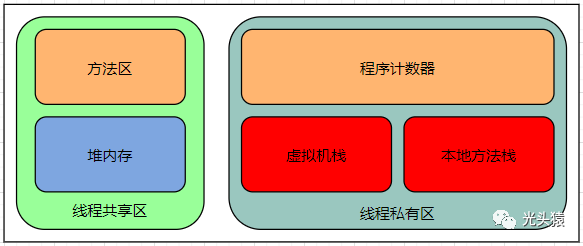
# 1、简述一下JVM的内存结构？

JVM在执行Java程序时，会把它管理的内存划分为若干个的区域，每个区域都有自己的用途。如下图所示，可以分为两大部分，线程私有区和共享区。



**01 程序计数器**

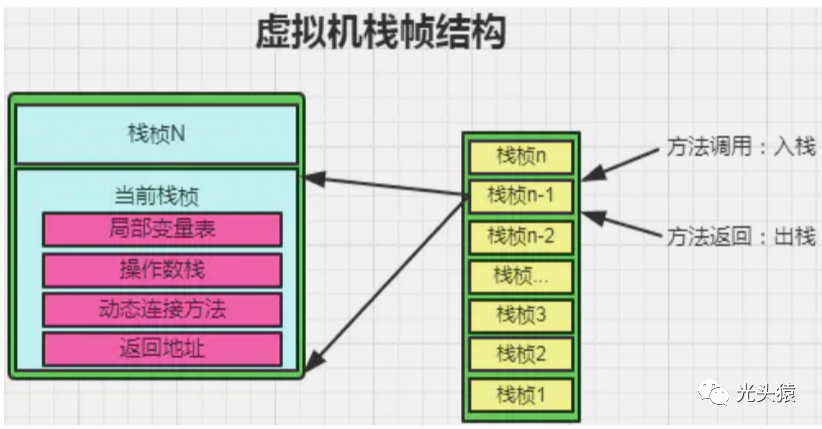
作用：是一块较小的内存空间，可以理解为是当前线程所执行程序的字节码文件的行号指示器，存储的是当前线程所执行的**行号**

特点：线程私有 ，唯一一个不会出现内存溢出的内存空间

**02 虚拟机栈**

作用：管理JAVA方法执行的内存区域。每个方法执行时都会创建一个桢栈来存储方法中变量的变量表、操作数栈、动态链接方法、返回值、返回地址等信息。栈的大小决定了方法调用的可达深度（递归多少层次，或嵌套调用多少层其他方法，-Xss参数可以设置虚拟机栈大小）

如下所示：



特点：

1、线程私有

2、局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）以及对象引用（reference 类型）

3、栈太小方法调用过深，将抛出StackOverflowError异常

测试代码：

public class StackDemo02 {  
 *// 记录调用了多少次出现了栈内存溢出*    private static int count = 0 ;  
 *// 入口方法*    public static void main(String[] args) {  
 try { show() ; }catch (Throwable e) { e.printStackTrace();        }  
        System.out.println("show方法被调用了:" + count + "次");  
    }     *// 测试方法* public static void show() { count++ ; System.out.println("show方法执行了....."); show();    }  
}

配置虚拟机参数-Xss可以指定栈内存大小；例如：-Xss180k

栈内存的默认值问题：

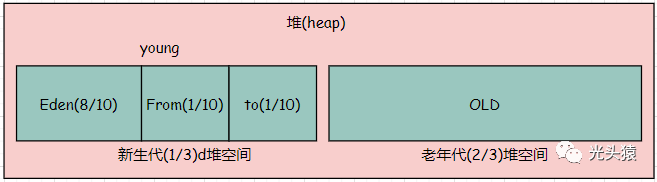
The default value depends on the platform: \* Linux/x64 (64-bit): 1024 KB \* macOS (64-bit): 1024 KB \* Oracle Solaris/x64 (64-bit): 1024 KB \* Windows: The default value depends on virtual memory

**03 本地方法栈**

与虚拟机栈作用相似。但它不是为Java方法服务的，而是本地方法（C语言）。由于规范对这块没有强制要求，不同虚拟机实现方法不同。

**04 堆内存**

作用：是Java内存区域中一块用来存放对象实例的区域，新创建的对象，数组都使用堆内存；【从Java7开始，常量池也会使用堆内存】



Java 堆从GC的角度还可以细分为: 新生代( Eden区 、From Survivor区和 To Survivor区 )和老年代。

特点：

1、被线程共享，因此需要考虑线程安全问题

2、会产生内存溢出问题

测试代码：

public class HeapDemo01 {    public static void main(String[] args) { *// 定义一个变量*        int count = 0 ;  
 *// 创建一个ArrayList对象*        ArrayList arrayList = new ArrayList() ;        try { while(true) { arrayList.add(new Object()) ; count++ ;            } }catch (Throwable a) { a.printStackTrace(); *// 输出程序执行的次数* System.out.println("总共执行了:" + count + "次");        }    }}

虚拟机参数

-Xms:设置最小堆内存大小(不能小于1024K)

-Xmx:设置最大堆内存大小(不能小于1024K)

可以通过jmap工具进行查看堆内存分配情况

例如：-Xms1024K -Xmx2048K

注意：

图片

**05 方法区**

作用：它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

特点：

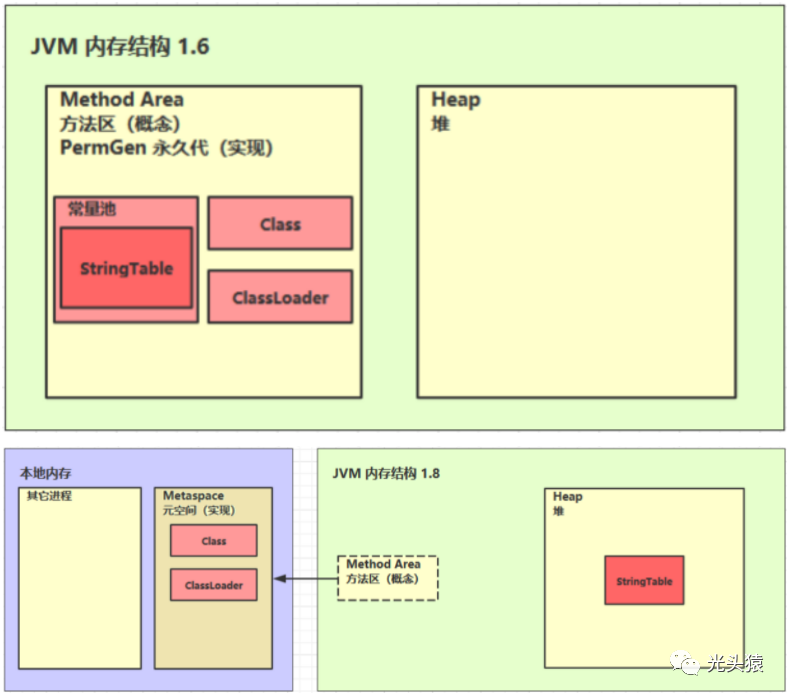
1、方法区是一块线程共享的内存区域

2、方法区的大小决定了系统可以保存多少个类，如果系统定义了太多的类，导致方法区溢出，虚拟机同样会抛出内存溢出的错误

3、jdk1.6和jdk1.7方法区也常常被称之为永久区(永久代),大小一般都是几百兆；

4、jdk1.8已经将方法区取消，替代的是元数据区(元空间),如果不指定大小，默认情况下，虚拟机会耗尽可用系统内存

5、jdk7以后就将方法区中的常量池移动至**堆内存**



变化的原因：

1、提高内存的回收效率(方法区内存的回收效率远远低于堆内存，因为方法区中存储的都是类信息，静态变量...这些信息不能被轻易回收)

2、字符串常量池在方法区，那么很容易产生内存溢出(因为方法区的垃圾回收效率比较低);

测试代码：

*/\*\* jdk1.8的元数据区可以使用参数-XX:MaxMetaspaceSzie设定大小 \* 演示元空间内存溢出 \* -XX:-UseCompressedClassPointers -XX:MaxMetaspaceSize=10m UseCompressedClassPointers使用指针压缩，如果不使用这个参数可能会出现: Compressed class space内存溢出 \*/*public class MaxMetaspaceDemo extends ClassLoader { *// 当前这个类就是一个类加载器* public static void main(String[] args) { *// 定义变量，记录程序产生类的个数* int j = 0; try { MaxMetaspaceDemo test = new MaxMetaspaceDemo(); for (int i = 0; i < 10000; i++, j++) { *// 字节码写入器* ClassWriter cw = new ClassWriter(0); *// 定义一个类版本为Opcodes.V1\_1，它的访问域为public，名称为Class{i},父类为java.lang.Object，不实现任何接口* cw.visit(Opcodes.V1\_1, Opcodes.ACC\_PUBLIC, "Class" + i, null, "java/lang/Object", null); byte[] code = cw.toByteArray(); *// 加载该类* test.defineClass("Class" + i, code, 0, code.length); } } finally { System.out.println(j); } }}

# 2、堆和栈的区别？

① 功能不同：栈内存用来存储局部变量和方法调用，而堆内存用来存储Java中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中。

② 共享性不同：栈内存是线程私有的。堆内存是所有线程共有的。

③ 异常错误不同：如果栈内存或者堆内存不足都会抛出异常。栈空间不足：java.lang.StackOverFlowError。堆空间不足：java.lang.OutOfMemoryError。

④ 空间大小：栈的空间大小远远小于堆的。

# 3、怎么获取Java程序使用的内存？堆使用的百分比？

可以通过java.lang.Runtime类中与内存相关方法来获取剩余的内存，总内存及最大堆内存。通过这些方法你也可以获取到堆使用的百分比及堆内存的剩余空间。

1、Runtime.freeMemory() 方法返回剩余空间的字节数

2、Runtime.totalMemory()方法总内存的字节数

# 4、栈帧都有哪些数据？

JVM的运行是基于栈的，和C语言的栈类似，它的大多数数据都是在堆里面的，只有少部分运行时的数据存在于栈上。在JVM中，每个线程栈里面的元素，就叫栈帧。

栈帧包含：局部变量表、操作数栈、动态连接、返回地址等。

# 5、如何启动系统的时候设置JVM的启动参数？

其实都很简单，比如说采用"java -jar"的方式启动一个jar包里面的系统，那么就可以采用类似下面的格式：

# 6、如何判断一个对象是否为垃圾？

① 引用计数法：堆中每个对象实例都有一个引用计数。当一个对象被创建时，且将该对象实例分配给一个变量，该变量计数设置为1。当任何其它变量被赋值为这个对象的引用时，计数加1（a = b,则b引用的对象实例的计数器+1），但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减1。任何引用计数器为0的对象实例可以被当作垃圾收集。

特点：简单、无法解决循环引用问题

学生类：

public class Student { *// 定义成员变量*    public Object instance ;}

编写测试类：

*/\* jvm参数：-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -verbose:gc -XX:+PrintGCDetails：打印gc日志信息 -XX:+PrintGCTimeStamps: 打印gc日志的时间戳\*/*public class ReferenceCountGcDemo {    public static void main(String[] args) { *// 创建Student对象* Student a = new Student() ;        Student b = new Student() ; *// 进行循环引用* a.instance = b ;        b.instance = a ; *// 将a对象和b对象设置为null* a = null ;        b = null ; *// 调用System.gc进行垃圾回收*        System.gc();                    *// 如果没有触发垃圾回收说明Hotspot的jvm使用的就是引用计数法来判断对象是否为垃圾*    }}

控制台输出gc日志:

0.076: [GC (System.gc()) [PSYoungGen: 7802K->856K(151552K)] 7802K->864K(498688K), 0.0008493 secs] [Times: user=0.17 sys=0.02, real=0.00 secs] 0.077: [Full GC (System.gc()) [PSYoungGen: 856K->0K(151552K)] [ParOldGen: 8K->620K(347136K)] 864K->620K(498688K), [Metaspace: 3356K->3356K(1056768K)], 0.0044768 secs] [Times: user=0.00 sys=0.00, real=0.00 secs] Heap PSYoungGen total 151552K, used 3901K [0x0000000716c00000, 0x0000000721500000, 0x00000007c0000000) eden space 130048K, 3% used [0x0000000716c00000,0x0000000716fcf748,0x000000071eb00000) from space 21504K, 0% used [0x000000071eb00000,0x000000071eb00000,0x0000000720000000) to space 21504K, 0% used [0x0000000720000000,0x0000000720000000,0x0000000721500000) ParOldGen total 347136K, used 620K [0x00000005c4400000, 0x00000005d9700000, 0x0000000716c00000) object space 347136K, 0% used [0x00000005c4400000,0x00000005c449b318,0x00000005d9700000) Metaspace used 3365K, capacity 4496K, committed 4864K, reserved 1056768K class space used 370K, capacity 388K, committed 512K, reserved 1048576K

① 0.076: 代表gc发生的时间，从jvm启动以来经过的秒数② [GC和[Full Gc: 说明这次垃圾收集器的停顿类型，而不是用来区分新生代GC还是老年代GC的。如果有"Full"，说明此次GC发生了stop-the-world。System.gc()是说明显示的调用了 System.gc方法进行垃圾回收③ [PSYoungGen：表示GC发生的区域, 不同的垃圾收集器展示的区域名称不一样，PSYoungGen表示的是新生代，这里默认使用的是Parallel Scavenge收集器 (-XX:+UseSerialGC)④ 7802K->856K(151552K)：GC前该区域已使用容量 -> GC后该区域已使用容量(该区域的总容量)⑤ 7802K->864K(498688K)：GC前Java堆已使用容量 -> GC后Java堆已使用容量(Java堆总容量)⑥ 0.0008493 secs：该区域GC所占用的时间⑦ [Times: user=0.17 sys=0.02, real=0.00 secs]: 分别表示用户态消耗的CPU时间、内核态消耗的CPU时间和操作从开始到结束所经过的墙钟时间(墙钟时间包括非运算的等待耗时)。多线程操作会叠加这些CPU时间，所以user、sys时间超过real时间是完全正常的。

② 可达性分析算法 ：可达性分析算法又叫做跟搜索法，就是通过一系列的称之为"GC Roots"的对象作为起始点，从这些节点开始向下搜索，搜索走过的路径被称为（Reference Chain)，当一个对象到GC Roots没有任何引用链相连时（即从GC Roots节点到该节点不可达），则证明该对象是不可用的。(类似于葡萄串)；

# 7、可达性算法中，哪些对象可作为GC Roots对象？

可以作为GC ROOTS对象的情况：

1、虚拟机栈中引用的对象

2、方法区静态成员引用的对象

3、方法区常量引用对象

4、本地方法栈引用的对象

# 8、Java中都有哪些引用类型？

**01 强引用**

Java中默认声明的就是强引用，比如：

Object obj = new Object(); *//只要obj还指向Object对象，Object对象就不会被回收*obj = null; *//手动置null*

只要强引用存在，垃圾回收器将永远不会回收被引用的对象，哪怕内存不足时，JVM也会直接抛出OutOfMemoryError，不会去回收。如果想中断强引用与对象之间的联系，可以显示的将强引用赋值为null，这样一来，JVM就可以适时的回收对象了

示例：

*/\*\* \* JVM参数：-verbose:gc -XX:+PrintGCDetails -Xms10M -Xmx10M -Xmn5M \*/*public class StrongReferenceDemo01 {    private static List<Object> list = new ArrayList<Object>() ;    public static void main(String[] args) { *// 创建对象* for(int x = 0 ; x < 10 ; x++) { byte[] buff = new byte[1024 \* 1024 \* 1]; list.add(buff);        }    }}

**02 软引用**

软引用是用来描述一些非必需但仍有用的对象。在内存足够的时候，软引用对象不会被回收，只有在内存不足时，系统则会回收软引用对象，如果回收了软引用对象之后仍然没有足够的内存，才会抛出内存溢出异常。这种特性常常被用来实现缓存技术，比如网页缓存，图片缓存等。

在 JDK1.2 之后，用java.lang.ref.**SoftReference**类来表示软引用。

示例代码：

*/\*\* \* JVM参数：-verbose:gc -XX:+PrintGCDetails -Xms10M -Xmx10M -Xmn5M \*/*public class SoftReferenceDemo01 {    private static List<Object> list = new ArrayList<>();    public static void main(String[] args) { *// 创建数组对象* for(int x = 0 ; x < 10 ; x++) { SoftReference<byte[]> softReference = new SoftReference<byte[]>(new byte[1024 \* 1024 \* 1]) ; list.add(softReference) ;        }        System.gc();  *// 主动通知垃圾回收器进行垃圾回收* for(int i=0; i < list.size(); i++){ Object obj = ((SoftReference) list.get(i)).get(); System.out.println(obj); }    }}

我们发现无论循环创建多少个软引用对象，打印结果总是有一些为null，这里就说明了在内存不足的情况下，软引用将会被自动回收。

**03 弱引用**

弱引用的引用强度比软引用要更弱一些，无论内存是否足够，只要 JVM 开始进行垃圾回收，那些被弱引用关联的对象都会被回收。在 JDK1.2之后,用java.lang.ref.WeakReference来表示弱引用。

示例代码：

*/\*\* \* JVM参数：-verbose:gc -XX:+PrintGCDetails -Xms10M -Xmx10M -Xmn5M \*/*public class WeakReferenceDemo01 {    private static List<Object> list = new ArrayList<>();    public static void main(String[] args) { *// 创建数组对象* for(int x = 0 ; x < 10 ; x++) { WeakReference<byte[]> weakReference = new WeakReference<byte[]>(new byte[1024 \* 1024 \* 1]) ; list.add(weakReference) ;        }        System.gc();  *// 主动通知垃圾回收器进行垃圾回收* for(int i=0; i < list.size(); i++){ Object obj = ((WeakReference) list.get(i)).get(); System.out.println(obj); }    }}

**04 虚引用**

虚引用是最弱的一种引用关系，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，它随时可能会被回收，在 JDK1.2 之后，用 PhantomReference 类来表示，通过查看这个类的源码，发现它只有一个构造函数和一个 get() 方法，而且它的 get() 方法仅仅是返回一个null，也就是说将永远无法通过虚引用来获取对象，虚引用必须要和 ReferenceQueue 引用队列一起使用。

public class PhantomReference<T> extends Reference<T> { public T get() { return null;    } public PhantomReference(T referent, ReferenceQueue<? super T> q) { super(referent, q);    }}

特点：

1、每次垃圾回收时都会被回收，主要用于监测对象是否已经从内存中删除

2、虚引用必须和引用队列关联使用, 当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会把这个虚引用加入到与之关联的引用队列中

3、程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。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

示例代码：

public class PhantomReferenceDemo {    public static void main(String[] args) throws InterruptedException { *// 创建一个引用队列*        ReferenceQueue<Object> referenceQueue = new ReferenceQueue<Object>();         *// 创建一个虚引用，指向一个Object对象*        PhantomReference<Object> phantomReference = new PhantomReference<Object>(new Object(), referenceQueue);       *// 主动通知垃圾回收器进行垃圾回收*        System.gc();        *// 从引用队列中获取元素, 该方法是阻塞方法*        System.out.println(referenceQueue.remove());  }}

# 9、常见的垃圾回收算法都有哪些？

**01 标记清除**

执行过程：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

优点：速度比较快

缺点：会产生内存碎片，碎片过多，仍会使得连续空间少

**02 标记整理**

执行过程：首先标记出所有需要回收的对象，在标记完成后统一进行整理，整理是指存活对象向一端移动来减少内存碎片，相对效率较低

优点：无内存碎片

缺点：效率较低

**03 复制算法**

执行过程：开辟两份大小相等空间，一份空间始终空着，垃圾回收时，将存活对象拷贝进入空闲空间；

优点：无内存碎片

缺点：占用空间多

注意：如果有很多对象的存活率较高，这时我们采用复制算法，那么效率就比较低；

**04 分代回收**

概述：根据对象存活周期的不同，将对象划分为几块，比如Java的堆内存，分为新生代和老年代，然后根据各个年代的特点采用最合适的算法；

新生代对象的存活的时间都比较短，因此使用的是【复制算法】；而老年代对象存活的时间比较长那么采用的就是【标记清除】或者【标记整理】；

# 10、简述Java垃圾回收机制？有什么办法主动通知虚拟机进行垃圾回收？

在Java中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。在JVM中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫描那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。

# 11、对象在内存中是如何进行分配的？

**01 对象优先在伊甸园分配**

对象优先在『伊甸园』分配，当『伊甸园』没有足够的空间时，触发 'Minor GC'(小范围的GC)

情况一：伊甸园的内存空间足够，不会发生'Minor GC'

情况二：伊甸园的空间不够了

垃圾回收线程启动，进行垃圾回收，此时会触发"stop the world"(停止所有用户线程)，

Eden区中所有存活的对象都会被复制到“To”，而在“From”区中，仍存活的对象会根据他们的年龄值来决定去向。年龄最多到一定值(最大值是15，对象在Survivor区中每熬过一次Minor GC，年龄就会增加1岁)(年龄阈值，可以通过-XX:MaxTenuringThreshold来设置)的对象会被移动到年老代中，没有达到阈值的对象会被复制到“To”区域。

"From"和"To"会交换他们的角色,下一次垃圾回收的时候也是从Eden将存活的对象复制到TO区

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，会将所有对象移动到年老代中。

案例演示：

jvm参数设置：

-XX:+UseSerialGC -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:./gc.log -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8-XX:+UseSerialGC 是指使用 Serial + SerialOld 回收器组合-XX:+PrintGCDetails -verbose:gc 是指打印 GC 详细信息-XX:+PrintGCTimeStamps 打印gc日志的时间戳-Xloggc:./gc.log 将gc日志输出到一个日志文件中-Xms20M -Xmx20M -Xmn10M 是指分配给JVM的最小，最大以及新生代内存-XX:SurvivorRatio=8 是指『伊甸园』与『幸存区 From』和『幸存区 To』比例为 8:1:1

定义内存大小变量

private static final int \_512KB = 512 \* 1024;private static final int \_1MB = 1024 \* 1024;private static final int \_4MB = 4 \* 1024 \* 1024;private static final int \_7MB = 7 \* 1024 \* 1024;private static final int \_8MB = 8 \* 1024 \* 1024;

案例1：没有创建数组对象，看参数运行情况

案例2：创建一个4M的数组，查看内存分配情况

*// 创建一个4M大小的数组*byte[] bytes = new byte[\_4MB] ;

Heap def new generation total 9216K, used 8192K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000) eden space 8192K, 100% used [0x00000000fec00000, 0x00000000ff400000, 0x00000000ff400000) *// 在伊甸园中创建对象* from space 1024K, 0% used [0x00000000ff400000, 0x00000000ff400000, 0x00000000ff500000) to space 1024K, 0% used [0x00000000ff500000, 0x00000000ff500000, 0x00000000ff600000) tenured generation total 10240K, used 0K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000) the space 10240K, 0% used [0x00000000ff600000, 0x00000000ff600000, 0x00000000ff600200, 0x0000000100000000) Metaspace used 3444K, capacity 4496K, committed 4864K, reserved 1056768K class space used 376K, capacity 388K, committed 512K, reserved 1048576K

没有触发GC操作，对象直接在Eden分配；

案例3：创建一个7M的数组，查看内存分配情况

*// 创建一个7M大小的数组*byte[] bytes1 = new byte[\_7MB] ;

-- 触发垃圾回收[GC (Allocation Failure) [DefNew: 2004K->647K(9216K), 0.0023439 secs] 2004K->647K(19456K), 0.0024142 secs] [Times: user=0.00 sys=0.00, real=0.00 secs] Heap def new generation total 9216K, used 7897K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000) eden space 8192K, 88% used [0x00000000fec00000, 0x00000000ff314930, 0x00000000ff400000) from space 1024K, 63% used [0x00000000ff500000, 0x00000000ff5a1e58, 0x00000000ff600000) to space 1024K, 0% used [0x00000000ff400000, 0x00000000ff400000, 0x00000000ff500000) tenured generation total 10240K, used 0K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000) the space 10240K, 0% used [0x00000000ff600000, 0x00000000ff600000, 0x00000000ff600200, 0x0000000100000000) Metaspace used 3446K, capacity 4496K, committed 4864K, reserved 1056768K class space used 376K, capacity 388K, committed 512K, reserved 1048576K

由于程序在启动的时候jdk内部还会存在一些对象的创建，因此当我们分配了一个7M的内存空间，eden内存不足，因此发生了一次Minor GC！并且将存活下的对象最终存储到from区中。

案例4：在案例3的基础上，在分配一个512KB的数组内存空间

byte[] bytes1 = new byte[\_7MB] ;byte[] bytes2 = new byte[\_512KB] ;

[GC (Allocation Failure) [DefNew: 2005K->623K(9216K), 0.0015235 secs] 2005K->623K(19456K), 0.0015799 secs] [Times: user=0.00 sys=0.00, real=0.00 secs] Heap def new generation total 9216K, used 8713K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000) eden space 8192K, 98% used [0x00000000fec00000, 0x00000000ff3e6820, 0x00000000ff400000) from space 1024K, 60% used [0x00000000ff500000, 0x00000000ff59bdb8, 0x00000000ff600000) to space 1024K, 0% used [0x00000000ff400000, 0x00000000ff400000, 0x00000000ff500000) tenured generation total 10240K, used 0K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000) the space 10240K, 0% used [0x00000000ff600000, 0x00000000ff600000, 0x00000000ff600200, 0x0000000100000000) Metaspace used 3444K, capacity 4496K, committed 4864K, reserved 1056768K class space used 376K, capacity 388K, committed 512K, reserved 1048576K

触发一次GC操作！并且将存活下的对象最终存储到from区中，第二次分配\_512KB大小的内存空间的时候，直接在伊甸园分配即可。

案例5：在4的基础上在分配一个512KB的数组内存空间

byte[] bytes1 = new byte[\_7MB] ;byte[] bytes2 = new byte[\_512KB] ;byte[] bytes3 = new byte[\_512KB] ;

[GC (Allocation Failure) [DefNew: 2004K->620K(9216K), 0.0018706 secs] 2004K->620K(19456K), 0.0019275 secs] [Times: user=0.00 sys=0.00, real=0.00 secs] [GC (Allocation Failure) [DefNew: 8628K->539K(9216K), 0.0063389 secs] 8628K->8323K(19456K), 0.0063773 secs] [Times: user=0.00 sys=0.01, real=0.01 secs] Heap def new generation total 9216K, used 1133K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000) eden space 8192K, 7% used [0x00000000fec00000, 0x00000000fec94930, 0x00000000ff400000) from space 1024K, 52% used [0x00000000ff400000, 0x00000000ff486de0, 0x00000000ff500000) to space 1024K, 0% used [0x00000000ff500000, 0x00000000ff500000, 0x00000000ff600000) tenured generation total 10240K, used 7784K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000) the space 10240K, 76% used [0x00000000ff600000, 0x00000000ffd9a040, 0x00000000ffd9a200, 0x0000000100000000) Metaspace used 3443K, capacity 4496K, committed 4864K, reserved 1056768K class space used 376K, capacity 388K, committed 512K, reserved 1048576K

触发了2次垃圾回收！并且将from区中存活的对象存储到老年代！

**02 大对象直接晋升至老年代**

当对象太大，伊甸园包括幸存区都存放不下时，这时候老年代的连续空间足够，此对象会直接晋升至老年代，不会发生 GC

结果

案例演示：

案例1：直接分配一个8M的内存空间

byte[] bytes1 = new byte[\_8MB] ;

伊甸园总大小只有 8 MB，但新分配的对象大小已经是 8MB，而幸存区都仅有 1MB，也无法容纳这个对象

Heap def new generation total 9216K, used 2169K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000) eden space 8192K, 26% used [0x00000000fec00000, 0x00000000fee1e560, 0x00000000ff400000) from space 1024K, 0% used [0x00000000ff400000, 0x00000000ff400000, 0x00000000ff500000) to space 1024K, 0% used [0x00000000ff500000, 0x00000000ff500000, 0x00000000ff600000) tenured generation total 10240K, used 8192K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000) the space 10240K, 80% used [0x00000000ff600000, 0x00000000ffe00010, 0x00000000ffe00200, 0x0000000100000000) Metaspace used 3443K, capacity 4496K, committed 4864K, reserved 1056768K class space used 376K, capacity 388K, committed 512K, reserved 1048576K

可以看到结果并没有发生 GC，大对象直接被放入了老年代「tenured generation   total 10240K, used 8192K」

案例2：老年代连续空间不足，触发 Full GC

byte[] bytes1 = new byte[\_8MB] ;byte[] bytes2 = new byte[\_8MB] ;

第一个 8MB 直接进入老年代，第二个 8MB 对象在分配时发现老年代空间不足，只好尝试先进行一次 Minor GC，结果发现新生代没有连续空间，只好触发一次 Full GC，最后发现老年代也没有连续空间，这时出现 OutOfMemoryError

[GC (Allocation Failure) [DefNew: 2004K->647K(9216K), 0.0022693 secs][Tenured: 8192K->8838K(10240K), 0.0452151 secs] 10197K->8838K(19456K), [Metaspace: 3438K->3438K(1056768K)], 0.0504669 secs] [Times: user=0.00 sys=0.00, real=0.05 secs] [Full GC (Allocation Failure) [TenuredException in thread "main" : 8838K->8820K(10240K), 0.0027463 secs] 8838K->8820K(19456K), [Metaspace: 3438K->3438K(1056768K)], 0.0027877 secs] [Times: user=0.00 sys=0.00, real=0.00 secs] java.lang.OutOfMemoryError: Java heap space at com.itheima.jvm.gc.ObjectMemoryDemo.main(ObjectMemoryDemo.java:14)Heap def new generation total 9216K, used 246K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000) eden space 8192K, 3% used [0x00000000fec00000, 0x00000000fec3d890, 0x00000000ff400000) from space 1024K, 0% used [0x00000000ff500000, 0x00000000ff500000, 0x00000000ff600000) to space 1024K, 0% used [0x00000000ff400000, 0x00000000ff400000, 0x00000000ff500000) tenured generation total 10240K, used 8820K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000) the space 10240K, 86% used [0x00000000ff600000, 0x00000000ffe9d220, 0x00000000ffe9d400, 0x0000000100000000) Metaspace used 3470K, capacity 4496K, committed 4864K, reserved 1056768K class space used 379K, capacity 388K, committed 512K, reserved 1048576K

# 12、对象是怎么从年轻代进入老年代的？

存在3种情况：

1、如果对象够老，会通过提升（Promotion）进入老年代，这一般是根据对象的年龄进行判断的。

2、动态对象年龄判定。有的垃圾回收算法，比如G1，并不要求age必须达到15才能晋升到老年代，它会使用一些动态的计算方法。

3、超出某个大小的对象将直接在老年代分配。不过这个值默认为0，意思是全部首选Eden区进行分配。

# 13、MinorGC，MajorGC、FullGC都什么时候发生？

MinorGC在年轻代空间不足的时候发生，MajorGC指的是老年代的GC，出现MajorGC一般经常伴有MinorGC。

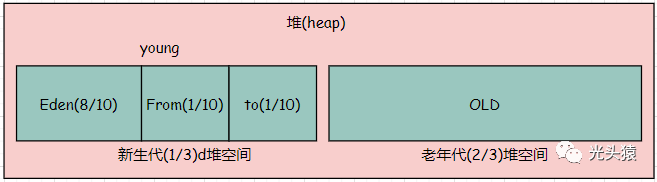
FullGC有三种情况：

1、当老年代无法再分配内存的时候

2、元空间不足的时候

3、显式调用System.gc的时候

# 14、简单描述一下（分代）垃圾回收的过程？



分代回收器有两个分区：老生代和新生代，新生代默认的空间占比总空间的 1/3，老生代的默认占比是2/3。

新生代使用的是复制算法，新生代里有3个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1，它的执行流程如下：

当年轻代中的Eden区分配满的时候，就会触发年轻代的GC（Minor GC）。具体过程如下：

1、在Eden区执行了第一次GC之后，存活的对象会被移动到其中一个Survivor分区（以下简称to）

2、From区中的对象根据对象的年龄值决定去向，达到阈值15移动到老年代，没有达到复制到to区域(复制算法)

3、在把Eden和From区中的对象清空掉

# 15、JVM的永久代中会发生垃圾回收么？

永久代会触发垃圾回收的，如果永久代满了或者是超过了临界值，会触发完全垃圾回收（Full GC）。

注：Java 8 中已经移除了永久代，新加了一个叫做元数据区(Metaspace)的内存区。

# 16、常见的垃圾收集器都有哪些？

常见的垃圾收集器如下所示：

不同的垃圾收集器，作用的堆内存空间是不一样的；上面的  serial ， parnew ， Paraller Scavenge 是新生代的垃圾回收器；CMS ， Serial Old ， Paralle Old是老年代的垃圾收集器 ， G1垃圾收集器可以作用于新生代和老年代；连线表示垃圾收集器可以搭配使用；

**01 Serial**

特点：

1、Serial是一个单线程的垃圾收集器

2、"Stop The World"，它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。在用户不可见的情况下把用户正常工作的线程全部停掉。

应用场景：

1、多用于桌面应用，Client端的垃圾回收器

2、桌面应用内存小，进行垃圾回收的时间比较短，只要不频繁发生停顿就可以接受

Serial Old收集器是Serial的老年代版本和Serial一样是单线程，使用的算法是"标记-整理"

**02 ParNew**

概述: ParNew 收集器其实就是 Serial 收集器的多线程版本

特点：

1、会触发stop the world

2、多线程方式进行垃圾回收

应用场景：它却是许多运行在 Server 模式下的虚拟机中首选的新生代收集器

注意：如果是单核cpu即使使用该垃圾回收器也无法提高执行效率

**03 Parallel Scavenge**

概述：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

特点：由于与吞吐量关系密切，Parallel Scavenge 收集器也经常称为“吞吐量优先”收集器。所谓吞吐量就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间），虚拟机总共运行了 100 分钟，其中垃圾收集花掉 1 分钟，那吞吐量就是 99%

应用场景：高吞吐量则可以高效率地利用 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。

Parallel old收集器Parallel Scavenge收集器的老年代版本，使用多线程+标记整理算法

**04 CMS**

概述：CMS  (Concurrent Mark Sweep）收集器是-种以获取最短回收停顿时间为目标的收集器。

特点：

1、CMS 收集器是基于“标记-清除”算法实现的

2、目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上,这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。

步骤流程：

1、初始标记（CMS initial mark)：标记一下 GC Roots 能直接关联到的对象，速度很快（stop the world）

2、并发标记（CMS concurrent mark）：对初始标记标记过的对象，进行trace（进行追踪，得到所有关联的对象，进行标记）

3、重新标记（CMS remark) ：为了修正并发标记期间因用户程序导致标记产生变动的标记记录（stop the world）

4、并发清除（CMS concurrent sweep)

缺点：会产生垃圾碎片

**05 G1**

概述：G1是一个分代的，并行与并发的"**标记-整理**"垃圾回收器。它的设计目标是为了适应现在不断扩大的内存和不断增加的处理器数量，进一步降低暂停时间（pause time），同时兼顾良好的吞吐量。

相比于CMS：

1、G1垃圾回收器使用的是"标记-整理"，因此其回收得到的空间是连续的。

2、G1回收器的内存与CMS回收器要求的内存模型有极大的不同。G1将内存划分一个个固定大小的region，每个region可以是年轻代、老年代的一个。内存的回收是以region作为基本单位的；

# 17、你都用过G1垃圾回收器的哪几个重要参数？

1、-XX:MaxGCPauseMillis

暂停时间，默认值200ms。这是一个软性目标，G1会尽量达成，如果达不成，会逐渐做自我调整。

2、-XX:G1HeapRegionSize

Region大小，若未指定则默认最多生成2048块，每块的大小需要为2的幂次方，如1,2,4,8,16,32，最大值为32M。

3、-XX:G1NewSizePercent

新生代比例有两个数值指定，下限：

-XX:G1NewSizePercent，默认值5%

4、-XX:G1MaxNewSizePercent

新生代比例有两个数值指定，上限：

-XX:G1MaxNewSizePercent，默认值60%。

# 18、串行（serial）收集器和吞吐量（throughput）收集器的应用场景？

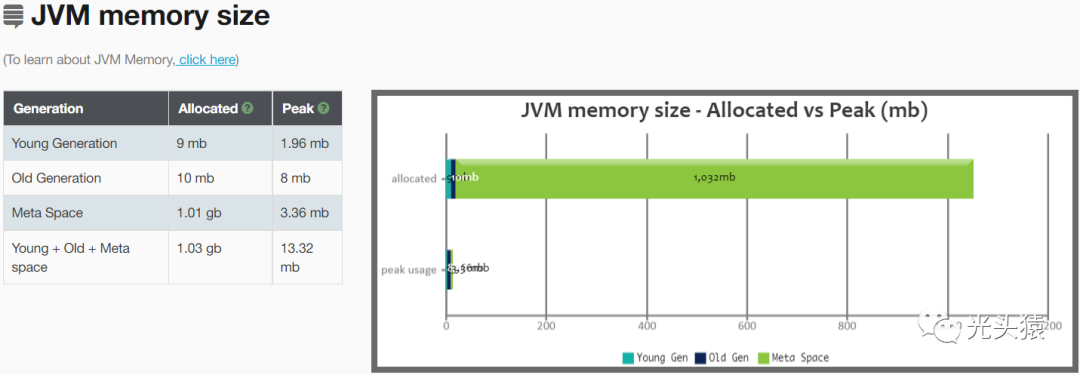
吞吐量收集器使用并行版本的新生代垃圾收集器，它用于中等规模和大规模数据的应用程序。而串行收集器对大多数的小应用（在现代处理器上需要大概 100M 左右的内存）就足够了。

# 19、生产上如何配置垃圾收集器的？

1、首先是内存大小问题，基本上每一个内存区域我都会设置一个上限，来避免溢出问题，比如元空间。通常，堆空间我会设置成操作系统的2/3（这是想给其他进程和操作系统预留一些时间），超过8GB的堆优先选用G1。

2、接下来，我会对JVM进行初步优化。比如根据老年代的对象提升速度，来调整年轻代和老年代之间的比例。

3、再接下来，就是专项优化，主要判断的依据就是系统容量、访问延迟、吞吐量等。我们的服务是高并发的，所以对STW的时间非常敏感。我会通过记录详细的GC日志，来找到这个瓶颈点，借用gceasy(https://gceasy.io/)这样的日志分析工具，很容易定位到问题。之所以选择采用工具，是因为gc日志看起来实在是太麻烦了，gceasy号称是AI学习分析问题，可视化做的较好。



# 20、假如生产环境CPU占用过高，请谈谈你的分析思路和定位？

1、首先，使用top -H命令获取占用CPU最高的线程，并将它转化为16进制。

2、然后，使用jstack命令获取应用的栈信息，搜索这个16进制。这样能够方便的找到引起CPU占用过高的具体原因。如果有条件的话，直接使用arthas进行操作就好了，不用再做这些费事费力的操作。

# 21、什么是类加载器，类加载器有哪些？

类加载器的作用：负载将的class文件加载到java虚拟机中，并为之创建一个Class对象从Java虚拟机的角度来讲，只存在如下两种不同的类加载器：

1、启动类加载器（Bootstrap ClassLoader）, 这个类加载器使用C++语言实现，是虚拟机自身的一部分

2、其他类加载器，这些类加载器都由Java语言实现，独立于虚拟机外部，并且全部都继承自抽象类（java.lang.ClassLoader）

从Java开发人员的角度来讲，类加载器还可以划分的更细致一下，绝大部分Java程序都会使用到以下3种系统提供的类加载器：

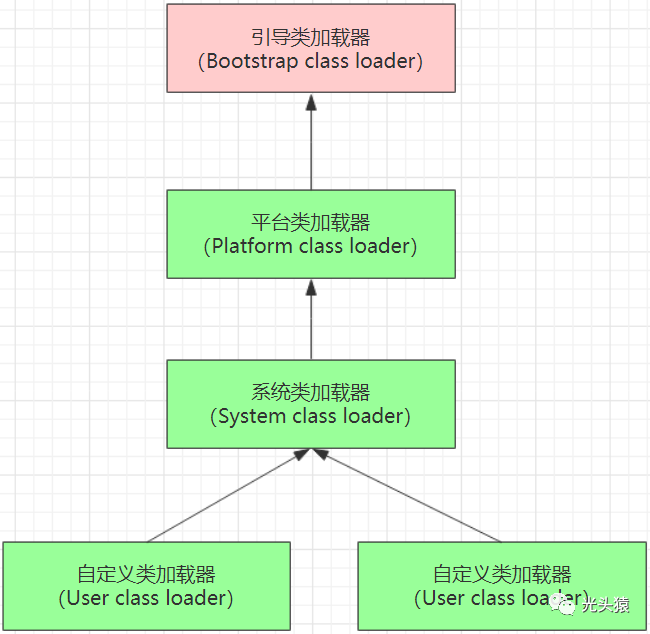
1、启动类加载器（Bootstrap class loader）：它是虚拟机的内置类加载器，通过表示为null

2、平台类加载器（Platform class loader）  ：它是平台类加载器；负责加载JDK中一些特殊的模块；

3、系统类加载器（System class loader）    ：它也被称为应用程序类加载器， 它负责加载用户类路径上所指定的类库，一般情况下这个就是程序中默认的类加载器

# 22、Java的双亲委托机制是什么？

概述：我们的应用程序都是由这三种类加载器互相配合进行加载的，如果有必要，还可以加入自定义的类加载器。这些类加载器之间的层次关系一般会如下图所示：



上图所展示的类加载器之间的这种层次关系，就称之为类加载器的双亲委派模型。双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都应该有自己的父类加载器。这里的类加载器的父子关系不是真正物理意义上的继承，而是逻辑上的继承。

工作过程：

双亲委派模型的工作过程是：如果一个类加载器收到了类加载的请求，它首先不会自己尝试加载这个类，而是把这请求委派给父类加载器去完成，没一个层次的类加载器都是如此，因此所有的加载请求最终都应该传说到顶层的启动类加载器中，只有当父类加载器返回自己无法完成这个加载请求（它的搜索返回中没有找到所需的类）时，子类加载器才会尝试自己去加载。

# 23、调优命令有哪些？

1、jps，JVM Process Status Tool显示指定系统内所有的HotSpot虚拟机进程。

2、jstat，JVM statistics Monitoring是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。

查询帮助文档：jstat -options

图片  
  


3、jmap，JVM Memory Map命令用于查看堆内存的分配情况以及生成heap dump文件

查询帮助文档：jmap -h

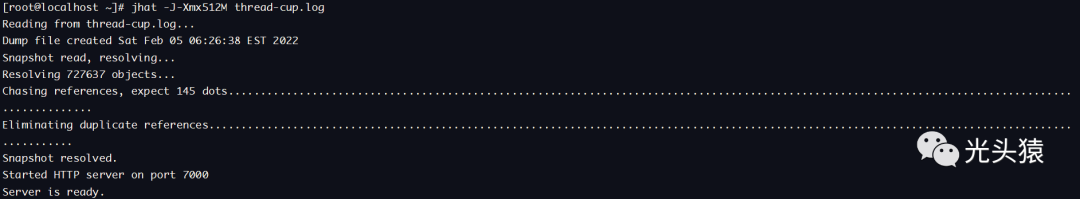
示例1：jmap -heap 33193 查询堆内存的分配情况

示例2：jmap -dump:format=b,file=thread-cup.log 33193

4、jhat，JVM Heap Analysis Tool命令是与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看查询帮助文档：

jhat -h

示例：jhat -J-Xmx512M thread-cup.log



5、jstack，用于生成java虚拟机当前时刻的线程快照。

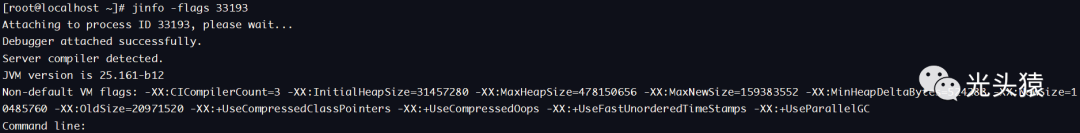
查看帮助文档：jstack -h

示例：jstack -l 33193

6、jinfo，JVM Configuration info 这个命令作用是实时查看和调整虚拟机运行参数。

查看帮助文档：jinfo -h

示例：jinfo -flags 33193



# 24、你知道哪些JVM性能调优参数？

1、设定堆内存大小：

-Xms 设置最小堆内存大小(不能小于1024K)

-Xmx 设置最大堆内存大小(不能小于1024K)

2、设定新生代大小：

-XX:NewSize：新生代大小

-XX:NewRatio 新生代和老生代占比

3、-XX:SurvivorRatio：伊甸园空间和幸存者空间的占比

4、设定垃圾回收器

年轻代用 -XX:+UseParNewGC

年老代用-XX:+UseConcMarkSweepGC

# 25、你用过哪些性能调优工具？

常用调优工具分为两类

1、jdk自带监控工具

* jconsole，Java Monitoring and Management Console是从java5开始，在JDK中自带的java监控和管理控制台，用于对JVM中内存，线程和类等的监控
* jvisualvm，jdk自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC变化等。

2、第三方

* MAT，Memory Analyzer Tool，一个基于Eclipse的内存分析工具，是一个快速、功能丰富的Java heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗
* GChisto，一款专业分析gc日志的工具

# 26、你都有哪些手段用来排查内存溢出？

内存溢出包含很多种情况，我在平常工作中遇到最多的就是堆溢出。有一次线上遇到故障，重新启动后，使用jstat命令，发现Old区在一直增长。我使用jmap命令，导出了一份线上堆栈，然后使用MAT进行分析。通过对GC Roots的分析，我发现了一个非常大的HashMap对象，这个原本是有位同学做缓存用的，但是一个无界缓存，造成了堆内存占用一直上升。后来，将这个缓存改成 guava Cache，并设置了弱引用，故障就消失了。