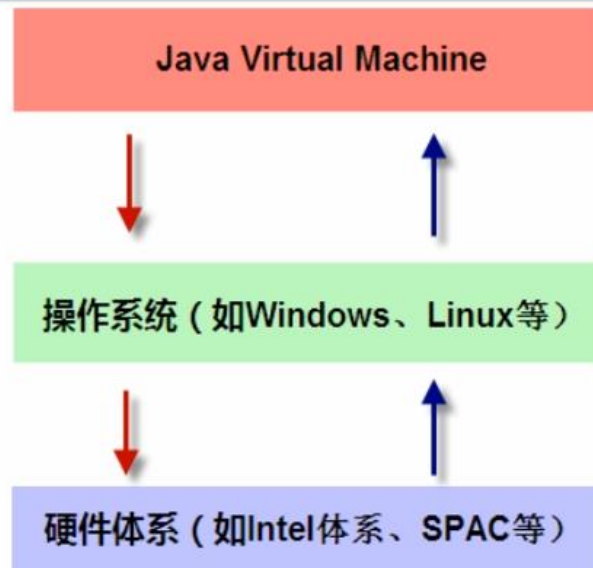


简历上写：熟悉 jvm 架构和 GC 垃圾回收机制以及相应的参数调优，有在 linux 进行系统优化的经验。

01JVM 解析

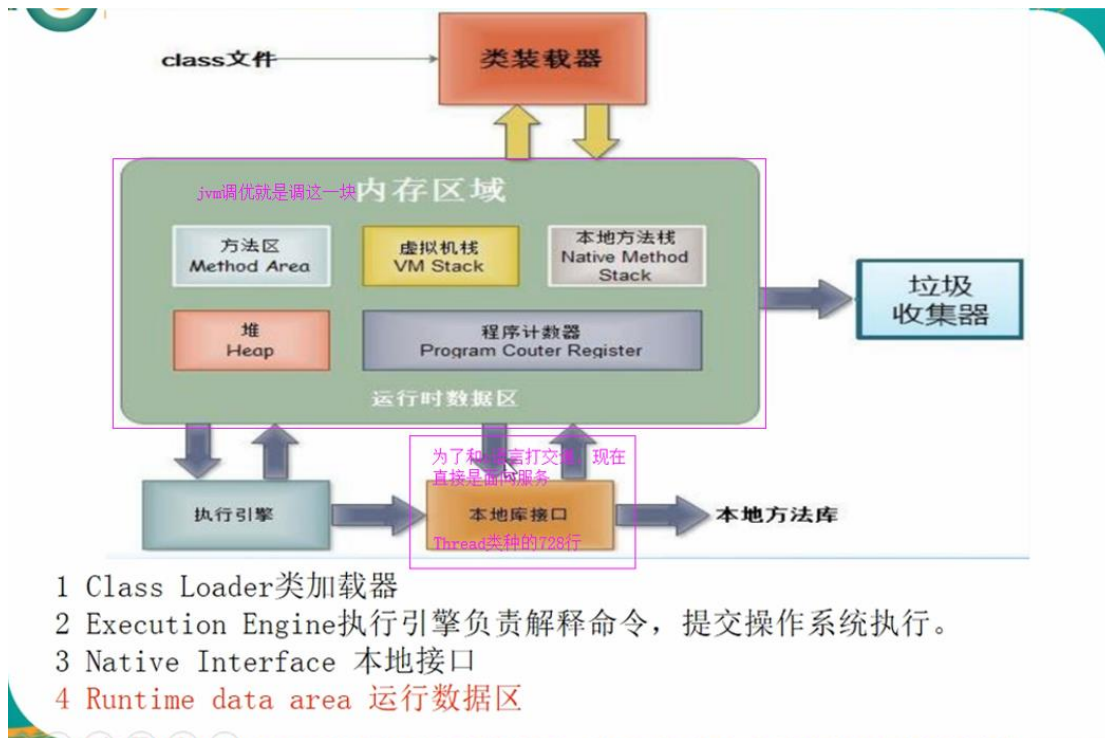
• JVM组成结构谈谈



JVM是运行在操作系统之上的，它与硬件没有直接的交互

JVM 是运行在操作系统之上的，它与硬件没有直接的交互。

不要说 jvm 是 java 虚拟机（面试官不想听）



多线程与操作系统有关系

Thread.state(线程的 5 种状态)

new
 runnable
 blocked
 waiting timed_waiting
 terminated

```
private native void start0();
```

让 java 去找操作系统（直接调系统）。

1 Class Loader类加载器

负责加载class文件，class文件在文件开头有特定的文件标示，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

2 Native Interface

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 诞生的时候是 C/C++横行的时候，要想立足，必须有调用 C/C++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 Native Method Stack中登记 native方法，在Execution Engine 执行时加载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的通信很发达，比如可以使用 Socket 通信，也可以使用 Web Service等等，不多做介绍。

重点

3 Method Area 方法区 优化一定是优化共享的 0.99优化堆, 0.01优化方法区

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

静态变量+常量+类信息+运行时常量池存在方法区中+
实例变量存在堆内存中

4 PC Register 程序计数器

每个线程都有一个程序计数器，就是一个指针，指向方法区中的方法字节码（下一个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，几乎可以忽略不记。

5 Native Method Stack 本地方法栈

它的具体做法是 Native Method Stack中登记native方法，在Execution Engine 执行时加载native libraies。

什么是栈什么是堆：栈管运行，堆管存储（8字真言）

02JVM(栈和堆)

栈存储什么

6 Stack 栈是什么

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，**对于栈来说不存在垃圾回收问题**，只要线程一结束该栈就Over，生命周期和线程一致，**是线程私有的**。**基本类型的变量和对象的引用变量都是在函数的栈内存中分配。**

6.1 栈存储什么？

栈帧中主要保存3 类数据：

本地变量（Local Variables）：**输入参数**和**输出参数**以及**方法内的变量**；
方法的参数 方法的返回值

栈操作（Operand Stack）：记录出栈、入栈的操作；

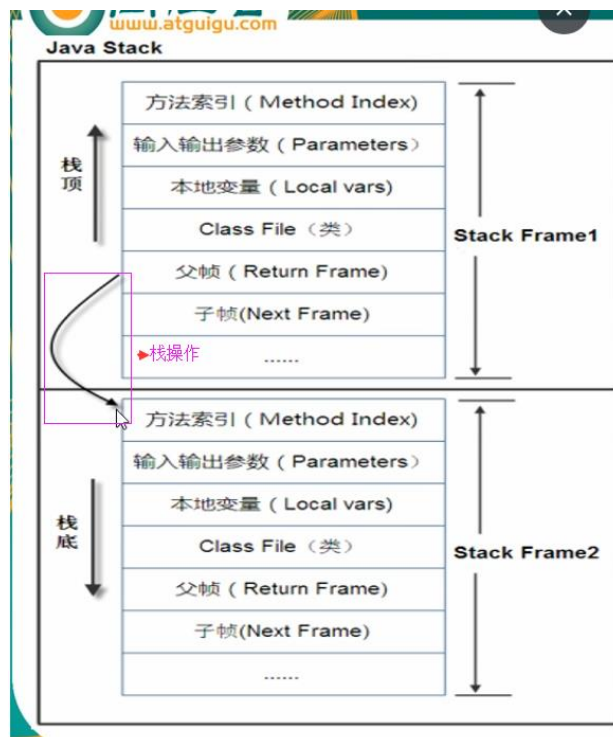
栈帧数据（Frame Data）：包括类文件、方法等等。

6.2 栈运行原理：

栈中的数据都是以**栈帧（Stack Frame）**的格式存在，栈帧是一个**内存区块**，是一个数据集，是一个有关方法(Method)和运行期数据的数据集，
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，于是产生栈帧 F2 也被压入栈，
B方法又调用了 C方法，于是产生栈帧 F3 也被压入栈，
.....

执行完毕后，先弹出F3栈帧，再弹出F2栈帧，再弹出F1栈帧.....

遵循“先进后出”/“后进先出”原则。



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。

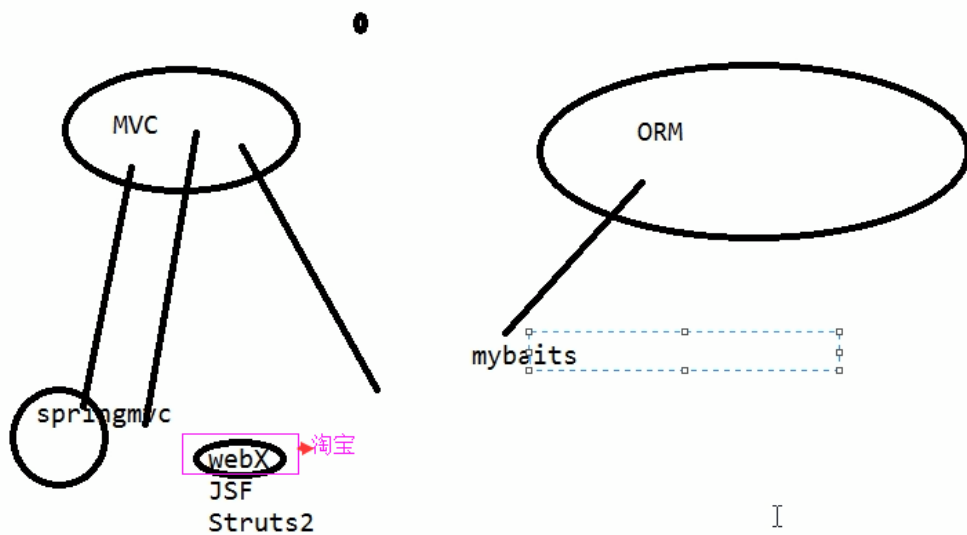
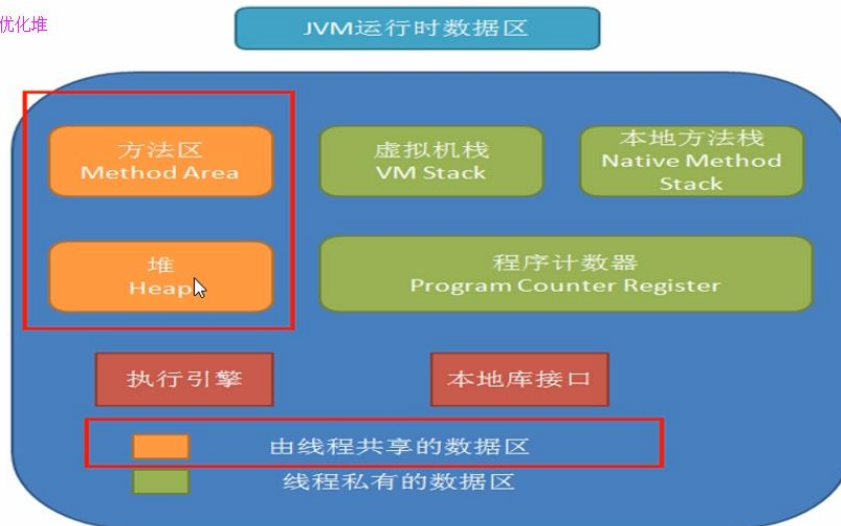
```
public class StackDemo {
    public static void main(String[] args) {
        test1(); // StackOverflowError
        System.out.println("main");
    }

    private static void test1() {
        test1();
    }
}
```

03jvm(JVM 是一种规范, HotSpot 是一种实现)

• 判断JVM优化是哪里?

主要是优化堆



jvm 总共有 7 种, 一般没说默认是 hotspot

三种 JVM(从 7 个活出了 3 个)

- 三种JVM

- Sun公司的HotSpot

oracle
合并产生java8

- BEA公司的JRockit

- IBM公司的J9 VM

堆

堆内存示意图：



new 一个对象 new 在堆中新生区的 Eden space 中

7.1 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。幸存者区有两个：0区（Survivor 0 space）和1区（Survivor 1 space）。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所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者0区。若幸存者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？再移动到养老区。若养老区也满了，那么这个时候将产生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，就会产生OOM异常“OutOfMemoryError”。
和看的书不一样

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）。

默认情况下：jvm 虚拟机的内存用固态内存的 0.25，对象存活次数最大 31 次

7.2 养老区

养老区用于保存从新生区筛选出来的 JAVA 对象，一般池对象都在这个区域活跃。

7.3 永久区

Object, 祖先对象就是元数据

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。

Jdk1.6及之前：有永久代，常量池1.6在方法区

Jdk1.7：有永久代，但已经逐步“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

java7叫永久代，java8叫元空间

• 熟悉三区结构后方可学习-JVM垃圾收集

Maven (Jar 包管理工具) 避免永久代益处，不用的 jar 包不会加载，避免最恐怖一种 bug,jar 包冲突。

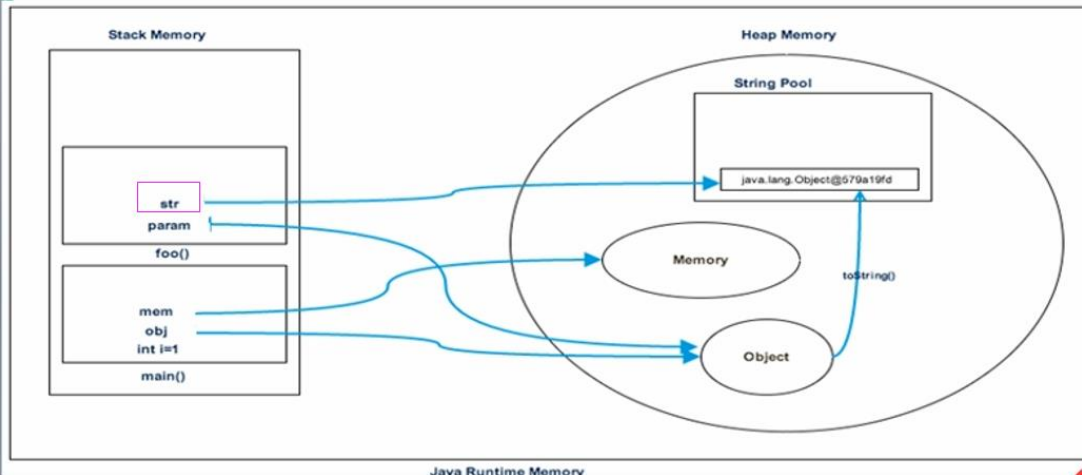
04JVM

- As soon as we run the program, it loads all the Runtime classes into the Heap space. When main() method is found at line 1, Java Runtime creates stack memory to be used by main() method thread.
- We are creating primitive local variable at line 2, so it's created and stored in the stack memory of main() method.
- Since we are creating an Object in line 3, it's created in Heap memory and stack memory contains the reference for it. Similar process occurs when we create Memory object in line 4.
- Now when we call foo() method in line 5, a block in the top of the stack is created to be used by foo() method. Since Java is pass by value, a new reference to Object is created in the foo() stack block in line 6.
- A string is created in line 7, it goes in the String Pool in the heap space and a reference is created in the foo() stack space for it.
- foo() method is terminated in line 8, at this time memory block allocated for foo() in stack becomes free.
- In line 9, main() method terminates and the stack memory created for main() method is destroyed. Also the program ends at this line, hence Java Runtime frees all the memory and end the execution of the program.

• 程序内存划分小总结(jdk1.7)

```
public static void main(String[] args){//Line 1
    int i=1; // Line 2
    Object obj = new Object();// Line 3
    Memory mem = new Memory();// Line 4
    mem.foo(obj); //Line 5
} // Line 9

private void foo(Object param) { // Line 6
    String str = param.toString(); // Line 7
    System.out.println(str);
} // Line 8
```

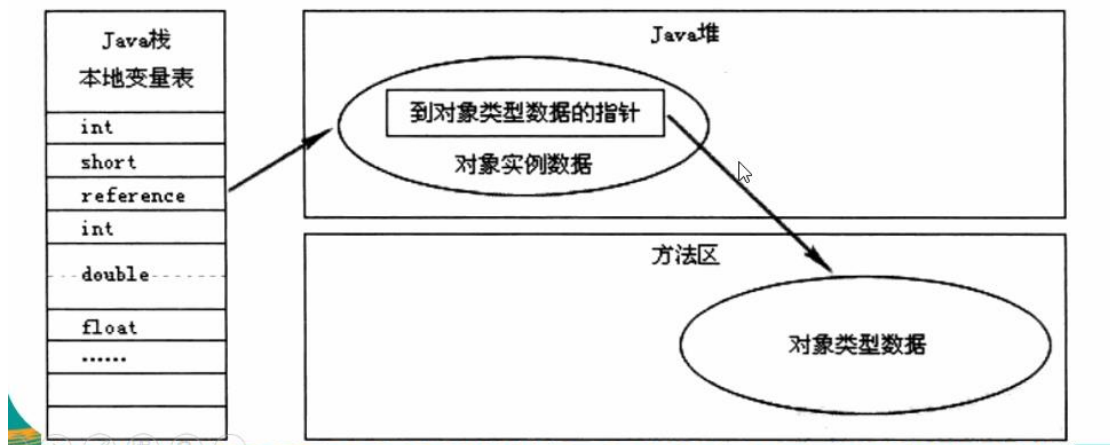


- As soon as we run the program, it loads all the Runtime classes into the Heap space. When `main()` method is found at line 1, Java Runtime creates stack memory to be used by `main()` method thread.
- We are creating primitive local variable at line 2, so it's created and stored in the stack memory of `main()` method.
- Since we are creating an `Object` in line 3, it's created in Heap memory and stack memory contains the reference for it. Similar process occurs when we create `Memory` object in line 4.
- Now when we call `foo()` method in line 5, a block in the top of the stack is created to be used by `foo()` method. Since Java is pass by value, a new reference to `Object` is created in the `foo()` stack block in line 6.
- A string is created in line 7, it goes in the **String Pool** in the heap space and a reference is created in the `foo()` stack space for it.
java6常量池在方法区中, java7常量池在堆, java8常量池在元空间
- `foo()` method is terminated in line 8, at this time memory block allocated for `foo()` in stack becomes free.
- In line 9, `main()` method terminates and the stack memory created for `main()` method is destroyed. Also the program ends at this line, hence Java Runtime frees all the memory and end the execution of the program.

- A string is created in line 7, it goes in the **String Pool** in the heap space and a reference is created in the `foo()` stack space for it.

严格来说：Heap和Method Area是分开的！！ (1.6)

下图通过直接指针访问对象



java 内存真实示意图

java7（包括 7 之前）

堆由新生代，老年代，持久代组成但实际上是分裂的（持久代只是逻辑上让这三个代归于一统，就好比说持久代是接口，方法区是持久代的实现）

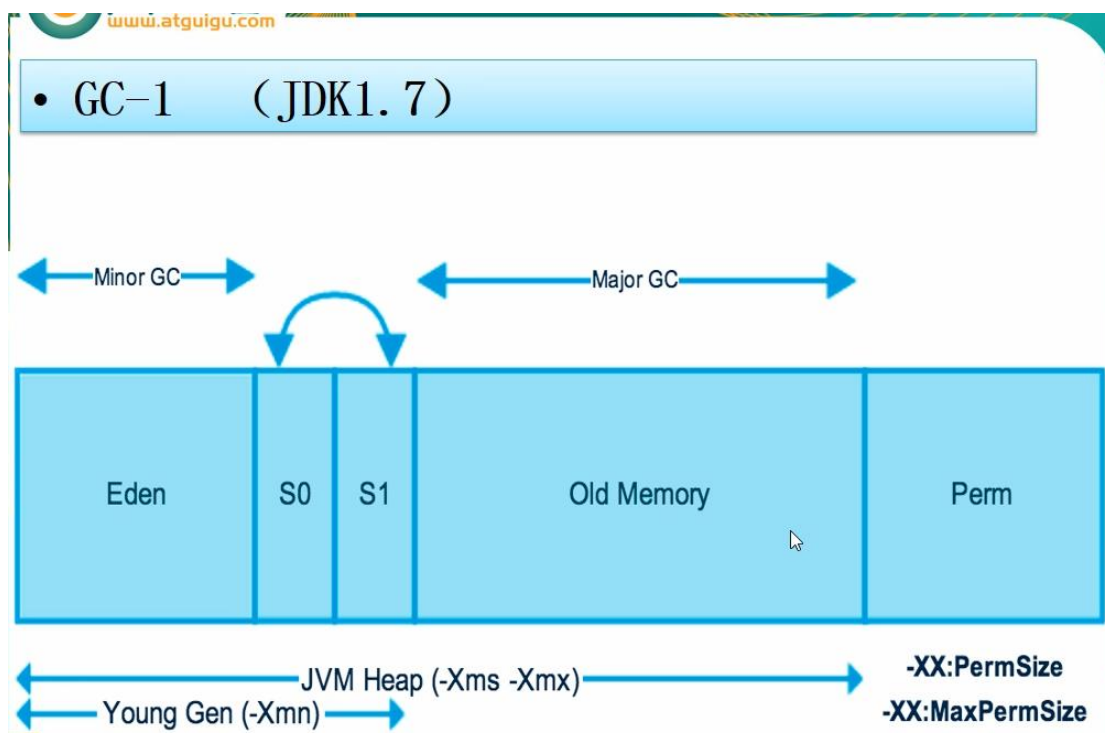
实际而言，方法区（Method Area）和堆一样，是各个线程共享的内存区域，它用于存储虚拟机加载的：类信息+普通常量+静态常量+编译器编译后的代码等等，虽然JVM规范将方法区描述为堆的一个逻辑部分，但它却还有一个别名叫做Non-Heap(非堆)，目的就是要和堆分开。

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，或者说使用永久代来实现方法区而已，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、接口等描述信息外，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。

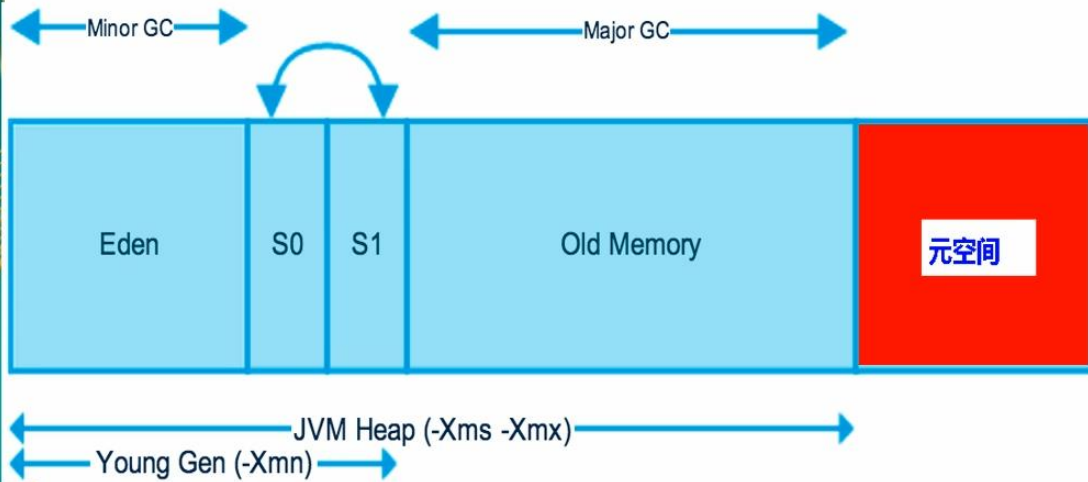


jvm 调优



• GC-2 (JDK1.8)

- 1 JDK 1.8之后将最初的永久代取消了，由元空间取代。
- 2 目的：将HotSpot与JRockit两个虚拟机标准



• GC-3 (堆内存调优简介)

-Xms	start	设置初始分配大小，默认为物理内存的“1 / 64”
-Xmx	max	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");  
}
```

VM arguments:

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




```

Javadoc Declaration Console Servers
<terminated> JVMShowcase [Java Application] D:\devSoft\Java\jdk1.7.0_51\bin\javaw.exe (Dec 5, 2016, 12:49:30 AM)
MAX_MEMORY = 1029701632 (字节)、982.0MB
TOTAL_MEMORY = 1029701632 (字节)、982.0MB
Heap
PSYoungGen total 306176K used 10506K [0x00000000eaa80000, 0x0000000100000000, 0x0000000100000000)
eden space 262656K, 4% used [0x00000000eaa80000, 0x00000000eb4c29c8, 0x00000000fab00000)
from space 43520K, 0% used [0x00000000fd580000, 0x00000000fd580000, 0x0000000100000000)
to space 43520K, 0% used [0x00000000fab00000, 0x00000000fab00000, 0x00000000fd580000)
ParOldGen total 699392K used 0K [0x00000000bff80000, 0x00000000eaa80000, 0x00000000eaa80000)
object space 699392K, 0% used [0x00000000bff80000, 0x00000000bff80000, 0x00000000eaa80000)
PSPermGen total 21504K, used 2533K [0x00000000bad80000, 0x00000000bc280000, 0x00000000bff80000)
object space 21504K, 11% used [0x00000000bad80000, 0x00000000baff9490, 0x00000000bc280000)

```

垃圾回收



尚硅谷
www.atguigu.com

• GC-3.3(自动触发垃圾回收)

```
String str = "www.atguigu.com";
while(true)
{
    str += str + new Random().nextInt(88888888) + new Random().nextInt(999999999);
}

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

```
GC [PSYoungGen: 2182K->504K(3072K)] 2182K->1020K(9216K), 0.0016934 secs] [Times: user=0.00 sys=0.00,
GC [PSYoungGen: 2174K->328K(3072K)] 2691K->1868K(9216K), 0.0019458 secs] [Times: user=0.06 sys=0.00,
GC [PSYoungGen: 2397K->296K(3072K)] 3937K->2860K(9216K), 0.0011896 secs] [Times: user=0.00 sys=0.00,
GC [PSYoungGen: 1320K->264K(3072K)] 5932K->4876K(9728K), 0.0008658 secs] [Times: user=0.00 sys=0.00,
GC-- [PSYoungGen: 2312K->2312K(3072K)] 6924K->6924K(9728K), 0.0009400 secs] [Times: user=0.00 sys=0.
Full GC [PSYoungGen: 2312K->0K(3072K)] [ParOldGen: 4612K->2528K(6656K)] 6924K->2528K(9728K) [PSPermG
Full GC [PSYoungGen: 2048K->0K(3072K)] [ParOldGen: 6624K->6624K(6656K)] 8672K->6624K(9728K) [PSPermG
GC [PSYoungGen: 0K->0K(3072K)] 6624K->6624K(9728K), 0.0004381 secs] [Times: user=0.00 sys=0.00, real
Full GC [PSYoungGen: 0K->0K(3072K)] [ParOldGen: 6624K->6613K(6656K)] 6624K->6613K(9728K) [PSPermGen:
```

面试题

• GC-4(面试题)

1) StackOverflowError和OutOfMemoryError，谈谈你的理解

2)一般什么时候会发生GC？如何处理？

答：Java中的GC会有两种回收：年轻代的Minor GC，另外一个就是老年代的Full GC；新对象创建时如果伊甸园空间不足会触发MinorGC，如果此时老年代的内存空间不足会触发Full GC，如果空间都不足抛出OutOfMemoryError。

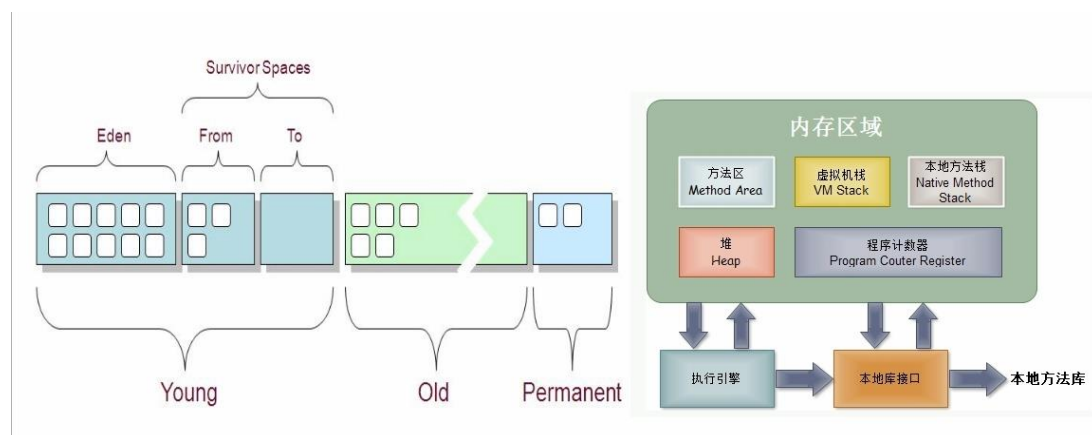
3) GC回收策略，谈谈你的理解

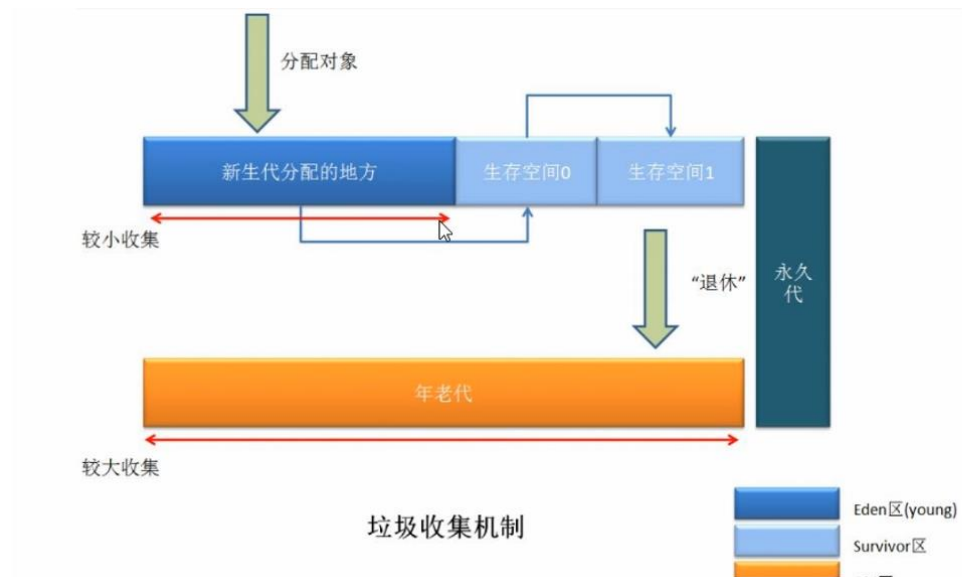
答：

年轻代(伊甸园区+两个幸存者区)，GC回收策略为“复制”；老年代的保存空间一般较大，GC回收策略为“整理-压缩”；

05jvm（复制清除算法）

jvm 复习





JVM在进行GC时，并非每次都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类型，一种是普通GC（minor GC），一种是全局GC（major GC or Full GC），

普通GC（minor GC）：只针对**新生代区域**的GC。

全局GC（major GC or Full GC）：针对**年老代**的GC，偶尔伴随对新生代的GC以及对永久代的GC。

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。

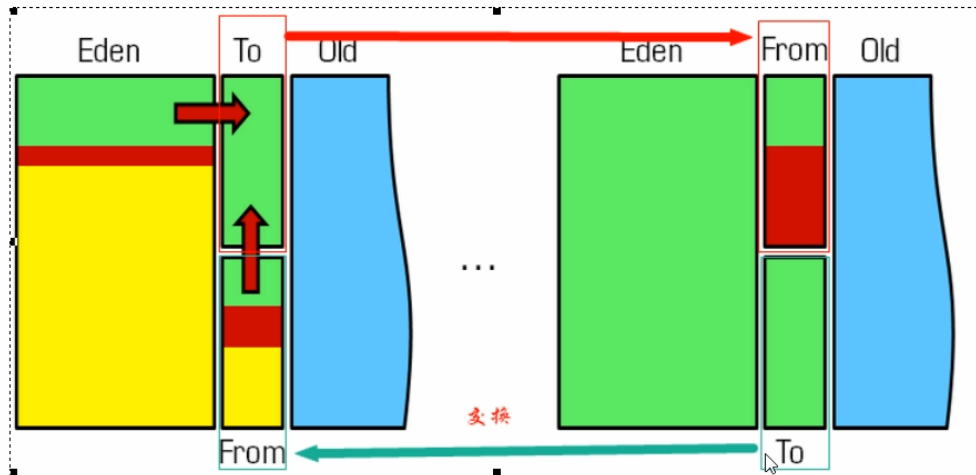
当对象在Eden（包括一个Survivor区域，这里假设是from区域）出生后，在经过一次Minor GC后，如果对象还存活，并且能够被另外一块Survivor区域所容纳（上面已经假设为from区域，这里应为to区域，即to区域有足够的内存空间来存储Eden和from区域中存活的对象），则使用**复制算法**将这些仍然还存活的对象复制到另外一块Survivor区域（即to区域）中，然后清理所使用过的Eden以及Survivor区域（即from区域），并且将这些对象的年龄设置为1，以后对象在Survivor区每熬过一次Minor GC，就将对象的年龄+1，当对象的年龄达到某个值时（默认是15岁，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

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，就会被移动到老年代中。因为年轻代中的对象基本都是朝生夕死的（80%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。**

在GC开始的时候，对象只会存在于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，仍存活的对象会根据他们的年龄值来决定去向。年龄达到一定值（年龄阈值，可以通过-XX:MaxTenuringThreshold来设置）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”就是上次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Survivor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，会将所有对象移动到老年代中。



Minor GC回收的是eden区和一个from区

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间，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%的活动区间与另外80%中存活的对象转移到10%的空闲区间，接下来，将之前90%的内存全部释放，以此类推。

I

优点是不会产生内存碎片，缺点浪费了内存空间。

复制算法弥补了标记/清除算法中，内存布局混乱的缺点。不过与此同时，它的缺点也是相当明显的。

1、它浪费了一半的内存，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，我们必须克服50%内存的浪费。

2

06jvm

标记清除算法

1. 标记 (Mark):

从根集合开始扫描，对存活的对象进行标记。



2. 清除 (Sweep):

扫描整个内存空间，回收未被标记的对象，使用free-list记录可以区域。



✓ 不需要额外空间

✗ 两次扫描，耗时严重；
✗ 会产生**内存碎片**

比如：arralist 和 linklist 整合产生了 hashmap,数组+链表+红黑树

劣势

标记清除的劣势

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

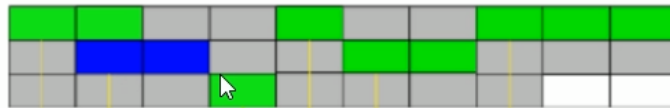
2、其次，主要的缺点则是这种方式清理出来的**空闲内存是不连续**的，这点不难理解，我们的死亡对象都是随即的出现在内存的各个角落的，现在把它们清除之后，内存的布局自然会乱七八糟。而为了应付这一点，JVM就不得不维持一个内存的空闲列表，这又是一种开销。而且在分配数组对象的时候，寻找连续的内存空间会不太好找。

标记整理算法

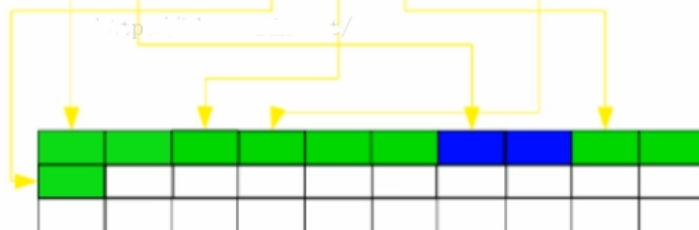
标记-整理 (Mark-Compact) :

标的是活的

Before GC



After GC



劣势

标记整理的劣势

标记/整理算法唯一的缺点就是效率也不高，不仅要标记所有存活对象，还要整理所有存活对象的引用地址。从效率上来说，标记/整理算法要低于复制算法。

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

难道就没有一种最优算法吗？猜查看，下面还有

回答：无，没有最好的算法，只有最合适的算法。=====>分代收集算法。