子类，用户可以定制类的加载器

1.2.3 类加载器的双亲委派机制、沙箱安全机制

> **双亲委派**：
>
> 当一个类收到了类加载请求，他首先不会尝试自己去加载这个类，而是把这个请求委派给父类去完成，每一个层次类加载器都是如此，因此所有的加载请求都应该传送到启动类加载器中，只有当父类加载器反馈自己无法完成这个请求的时候（在它的加载路径下没有找到所需加载的Class），子类加载器才会尝试自己去加载。
>
> **沙箱安全**：

> 采用双亲委派的一个好处是比如加载位于rt.jar包中的类java.lang.Object，不管是哪个加载器加载这个类，最终都是委托给顶层的启动类加载器进行加载，这样就保证了使用不同的类加载器最终得到的都是同样一个Object对象。

1.3 本地方法栈（Native Method Stack）【了解】

> **native** 方法：
>
> 目前该方法使用越来越少，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。
>
> 它的具体做法是Native Method Stack 中登记native方法，在Execution Engine 执行时加载本地方法库。

```
new Thread(() -> {
    try {
        TimeUnit.SECONDS.sleep(2);
        System.out.println("Hello Native...");
    } catch (InterruptedException e) {
        e.printStackTrace();
    };
},"A").start();
// start()方法源码
// ...
start0();
// ...
private native void start0();
```

1.4 PC寄存器（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（用来存储指向下一条指令的地址，也即将要执行的指令代码），由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。
>
> 这块内存区域很小，它是当前线程所执行的字节码的行号指示器，字节码解释器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。
>
> 如果执行的是一个Native方法，那这个计数器是空的。
>
> 用以完成分支，循环，跳转，异常处理，线程恢复等基础功能。不会发生内存溢出（OutOfMemory）OOM错误。

1.5 知识小总结：

> - JVM系统架构图
> - 类加载器
>
> ```
> // null
> System.out.println(m.getClass().getClassLoader().getParent().getParent());
> // sun.misc.Launcher$ExtClassLoader@2cdf8d8a
> System.out.println(m.getClass().getClassLoader().getParent());
> // sun.misc.Launcher$AppClassLoader@58644d46
> System.out.println(m.getClass().getClassLoader());
> ```
>
> 1. 有哪几种类加载器
> 2. 双亲委派
> 3. 沙箱安全机制

- Native
    1. native 是一个关键字
    2. 有声明，无实现，why ？Native Method Stack
- PC寄存器

记录了方法之间的调用和执行情况，类似排班值日表

用来存储指向下一条指令的地址，也即将要执行的指令代码

他是当前线程所执行的字节码的行号指示器

- 方法区

它存储了每一个类的结构信息

方法区是规范

- stack

栈管运行，堆管存储

栈保存哪些对象？

1.6 方法区（Method Area）

供各线程共享的运行时内存区域。**它存储了每一个类的结构信息**，例如运行时常量池（Runtime Constant Pool）、字段和方法数据、构造函数和普通方法的字节码内容。

方法区是规范，在不同虚拟机里头的实现是不一样的，最典型的就是永久代（PermGen space）和元空间（Metaspace）。

实例变量存在堆内存中，和方法区无关

1.7 java栈（Java stack）

1.7.1 介绍

程序 = 算法 + 数据结构

程序 = 框架 + 业务逻辑

两种数据结构：

- 队列（FIFO）：先进先出（排队打饭）
- 栈 （FILO） ：先进后出（枪支弹夹）

方法 = 栈帧

Stack栈

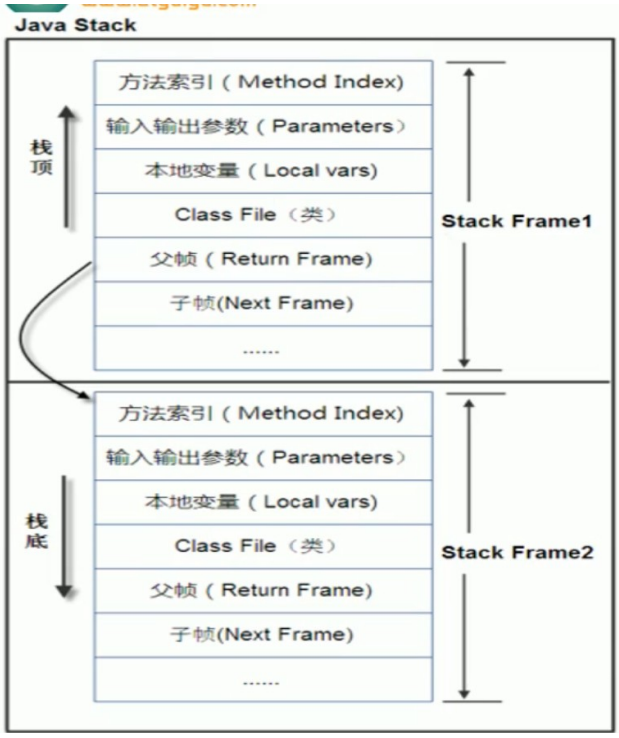
栈也叫栈内存，主管Java程序的运行，是在线程创建时创建，它的生命周期是跟随线程的生命周期，线程结束栈内存也就释放，**对于栈来说不存在垃圾回收机制**，只要线程一结束该栈就over，生命周期和线程是一致的，是线程私有的。

8种基本类型的变量 + 对象的引用变量 + 实例方法都是在函数的栈内存中分配

栈存储什么？

- 本地变量（Local Variables）：输入参数和输出参数以及方法内的变量
- 栈操作（Operand Stack）：记录出站、入栈的操作
- 栈帧数据（Frame Data）：包括类文件、方法等等

图示：



图示在一个栈中有两个栈帧：

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。

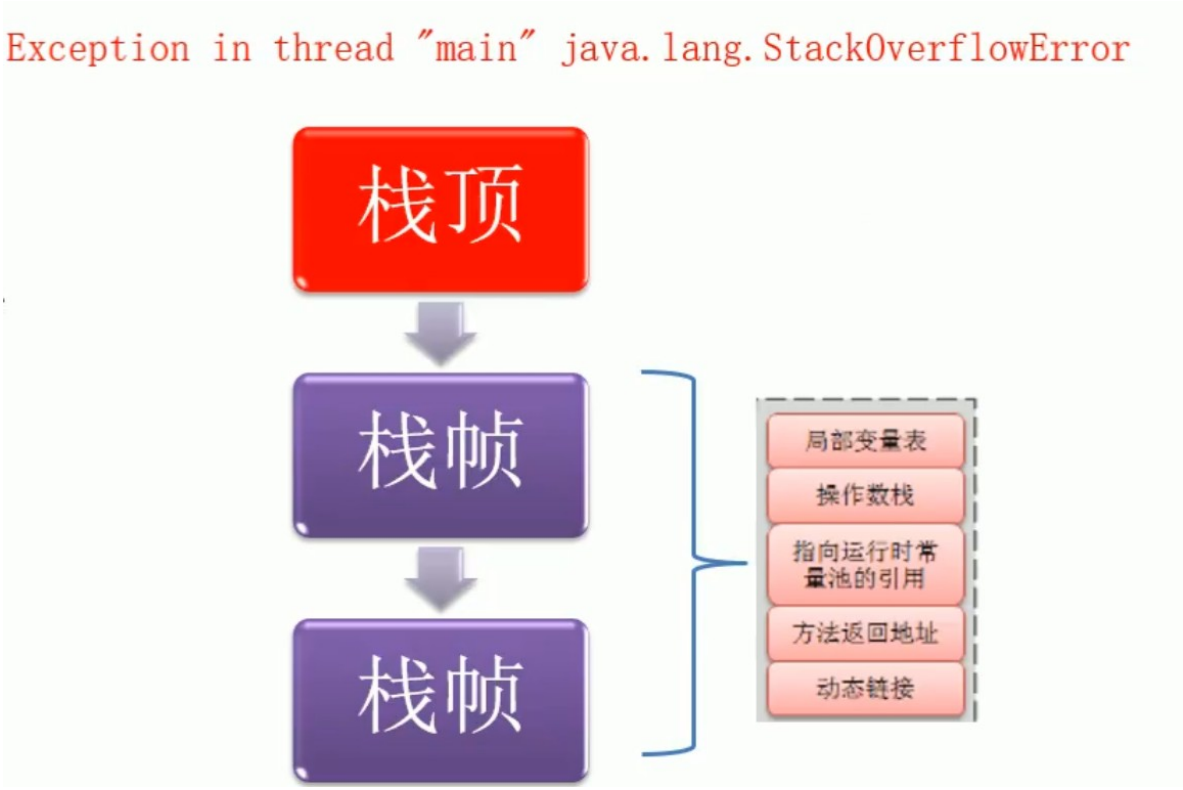
每执行一个方法都会产生一个栈帧，保存到栈(后进先出)的**顶部，顶部栈就是当前的方法，该方法执行完毕 后会自动将此栈帧出栈。**

1.7.2 StackOverflowError

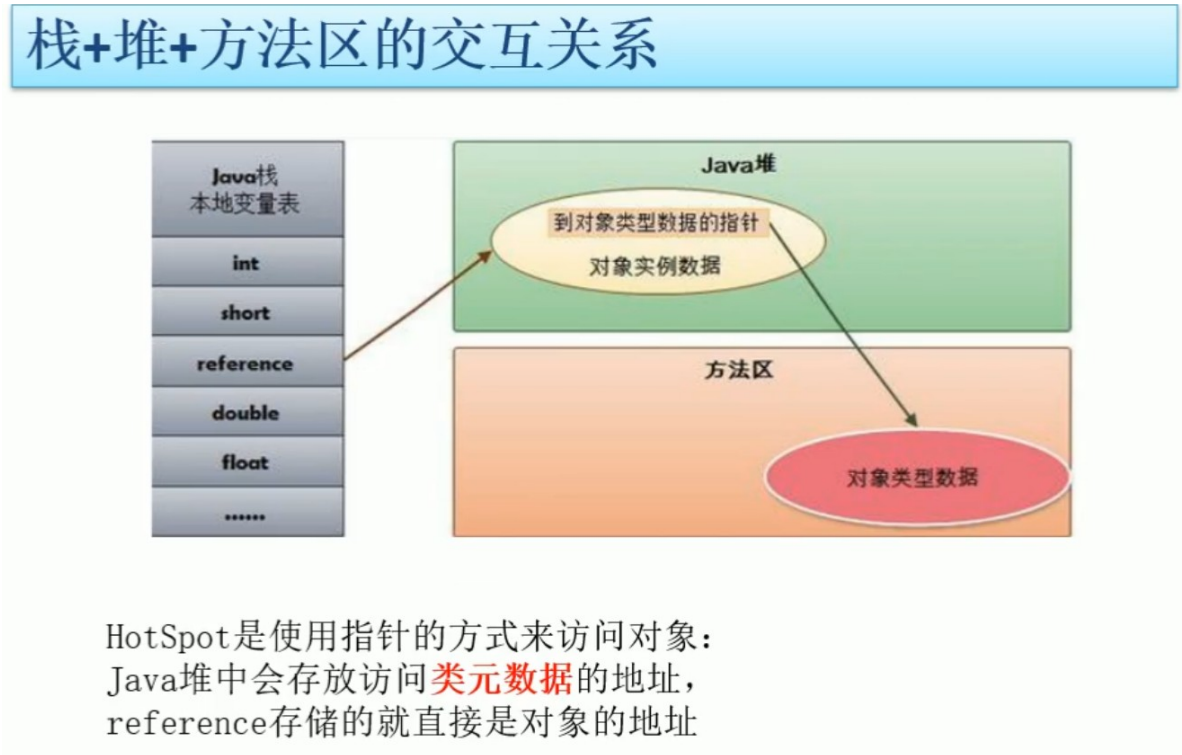
```
public class JVM {

    public void t() {
        t();
    }
    // Exception in thread "main" java.lang.StackOverflowError
    public static void main(String[] args) {
        JVM jvm = new JVM();
        jvm.t();
    }
}
```

1. java.lang.StackOverflowError是异常还是错误？

   是错误



1.8 栈+堆+方法区的交互关系



2、堆体系结构概述

> 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。
>
> 堆内存逻辑上分为三部分：新生+养老+永久

> 物理上：新生+养老

2.1 新生区

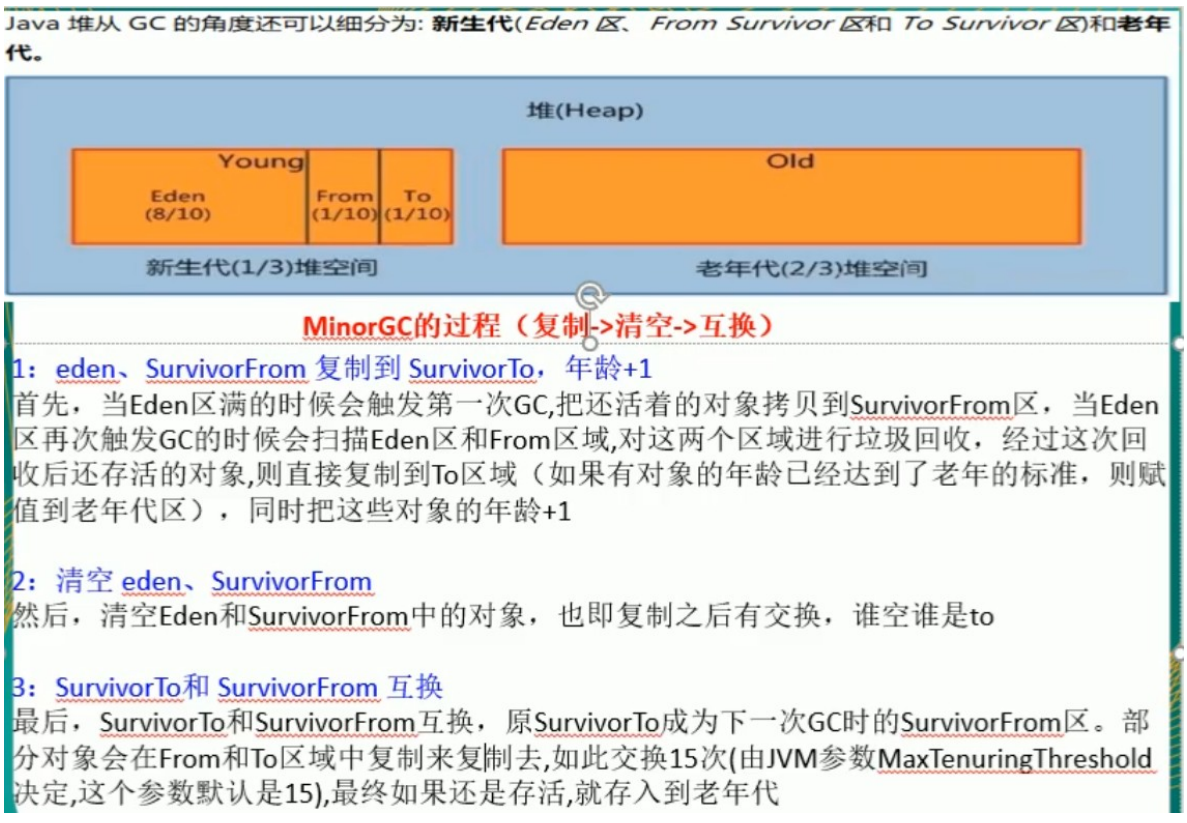
> 新生区是类的诞生，成长，消亡的区域，一个类在这里产生，应用，最后被垃圾回收器收集，结束生命。
>
> 新生区又分为两部分：伊甸区（Eden space）和幸存者区（Survivor space），所有的类都是在伊甸区被new出来的。幸存区有两个：0区和1区。
>
> 当伊甸区的空间用完时，程序又需要创建对象，JVM的垃圾回收器将对伊甸区进行垃圾回收（Minor GC）
>
> 将伊甸区中的不再被其他对象引用的对象进行销毁。然后将伊甸区中的剩余对象移动到幸存0区，若幸存0区也满了，再对该区进行垃圾回收，然后移动到1区，若1区也满了，再移动到养老区，若养老区也满了，那么这个时候讲产生Major GC（FullGC），进行养老区的内存清理。若养老区执行了Full GC之后发现依然无法进行对象的保存，将产生OOM（OutOfMemoryError）异常。
>
> 如果出现java.lang.OutOfMemoryError：java heap space异常，说明Java虚拟机的堆内存不够，原因有二：
>
> 1、**Java虚拟机的堆内存设置不够，可以通过参数 -Xms、-Xmx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）。**

2.1.1 伊甸区（Eden Space）

2.1.2 幸存0区/from（Survivor 0 Space）

2.1.3 幸存1区/to（Survivor 1 Space）

2.1.4 对象生命周期及GC



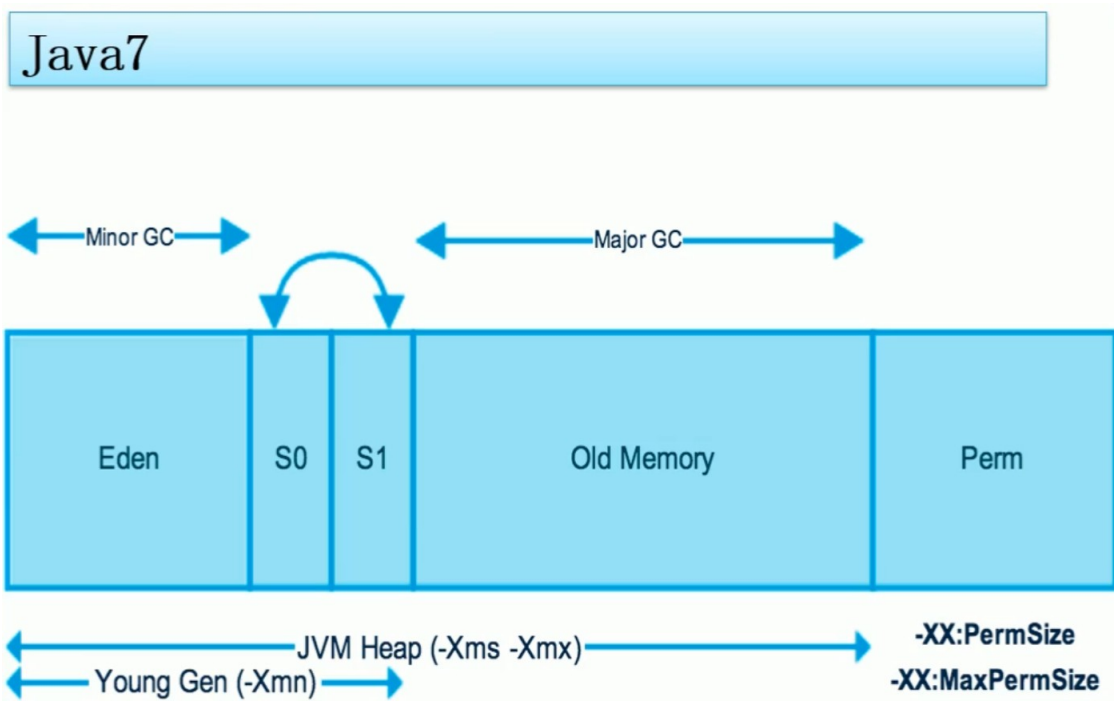
2.2 养老区（Tenure Generation Space）

2.3 元空间/永久代（Permanent Space）

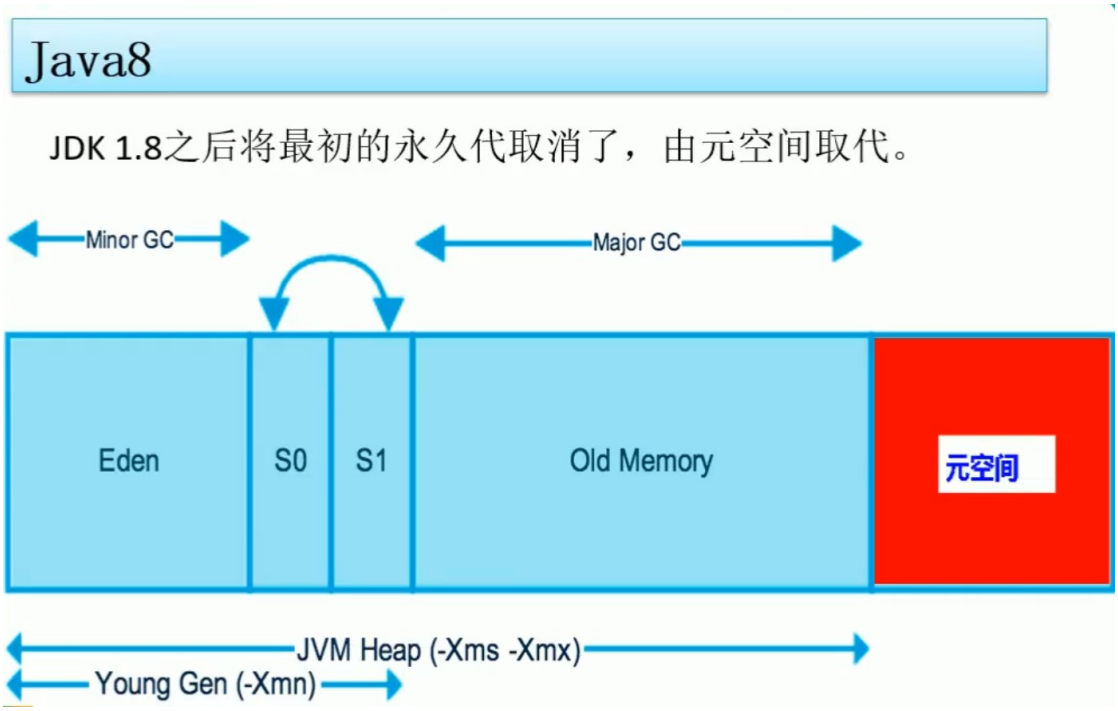
> 方法区的实现

3.堆参数调优入门

> JDK7的堆模型



> JDK8的堆模型



解读：

- GC发生在新生区
- Full GC发生在养老区
- 新生区/养老区 = 1/2
- Eden/S0/S1 = 8/1/1
- 调优参数 -Xms 及 -Xmx

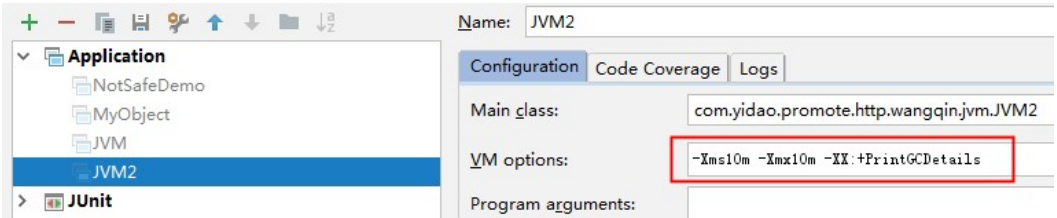
> JDK7与JDK8的区别：

> 在Java8中，永久代已经被移除，被一个称为元空间的区域所取代，元空间的本质和永久代类似。
>
> 元空间与永久代之间最大的区别在于：
>
> 永久代使用的JVM的堆内存，但是Java8以后的**元空间并不在虚拟机中而是使用本机物理内存**。
>
> 因此，默认情况下，元空间的大小仅受本地内存限制。类的元数据放入 native memory，字符串池和类的静态变量放入java堆中，这样可以加载多少类的元数据就不再由MaxPermSize控制，而由系统的实际可用空间来控制。

> • （堆内存调优简介01）
>
> | -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 |
>
> ```
> public static void main(String[] args){
> long maxMemory = Runtime.getRuntime().maxMemory() ;//返回Java 虚拟机试图使用的最大内存量。
> long totalMemory = Runtime.getRuntime().totalMemory() ;//返回Java 虚拟机中的内存总量。
> 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");
> }
> ```

3.1 内存参数配置（-Xms初始内存大小，-Xmx最大内存）

> -Xms1024m -Xmx1024m -XX:+PrintGCDetails
>
> - 初始内存和最大内存怎么配？
>
> 1. **Xms和Xmx的值必须设置一样大**
>
>   理由：避免GC和应用程序争抢内存，理论值的峰值和峰谷忽高忽低。
>
> 2. 怎么配？（IDEA）



3.2 查看heap结构（-XX:+PrintGCDetails）

```
// CPU核数
System.out.println(Runtime.getRuntime().availableProcessors());
// 最大内存
System.out.println("Xmx:"+Runtime.getRuntime().maxMemory()/(double)1024/1024);
// 初始内存
System.out.println("Xms:"+Runtime.getRuntime().totalMemory()/(double)1024/1024);
```

```
4
```

```
Xmx:981.5
Xms:981.5
Heap
// 新生区
PSYoungGen      total 305664K, used 20972K [0x00000000eab00000,
0x0000000100000000, 0x0000000100000000)
// 伊甸区
eden space 262144K, 8% used
[0x00000000eab00000,0x00000000ebf7b048,0x00000000fab00000)
// from区
from space 43520K, 0% used
[0x00000000fd580000,0x00000000fd580000,0x0000000100000000)
// to区
to   space 43520K, 0% used
[0x00000000fab00000,0x00000000fab00000,0x00000000fd580000)
// 养老区
ParOldGen       total 699392K, used 0K [0x00000000c0000000,
0x00000000eab00000, 0x00000000eab00000)
object space 699392K, 0% used
[0x00000000c0000000,0x00000000c0000000,0x00000000eab00000)
// 元空间
Metaspace       used 3264K, capacity 4496K, committed 4864K, reserved
1056768K
class space    used 350K, capacity 388K, committed 512K, reserved 1048576K
```

> 物理上：新生区 + 养老区
>
> 699392K + 305664K = 981.5M

3.3 OOM (java.lang.OutOfMemoryError: Java heap space)

> -Xms10m -Xmx10m -XX:+PrintGCDetails

```
byte[] bytes = new byte[40 * 1024 *1024];
```

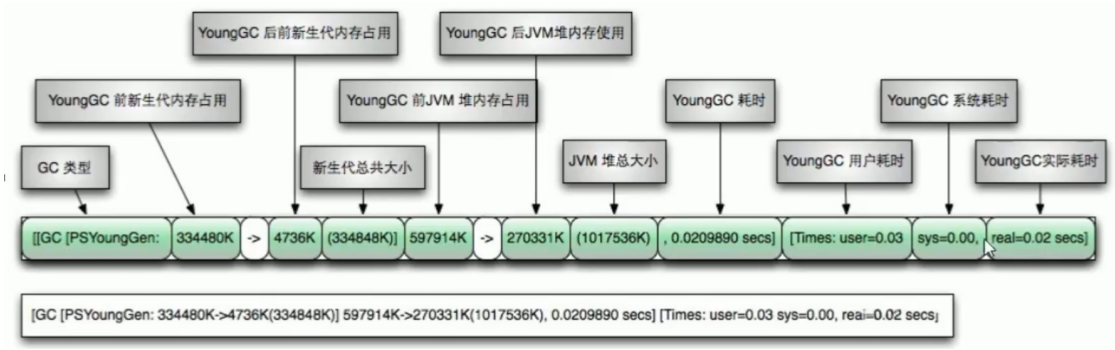
```
[GC (Allocation Failure) [PSYoungGen: 2048K->484K(2560K)] 2048K-
>1011K(9728K), 0.0021184 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 2532K->500K(2560K)] 3059K-
>1555K(9728K), 0.0015301 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 850K->500K(2560K)] 1905K-
>1603K(9728K), 0.0012114 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 500K->500K(2560K)] 1603K-
>1619K(9728K), 0.0064274 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
[Full GC (Allocation Failure) [PSYoungGen: 500K->0K(2560K)] [ParOldGen:
1119K->1482K(7168K)] 1619K->1482K(9728K), [Metaspace: 3272K-
>3272K(1056768K)], 0.0605470 secs] [Times: user=0.05 sys=0.00, real=0.06
secs]
[GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] 1482K->1482K(9728K),
0.0092194 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
[Full GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] [ParOldGen:
1482K->1454K(7168K)] 1482K->1454K(9728K), [Metaspace: 3272K-
>3272K(1056768K)], 0.0114834 secs] [Times: user=0.03 sys=0.00, real=0.01
secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at com.yidao.promote.http.wangqin.jvm.JVM2.main(JVM2.java:21)
```

3.4 解读GC日志

> **公式：[名称：GC前内存占用 -> GC后内存占用 （ 该区内存总大小 ）]**

- GC

```
[GC (Allocation Failure)
[PSYoungGen: 2048K->484K(2560K)] 2048K->1011K(9728K), 0.0021184 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
```



- Full GC

```
[Full GC (Allocation Failure)  // GC 类型 Full GC
// 新生区：GC前0k -> GC后0k （新生区总大小2560k）
[PSYoungGen: 0K->0K(2560K)]
// 养老区：GC前1482k -> GC后1454k（养老区总大小7168k）|FullGC前堆内存占用1482k -> FullGC后内存占用1454k（堆内存总大小9728k = 2560k[新生区总大小]+7168k[养老区总大小]）
[ParOldGen: 1482K->1454K(7168K)] 1482K->1454K(9728K),
// 元空间：GC前内存占用 -> GC后内存占用（元空间总大小） 耗时
[Metaspace: 3272K->3272K(1056768K)], 0.0114834 secs]
//       用户耗时    需要耗时    实际耗时
[Times: user=0.03 sys=0.00, real=0.01 secs]
```

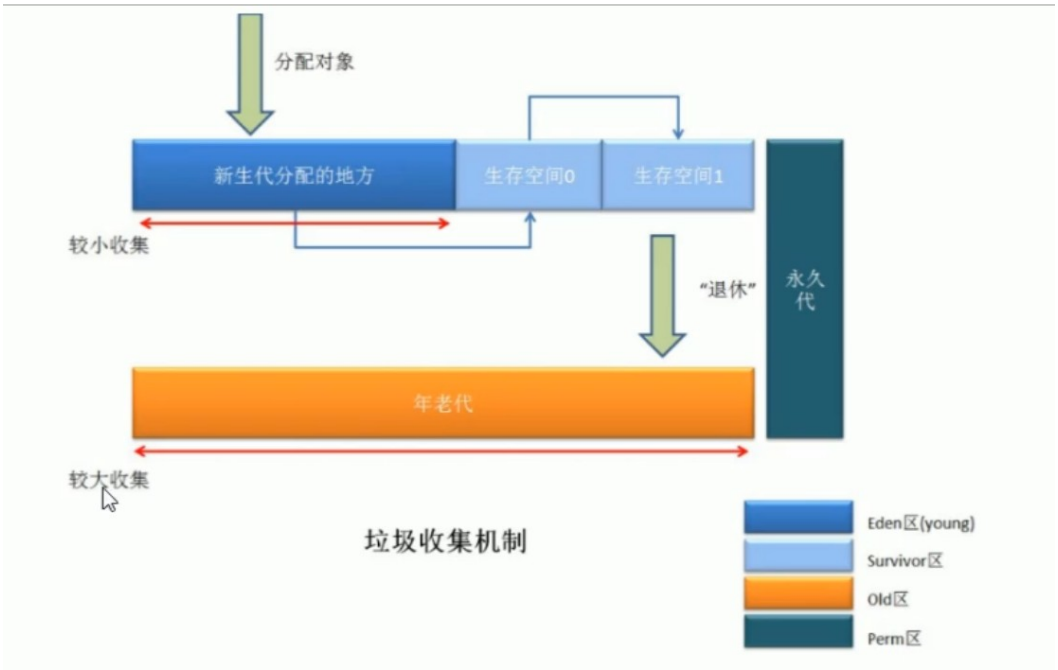
3.5 GC

3.5.1 GC是什么（*分代收集算法*）

1. 次数上频繁收集Young区
2. 次数上较少收集Old区
3. 基本不动元空间

3.5.2 GC4大算法

- GC算法概述



1. 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。
2. 因此GC按照回收的区域又分为两种类型，一种是普通GC（minor GC），一种是全局GC（major GC or Full GC）

Minor GC 和 Major GC的区别：

> **普通GC（minor GC）：只针对新生代区域的GC，指发生在新生代的垃圾收集动作，因此大多数Java对象存活率都不高，所以Minor GC 非常频繁，一般回收速度也比较快。**
>
> **全局GC（major GC or Full GC）：指发生在老年代的垃圾收集动作，出现了Major GC，经常会伴随至少一次Minor GC（并不是绝对的）。Major GC的速度一般比Minor GC慢上10倍以上。**
>
> Full GC为什么比GC慢？
>
> 普通GC 发生在新生区，Full GC发生在养老区，根据内存占用比例：1：2，Full GC发生至少会伴随至少一次的普通GC，区域更大，因此会慢。

- 4大算法
  1. 引用计数法（了解）

循环引用问题：

```
public class RefCountGC {

    private byte[] bytes = new byte[2 * 1024 *1024];
    Object instance = null;
```

```
    public static void main(String[] args) {
        RefCountGC o1 = new RefCountGC();
        RefCountGC o2 = new RefCountGC();
        o1.instance = o2;
        o2.instance = o1;
        o1 = null;
        o2 = null;
        System.gc();
    }
}
```

2. 复制算法（Copying）

- **年轻代中使用的是Minor GC，这种GC算法采用的是复制算法（Copying）**
- 原理

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到Old generation中，也即一旦收集后，Eden是就变成空的了。

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存活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 from 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，就将对象的年龄 + 1，当对象的年龄达到某个值时（默认是 15 岁，通过-XX:MaxTenuringThreshold 来设定参数），这些对象就会成为老年代。

-XX:MaxTenuringThreshold — 设置对象在新生代中存活的次数

- 解释

年轻代中的GC,主要是复制算法（Copying）

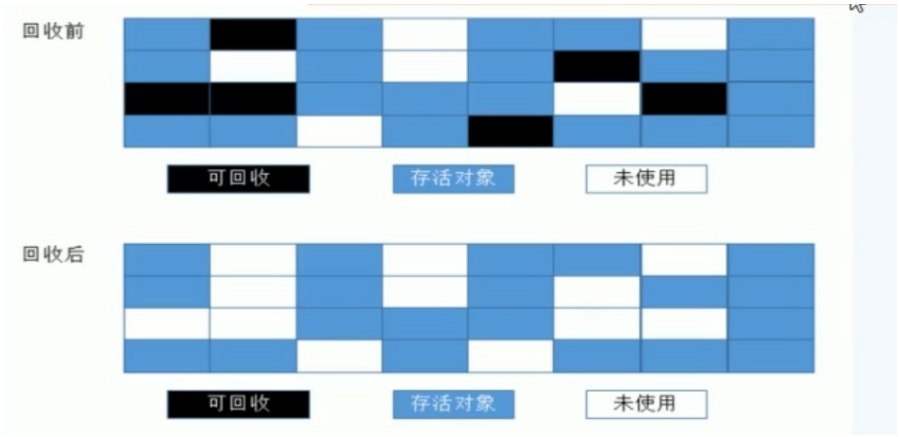
HotSpot JVM把年轻代分为了三部分：1个Eden区和2个Survivor区（分别叫from和to）。默认比例为8:1:1,一般情况下，新创建的对象都会被分配到Eden区(一些大对象特殊处理),这些对象经过第一次Minor GC后，如果仍然存活，将会被移到Survivor区。对象在Survivor区中每熬过一次Minor GC，年龄就会增加1岁，当它的年龄增加到一定程度时，就会被移动到年老代中。因为年轻代中的对象基本都是朝生夕死的(90%以上)，所以在年轻代的垃圾回收算法使用的是复制算法，复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片。

- 优缺点

1. 无内存碎片
2. 耗内存空间
3. 若存活对象占有率多，需要复制的就多，这种情况会比较耗时

3. 标记清除（Mark-Sweep）

- **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**
- 原理

> 算法分成标记和清除两个阶段，**先标记出要回收的对象，然后统一回收这些对象**
>
> 
>

- 优缺点
  1. 节约内存空间
  2. 两次扫描，耗时严重
  3. 产生内存碎片

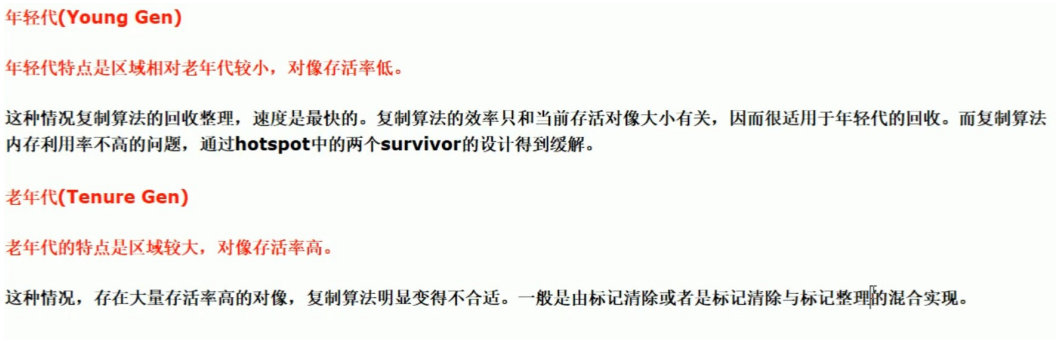
4. 标记整理（Mark-Compact）

> - **老年代一般是由标记清除或者标记清除与标记整理的混合实现**
> - 原理
>
>   1. 标记（Mark）
>
>   与标记清除一样
>
>   2. 压缩（Compact）
>
>   再次扫描，并往一端滑动存活对象
>
>   3. 清除标记对象
>
> - 优缺点
>
>   1. 没有内存碎片
>   2. 需要移动对象的成本，即耗时
>
> **标记-清除-压缩**
>
> 原理：
>
> - Mark-Sweep 和 Mark-Compact的结合
> - 和Mark-Sweep一致，当进行多次GC后才Compact
>
> 优点：
>
> 减少移动对象的成本

- 小总结

```
内存效率：复制算法>标记清除算法>标记整理算法（此处的效率只是简单的对比时间复杂度，实际情况不一定如此）。
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
```

> 问题：哪张算法最好？
>
> 无，没有最好的算法，只有最合适的算法 ========>>> 分代手机算法



- 面试题

> 1. 执行一个空的main方法，后台有几个线程？
>
>> 2个线程，一个main线程，一个GC线程
>
> 2. 谈谈对 public static void main(String[] args) 的理解？
>
>> public，static，void的解释
>>
>> 程序的入口。
>
> 3. 对 System.gc() 的理解？
>
>> 显示开启GC，但不是立即执行；实际开发中禁用！
>
> 4. JVM内存模型及分区，需要详细到每个分区放什么
> 5. 堆里面的分区，Eden，survivor from to，老年代，各自的特点
> 6. GC的三只手机方法，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
> 7. Minor GC 和Full GC 分别在什么时候发生

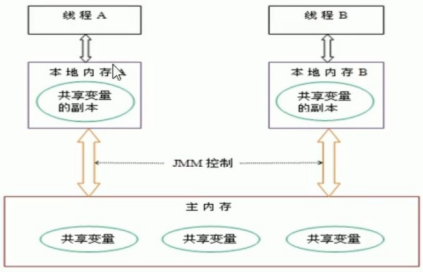
4.JMM（java内存模型）

4.1 volatile是java虚拟机提供的轻量级的同步机制

- 保证可见性
- 不保证原子性
- 禁止指令重排序

4.2 JMM 你谈谈

> JMM（Java内存模型 Java Memory Model，简称JMM）本身是一种抽象的概念，并不真实存在，它描述的是一组规则或规范，通过这组规范定义了程序中各个变量（包括实例字段，静态字段和构成数组对象的元素）的访问方式。
>
> JMM关于同步的规定：
>
> 1. 线程解锁前，必须把共享变量的值刷新回主内存
> 2. 线程加锁前，必须读取主内存的最新值到自己的工作内存
> 3. 加锁解锁是同一把锁

> 由于JVM运行程序的实体是线程，而每个线程创建时JVM都会为其创建一个工作内存(有些地方称为栈空间)，工作内存是每个线程的私有数据区域，而Java内存模型中规定所有变量都存储在主内存，主内存是共享内存区域，所有线程都可以访问，但线程对变量的操作(读取赋值等)必须在工作内存中进行，首先要将变量从主内存拷贝到的线程自己的工作内存空间，然后对变量进行操作，操作完成后再将变量写回主内存，不能直接操作主内存中的变量，各个线程中的工作内存中存储着主内存中的变量副本拷贝，因此不同的线程间无法访问对方的工作内存，线程间的通信(传值)必须通过主内存来完成，其简要访问过程如下图:
>
> 
>

4.2.1 可见性

4.2.2 原子性

4.2.3 有序性

4.2.4 VolatileDemo代码演示可见性+原子性代码

```
public class Jmm {
    public static void main(String[] args) {
        Test t = new Test();
        new Thread(() -> {
            System.out.println("come in A thread...");
            try {
                TimeUnit.SECONDS.sleep(3);
                t.change();
                System.out.println(Thread.currentThread().getName() + "修改了
flag" + t.flag);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        },"A").start();

        while (t.flag == 10) {
            // 需要有一种机制来告诉main线程，flag已经被修改了，跳出while
        }
        System.out.println(Thread.currentThread().getName() + " is over...");
    }
}
class Test{
//    Integer flag = 10;
    // volatile关键字的使用
    volatile Integer flag = 10;
    public void change() {
        this.flag = 20;
    }
}
```