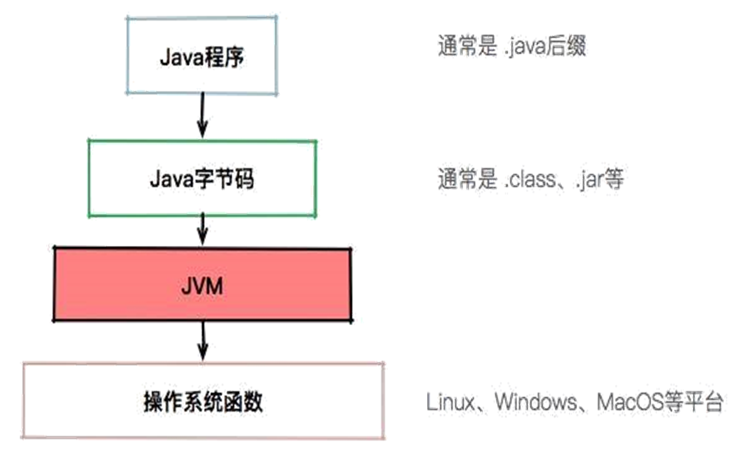
# JVM与操作系统的关系

## Java Virtual Machine

JVM 全称 Java Virtual Machine，也就是我们耳熟能详的 Java 虚拟机。它能识别 .class后缀的文件，并且能够解析它的指令，最终调用操作系统上的函数，完成我们想要的操作。

## 翻译

Java 程序不一样，使用 javac 编译成 .class 文件之后，还需要使用 Java 命令去主动执行它，操作系统并不认识这些 .class 文件。所以JVM就是一个翻译。



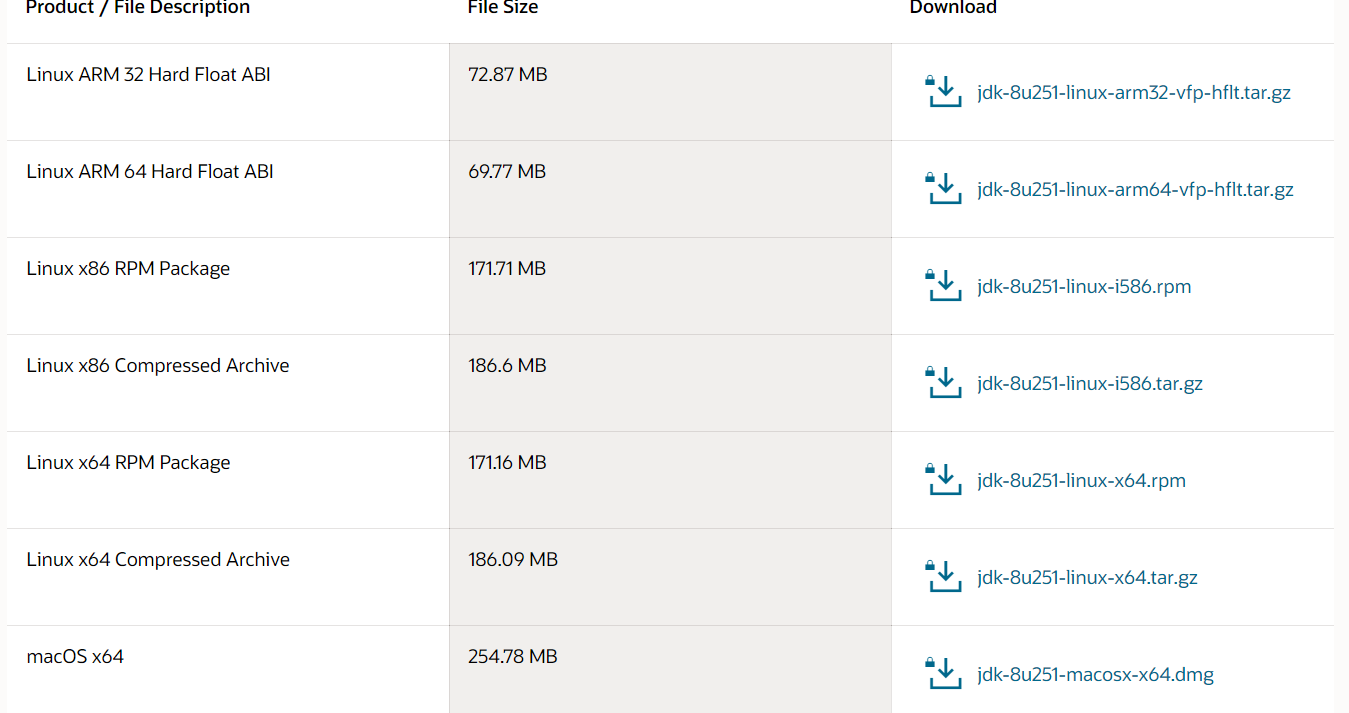
从图中可以看到，有了 JVM 这个抽象层之后，Java 就可以实现跨平台了。JVM 只需要保证能够正确执行 .class 文件，就可以运行在诸如 Linux、Windows、MacOS 等平台上了。

## 从跨平台到跨语言

**跨平台：**我们写的这个类Person这个类，在不同的操作系统上（Linux、Windows、MacOS 等平台）执行，效果是一样，这个就是JVM的跨平台性。

为了实现跨平台型，不同操作系统有不同的JDK的版本

<https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html>



**跨语言：**JVM只识别字节码，所以JVM其实跟语言是解耦的，也就是没有直接关联，并不是它翻译Java文件，而是识别class文件，这个一般称之为字节码。还有像Groovy 、Kotlin、Jruby等等语言，它们其实也是编译成字节码，所以也可以在JVM上面跑，这个就是JVM的跨语言特征。

# JVM、JRE、JDK的关系

JVM只是一个翻译，把Class翻译成机器识别的代码，但是需要注意，JVM 不会自己生成代码，需要大家编写代码，同时需要很多依赖类库，这个时候就需要用到JRE。

JRE是什么，它除了包含JVM之外，提供了很多的类库（就是我们说的jar包，它可以提供一些即插即用的功能，比如读取或者操作文件，连接网络，使用I/O等等之类的）这些东西就是JRE提供的基础类库。JVM 标准加上实现的一大堆基础类库，就组成了 Java 的运行时环境，也就是我们常说的 JRE（Java Runtime Environment）。

但对于程序员来说，JRE还不够。我写完要编译代码，还需要调试代码，还需要打包代码、有时候还需要反编译代码。所以我们会使用JDK，因为JDK还提供了一些非常好用的小工具，比如 javac（编译代码）、java、jar （打包代码）、javap（反编译<反汇编>）等。这个就是JDK。

具体可以文档可以通过官网去下载：<https://www.oracle.com/java/technologies/javase-jdk8-doc-downloads.html>

# JVM整体

一个 Java 程序，首先经过 javac 编译成 .class 文件，然后 JVM 将其加载到方法区，执行引擎将会执行这些字节码。执行时，会翻译成操作系统相关的函数。JVM 作为 .class 文件的翻译存在，输入字节码，调用操作系统函数。

过程如下：Java 文件->编译器>字节码->JVM->机器码。

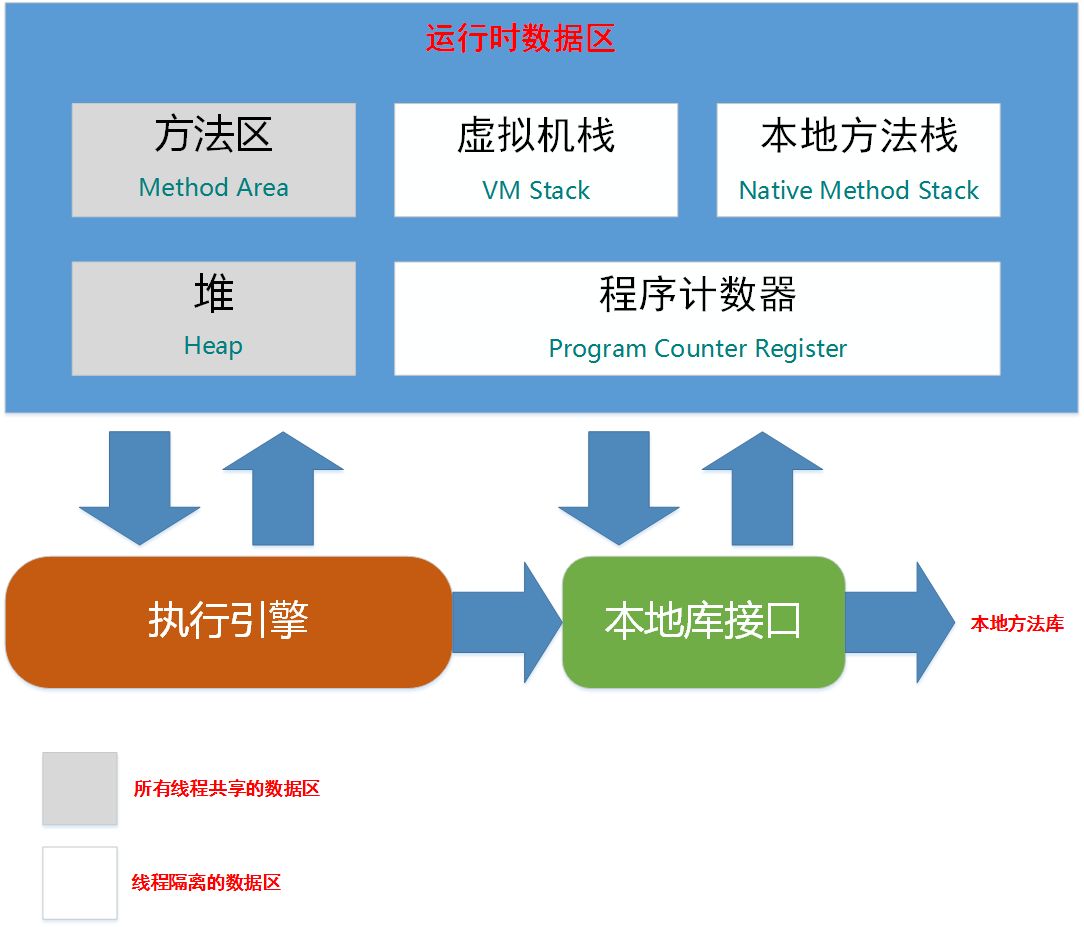
解释执行与GIT：

我们所说的 JVM，狭义上指的就 HotSpot（因为JVM有很多版本，但是使用最多的是HotSpot）。如非特殊说明，我们都以 HotSpot 为准。Java 之所以成为跨平台，就是由于 JVM 的存在。Java 的字节码，是沟通 Java 语言与 JVM 的桥梁，同时也是沟通 JVM 与操作系统的桥梁。

# 运行时数据区域

Java 引以为豪的就是它的自动内存管理机制。相比于 C++的手动内存管理、复杂难以理解的指针等，Java 程序写起来就方便的多。

在 Java 中，JVM 内存主要分为堆、程序计数器、方法区、虚拟机栈和本地方法栈。



## 程序计数器

较小的内存空间，当前线程执行的字节码的行号指示器；各线程之间独立存储，互不影响。

程序计数器是一块很小的内存空间，主要用来记录各个线程执行的字节码的地址，例如，分支、循环、跳转、异常、线程恢复等都依赖于计数器。

由于 Java 是多线程语言，当执行的线程数量超过 CPU 核数时，线程之间会根据时间片轮询争夺 CPU 资源。如果一个线程的时间片用完了，或者是其它原因导致这个线程的 CPU 资源被提前抢夺，那么这个退出的线程就需要单独的一个程序计数器，来记录下一条运行的指令。

程序计数器也是JVM中唯一不会OOM(OutOfMemory)的内存区域

## 虚拟机栈

栈是什么样的数据结构？先进后出(FILO)的数据结构，

虚拟机栈在JVM运行过程中存储当前线程运行方法所需的数据，指令、返回地址**。**

Java 虚拟机栈是基于线程的。哪怕你只有一个 main() 方法，也是以线程的方式运行的。在线程的生命周期中，参与计算的数据会频繁地入栈和出栈，栈的生命周期是和线程一样的。

栈里的每条数据，就是栈帧。在每个 Java 方法被调用的时候，都会创建一个栈帧，并入栈。一旦完成相应的调用，则出栈。所有的栈帧都出栈后，线程也就结束了。

每个栈帧，都包含四个区域：(局部变量表、操作数栈、动态连接、返回地址)

栈的大小缺省为1M，可用参数 –Xss调整大小，例如-Xss256k

**局部变量表:**顾名思义就是局部变量的表，用于存放我们的局部变量的。首先它是一个32位的长度，主要存放我们的Java的八大基础数据类型，一般32位就可以存放下，如果是64位的就使用高低位占用两个也可以存放下，如果是局部的一些对象，比如我们的Object对象，我们只需要存放它的一个引用地址即可。

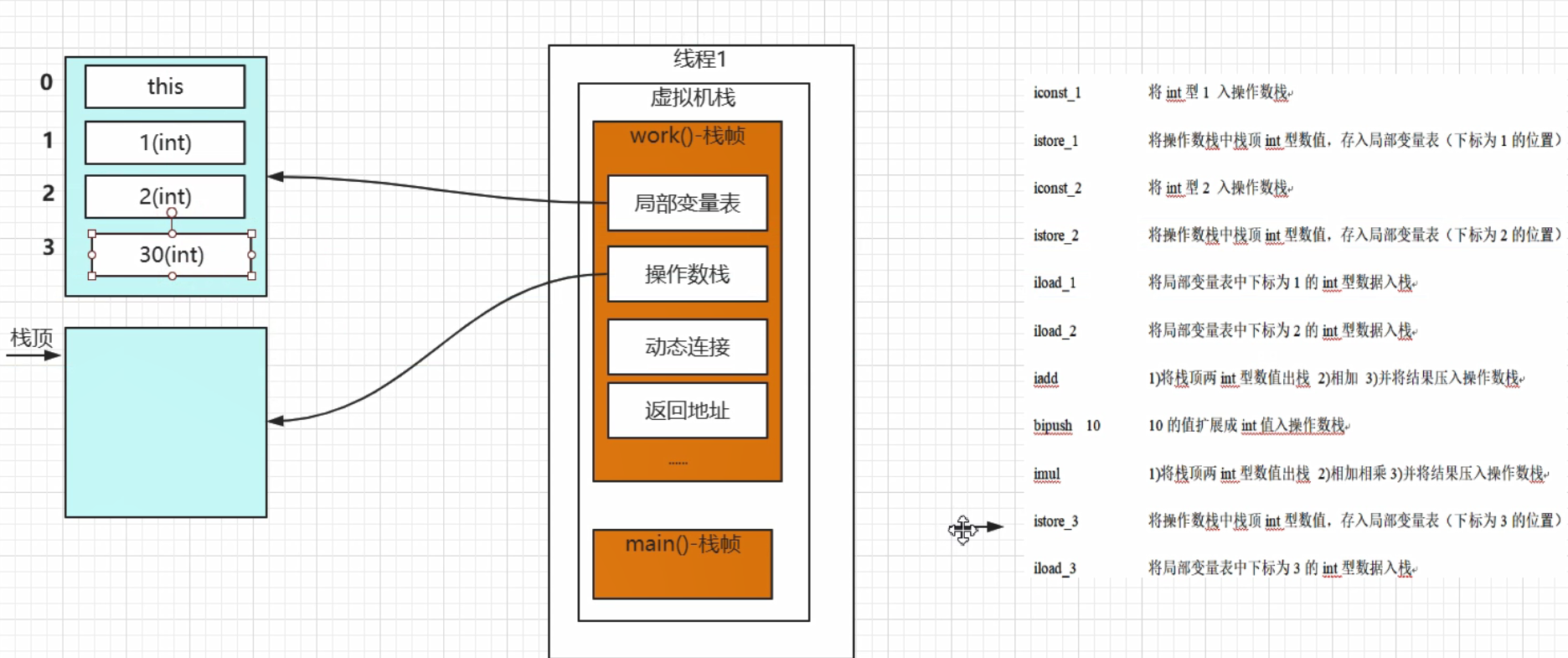
**操作数据栈**：存放我们方法执行的操作数的，它就是一个栈，先进后出的栈结构，操作数栈，就是用来操作的，操作的的元素可以是任意的java数据类型，所以我们知道一个方法刚刚开始的时候，这个方法的操作数栈就是空的，操作数栈运行方法就是JVM一直运行入栈/出栈的操作

**动态连接:**Java语言特性多态（需要类运行时才能确定具体的方法）。

**返回地址:**正常返回（调用程序计数器中的地址作为返回）、异常的话（通过异常处理器表<非栈帧中的>来确定）

# 栈帧执行对内存区域的影响

字节码助记码解释地址：<https://cloud.tencent.com/developer/article/1333540>



在JVM中，基于解释执行的这种方式是基于栈的引擎，这个说的栈，就是操作数栈。

# 本地方法栈

本地方法栈跟 Java 虚拟机栈的功能类似，Java 虚拟机栈用于管理 Java 函数的调用，而本地方法栈则用于管理本地方法的调用。但本地方法并不是用 Java 实现的，而是由 C 语言实现的。

本地方法栈是和虚拟机栈非常相似的一个区域，它服务的对象是 native 方法。你甚至可以认为虚拟机栈和本地方法栈是同一个区域。

虚拟机规范无强制规定，各版本虚拟机自由实现 ，HotSpot直接把本地方法栈和虚拟机栈合二为一 。

# 线程共享的区域

## 方法区/永久代

很多开发者都习惯将方法区称为“永久代”，其实这两者并不是等价的。

HotSpot 虚拟机使用永久代来实现方法区，但在其它虚拟机中，例如，Oracle 的 JRockit、IBM 的 J9 就不存在永久代一说。因此，方法区只是 JVM 中规范的一部分，可以说，在 HotSpot 虚拟机中，设计人员使用了永久代来实现了 JVM 规范的方法区。

方法区主要是用来存放已被虚拟机加载的类相关信息，包括类信息、静态变量、常量、运行时常量池、字符串常量池。

JVM 在执行某个类的时候，必须先加载。在加载类（加载、验证、准备、解析、初始化）的时候，JVM 会先加载 class 文件，而在 class 文件中除了有类的版本、字段、方法和接口等描述信息外，还有一项信息是常量池 (Constant Pool Table)，用于存放编译期间生成的各种字面量和符号引用。

字面量包括字符串（String a=“b”）、基本类型的常量（final 修饰的变量），符号引用则包括类和方法的全限定名（例如 String 这个类，它的全限定名就是 Java/lang/String）、字段的名称和描述符以及方法的名称和描述符。

而当类加载到内存中后，JVM 就会将 class 文件常量池中的内容存放到运行时的常量池中；在解析阶段，JVM 会把符号引用替换为直接引用（对象的索引值）。

例如，类中的一个字符串常量在 class 文件中时，存放在 class 文件常量池中的；在 JVM 加载完类之后，JVM 会将这个字符串常量放到运行时常量池中，并在解析阶段，指定该字符串对象的索引值。运行时常量池是全局共享的，多个类共用一个运行时常量池，class 文件中常量池多个相同的字符串在运行时常量池只会存在一份。

方法区与堆空间类似，也是一个共享内存区，所以方法区是线程共享的。假如两个线程都试图访问方法区中的同一个类信息，而这个类还没有装入 JVM，那么此时就只允许一个线程去加载它，另一个线程必须等待。在 HotSpot 虚拟机、Java7 版本中已经将永久代的静态变量和运行时常量池转移到了堆中，其余部分则存储在 JVM 的非堆内存中，而 Java8 版本已经将方法区中实现的永久代去掉了，并用元空间（class metadata）代替了之前的永久代，并且元空间的存储位置是本地

**元空间大小参数：**

jdk1.7及以前（初始和最大值）：-XX:PermSize；-XX:MaxPermSize；

jdk1.8以后（初始和最大值）：-XX:MetaspaceSize； -XX:MaxMetaspaceSize

jdk1.8以后大小就只受本机总内存的限制（如果不设置参数的话）

JVM参数参考：<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html>

Java8 为什么使用元空间替代永久代，这样做有什么好处呢？

官方给出的解释是：

移除永久代是为了融合 HotSpot JVM 与 JRockit VM 而做出的努力，因为 JRockit 没有永久代，所以不需要配置永久代。

永久代内存经常不够用或发生内存溢出，抛出异常 java.lang.OutOfMemoryError: PermGen。这是因为在 JDK1.7 版本中，指定的 PermGen 区大小为 8M，由于 PermGen 中类的元数据信息在每次 FullGC 的时候都可能被收集，回收率都偏低，成绩很难令人满意；还有，为 PermGen 分配多大的空间很难确定，PermSize 的大小依赖于很多因素，比如，JVM 加载的 class 总数、常量池的大小和方法的大小等。

## 堆

堆是 JVM 上最大的内存区域，我们申请的几乎所有的对象，都是在这里存储的。我们常说的垃圾回收，操作的对象就是堆。

堆空间一般是程序启动时，就申请了，但是并不一定会全部使用。

随着对象的频繁创建，堆空间占用的越来越多，就需要不定期的对不再使用的对象进行回收。这个在 Java 中，就叫作 GC（Garbage Collection）。

那一个对象创建的时候，到底是在堆上分配，还是在栈上分配呢？这和两个方面有关：对象的类型和在 Java 类中存在的位置。

Java 的对象可以分为基本数据类型和普通对象。

对于普通对象来说，JVM 会首先在堆上创建对象，然后在其他地方使用的其实是它的引用。比如，把这个引用保存在虚拟机栈的局部变量表中。

对于基本数据类型来说（byte、short、int、long、float、double、char)，有两种情况。当你在方法体内声明了基本数据类型的对象，它就会在栈上直接分配。其他情况，都是在堆上分配。

**堆大小参数：**

-Xms：堆的最小值；

-Xmx：堆的最大值；

-Xmn：新生代的大小；

-XX:NewSize；新生代最小值；

-XX:MaxNewSize：新生代最大值；

例如- Xmx256m

# 直接内存

不是虚拟机运行时数据区的一部分，也不是java虚拟机规范中定义的内存区域；如果使用了NIO,这块区域会被频繁使用，在java堆内可以用directByteBuffer对象直接引用并操作；

这块内存不受java堆大小限制，但受本机总内存的限制，可以通过-XX:MaxDirectMemorySize来设置（默认与堆内存最大值一样），所以也会出现OOM异常。

# 从底层深入理解运行时数据区

开启HSDB工具

Jdk1.8启动JHSDB的时候必须将sawindbg.dll复制到对应目录的jre下



C:\Program Files\Java\jdk1.8.0\_101\lib

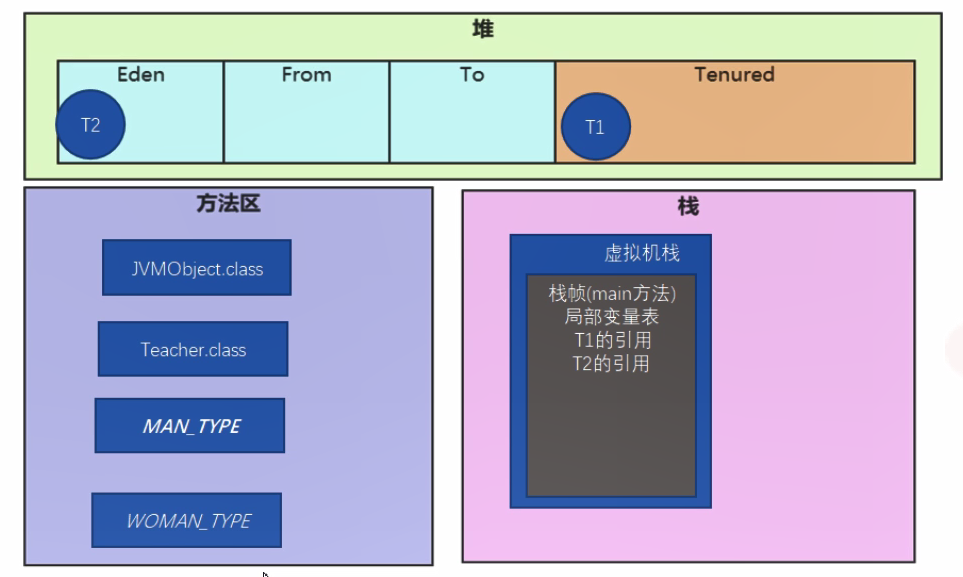
执行 java -cp .\sa-jdi.jar sun.jvm.hotspot.HSDB



当我们通过 Java 运行以上代码时，JVM 的整个处理过程如下：

1. JVM 向操作系统申请内存，JVM 第一步就是通过配置参数或者默认配置参数向操作系统申请内存空间。
2. JVM 获得内存空间后，会根据配置参数分配堆、栈以及方法区的内存大小。
3. 完成上一个步骤后， JVM 首先会执行构造器，编译器会在.java 文件被编译成.class 文件时，收集所有类的初始化代码，包括静态变量赋值语句、静态代码块、静态方法，静态变量和常量放入方法区
4. 执行方法。启动 main 线程，执行 main 方法，开始执行第一行代码。此时堆内存中会创建一个 Teacher 对象，对象引用 student 就存放在栈中。

执行其他方法时，具体的操作：栈帧执行对内存区域的影响。[栈帧执行对内存区域的影响](#_栈帧执行对内存区域的影响)



# 深入辨析堆和栈

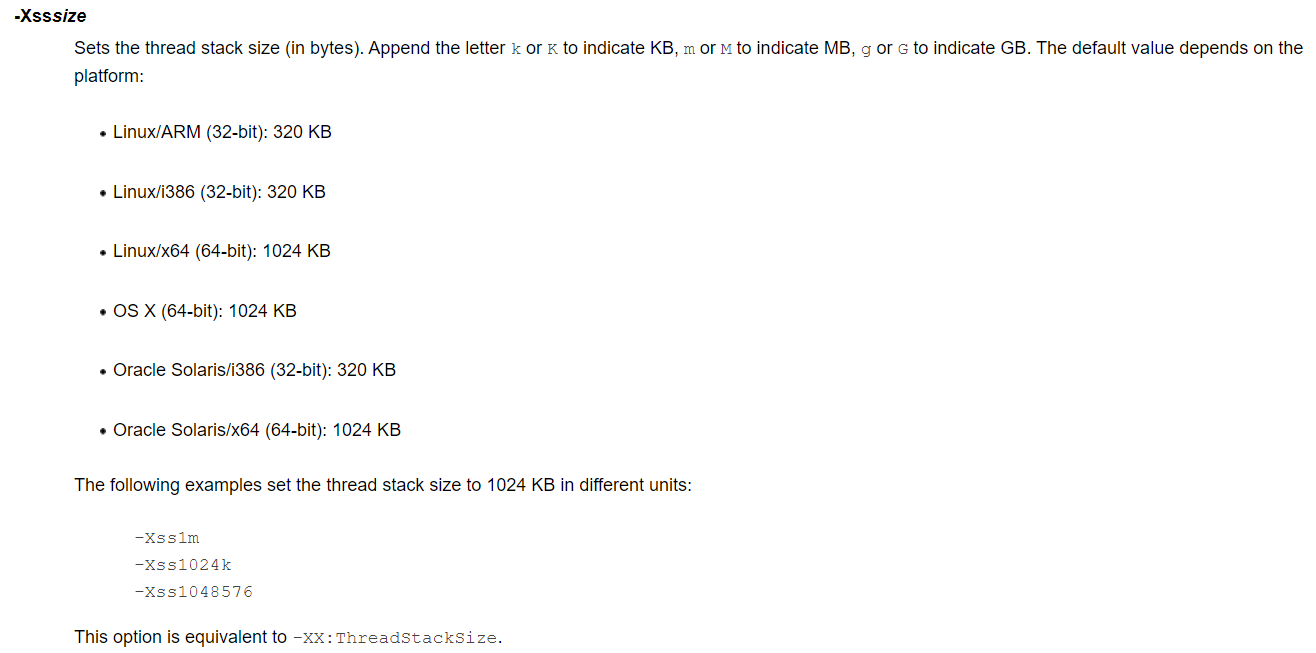
* **功能**
* 以栈帧的方式存储方法调用的过程，并存储方法调用过程中基本数据类型的变量（int、short、long、byte、float、double、boolean、char等）以及对象的引用变量，其内存分配在栈上，变量出了作用域就会自动释放；
* 而堆内存用来存储Java中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中；
* **线程独享还是共享**
* 栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存。
* 堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问。
* **空间大小**

栈的内存要远远小于堆内存

# 内存溢出

## 栈溢出

参数：-Xss1m， 具体默认值需要查看官网：<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html#BABHDABI>



HotSpot版本中栈的大小是固定的，是不支持拓展的。

java.lang.StackOverflowError 一般的方法调用是很难出现的，如果出现了可能会是无限递归。

虚拟机栈带给我们的启示：方法的执行因为要打包成栈桢，所以天生要比实现同样功能的循环慢，所以树的遍历算法中：递归和非递归(循环来实现)都有存在的意义。递归代码简洁，非递归代码复杂但是速度较快。

OutOfMemoryError：不断建立线程，JVM申请栈内存，机器没有足够的内存。（一般演示不出，演示出来机器也死了）

## 堆溢出

内存溢出：申请内存空间,超出最大堆内存空间。

如果是内存溢出，则通过 调大 -Xms，-Xmx参数。

如果不是内存泄漏，就是说内存中的对象却是都是必须存活的，那么久应该检查JVM的堆参数设置，与机器的内存对比，看是否还有可以调整的空间，再从代码上检查是否存在某些对象生命周期过长、持有状态时间过长、存储结构设计不合理等情况，尽量减少程序运行时的内存消耗。

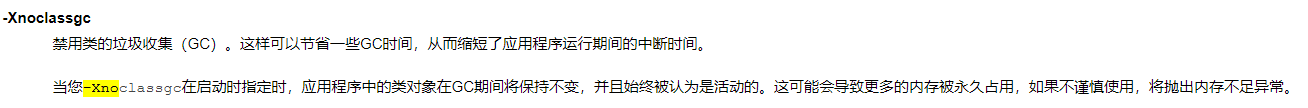
## 方法区溢出

**（1） 运行时常量池溢出**

**（2）方法区中保存的Class对象没有被及时回收掉或者Class信息占用的内存超过了我们配置。**

**注意Class要被回收，条件比较苛刻（仅仅是可以，不代表必然，因为还有一些参数可以进行控制）：**

1. **该类所有的实例都已经被回收，也就是堆中不存在该类的任何实例。**
2. 加载该类的ClassLoader已经被回收。
3. 该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。



**代码示例：**

**cglib是一个强大的，高性能，高质量的Code生成类库，它可以在运行期扩展Java类与实现Java接口。**

**CGLIB包的底层是通过使用一个小而快的**[**字节码**](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81)**处理框架ASM，来转换字节码并生成新的类。除了CGLIB包，**[**脚本语言**](https://baike.baidu.com/item/%E8%84%9A%E6%9C%AC%E8%AF%AD%E8%A8%80)**例如Groovy和BeanShell，也是使用ASM来生成java的字节码。当然不鼓励直接使用ASM，因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉。**

## 本机直接内存溢出

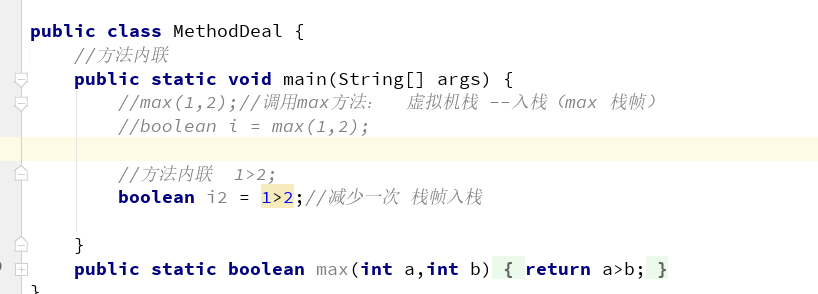
直接内存的容量可以通过MaxDirectMemorySize来设置（默认与堆内存最大值一样），所以也会出现OOM异常；

由直接内存导致的内存溢出，一个比较明显的特征是在HeapDump文件中不会看见有什么明显的异常情况，如果发生了OOM，同时Dump文件很小，可以考虑重点排查下直接内存方面的原因。

# 虚拟机优化技术

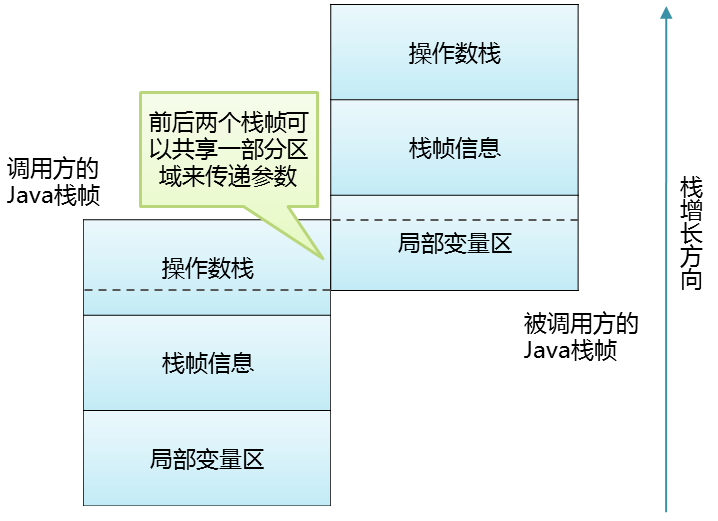
## 编译优化技术——方法内联

方法内联的优化行为，就是把目标方法的代码原封不动的“复制”到调用的方法中，避免真实的方法调用而已。

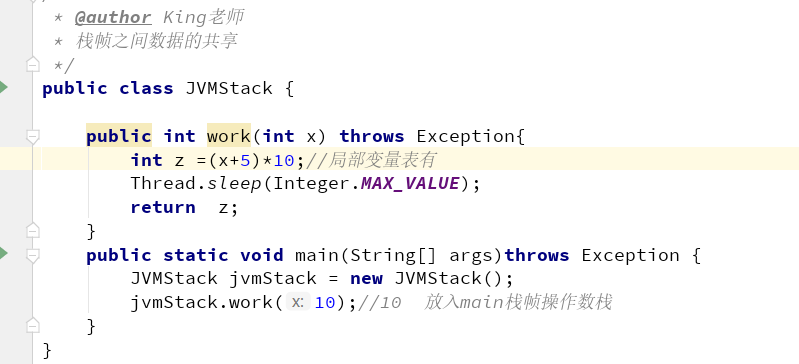


## 栈的优化技术——栈帧之间数据的共享

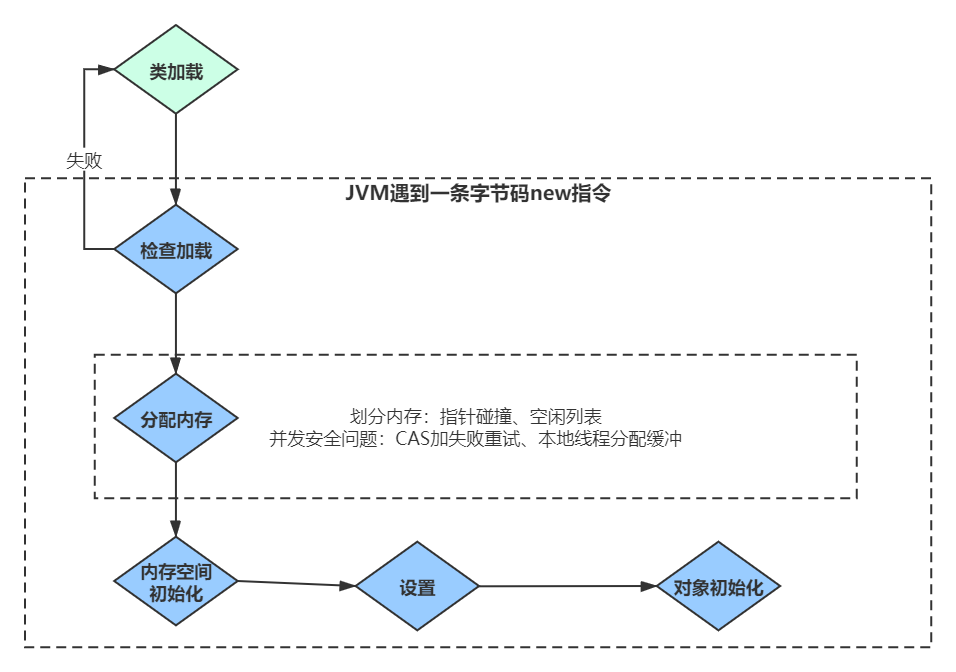
在一般的模型中，两个不同的栈帧的内存区域是独立的，但是大部分的JVM在实现中会进行一些优化，使得两个栈帧出现一部分重叠。（主要体现在方法中有参数传递的情况），让下面栈帧的操作数栈和上面栈帧的部分局部变量重叠在一起，这样做不但节约了一部分空间，更加重要的是在进行方法调用时就可以直接公用一部分数据，无需进行额外的参数复制传递了。



使用HSDB工具查看栈空间一样可以看到。



# 虚拟机中的对象



## 对象的分配

虚拟机遇到一条new指令时，首先检查是否被类加载器加载，如果没有，那必须先执行相应的类加载过程。

类加载就是把class加载到JVM的运行时数据区的过程（类加载后面有专门的专题讲）。

### 1）检查加载

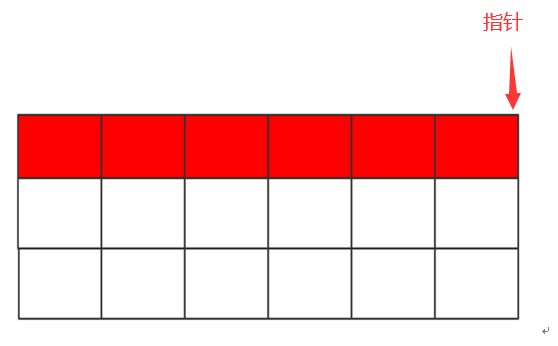
首先检查这个指令的参数是否能在常量池中定位到一个类的符号引用（**符号引用 ：**符号引用以一组符号来描述所引用的目标），并且检查类是否已经被加载、解析和初始化过。

### 2）分配内存

接下来虚拟机将为新生对象分配内存。为对象分配空间的任务等同于把一块确定大小的内存从Java堆中划分出来。

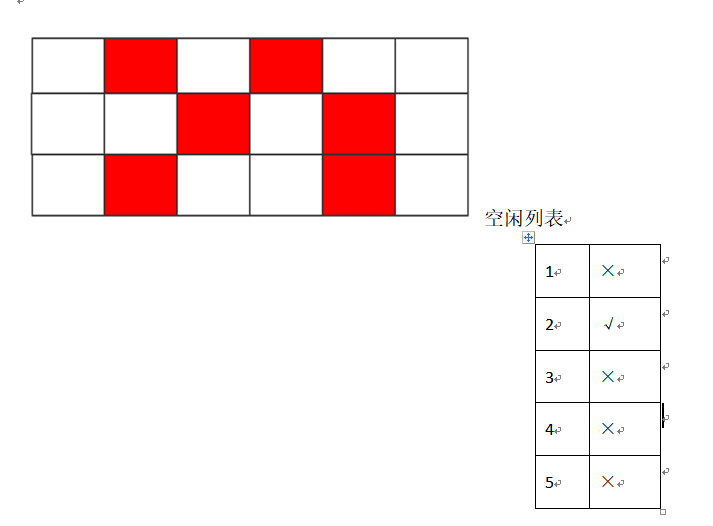
#### 指针碰撞

如果Java堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离，这种分配方式称为“**指针碰撞**”。



#### 空闲列表

如果Java堆中的内存并不是规整的，已使用的内存和空闲的内存相互交错，那就没有办法简单地进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为“**空闲列表**”。



选择哪种分配方式由Java堆是否规整决定，而Java堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

如果是Serial、ParNew等带有压缩的整理的垃圾回收器的话，系统采用的是指针碰撞，既简单又高效。

如果是使用CMS这种不带压缩（整理）的垃圾回收器的话，理论上只能采用较复杂的空闲列表。

#### 并发安全

除如何划分可用空间之外，还有另外一个需要考虑的问题是对象创建在虚拟机中是非常频繁的行为，即使是仅仅修改一个指针所指向的位置，在并发情况下也并不是线程安全的，可能出现正在给对象A分配内存，指针还没来得及修改，对象B又同时使用了原来的指针来分配内存的情况。

##### CAS机制

解决这个问题有两种方案，一种是对分配内存空间的动作进行同步处理——实际上虚拟机采用CAS配上失败重试的方式保证更新操作的原子性；

##### 分配缓冲

另一种是把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在Java堆中预先分配一小块私有内存，也就是本地线程分配缓冲（Thread Local Allocation Buffer,TLAB），JVM在线程初始化时，同时也会申请一块指定大小的内存，只给当前线程使用，这样每个线程都单独拥有一个Buffer，如果需要分配内存，就在自己的Buffer上分配，这样就不存在竞争的情况，可以大大提升分配效率，当Buffer容量不够的时候，再重新从Eden区域申请一块继续使用。

TLAB的目的是在为新对象分配内存空间时，让每个Java应用线程能在使用自己专属的分配指针来分配空间，减少同步开销。

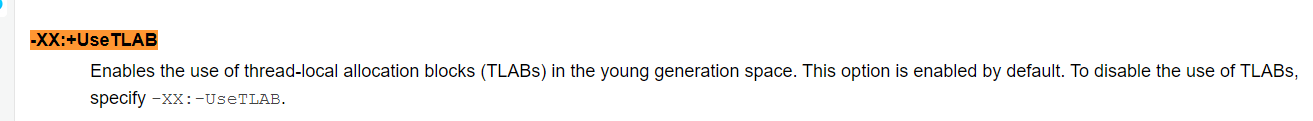
TLAB只是让每个线程有私有的分配指针，但底下存对象的内存空间还是给所有线程访问的，只是其它线程无法在这个区域分配而已。当一个TLAB用满（分配指针top撞上分配极限end了），就新申请一个TLAB。

**参数：**

-XX:+UseTLAB

允许在年轻代空间中使用线程本地分配块（TLAB）。默认情况下启用此选项。要禁用TLAB，请指定-XX:-UseTLAB。

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html>



### 3）内存空间初始化

（注意不是构造方法）内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值(如int值为0，boolean值为false等等)。这一步操作保证了对象的实例字段在Java代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

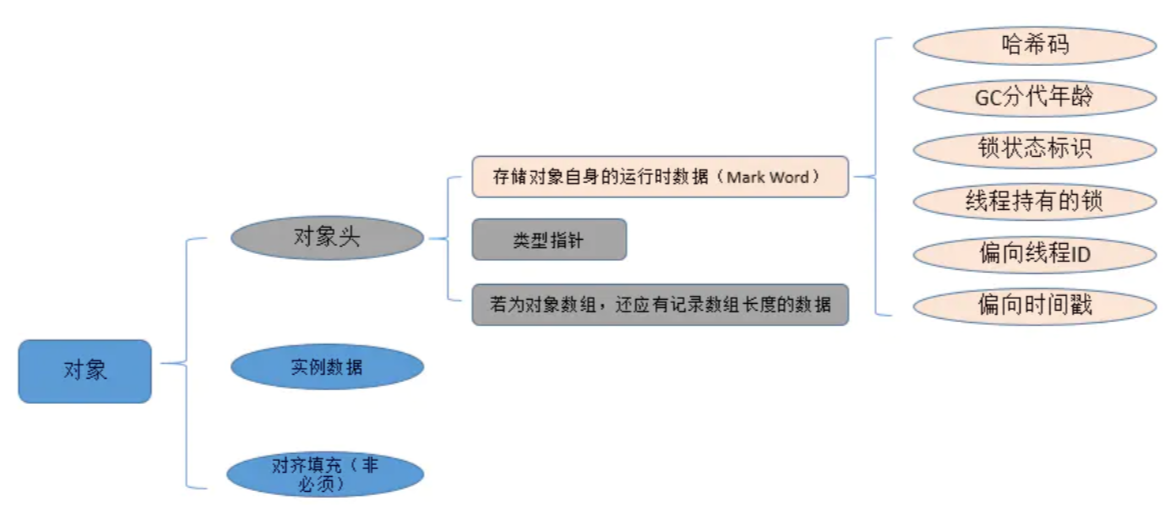
### 4）设置

接下来，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息（Java classes在Java hotspot VM内部表示为类元数据）、对象的哈希码、对象的GC分代年龄等信息。这些信息存放在对象的对象头之中。

### 5）对象初始化

在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从Java程序的视角来看，对象创建才刚刚开始，所有的字段都还为零值。所以，一般来说，执行new指令之后会接着把对象按照程序员的意愿进行初始化(构造方法)，这样一个真正可用的对象才算完全产生出来。

# 对象的内存布局



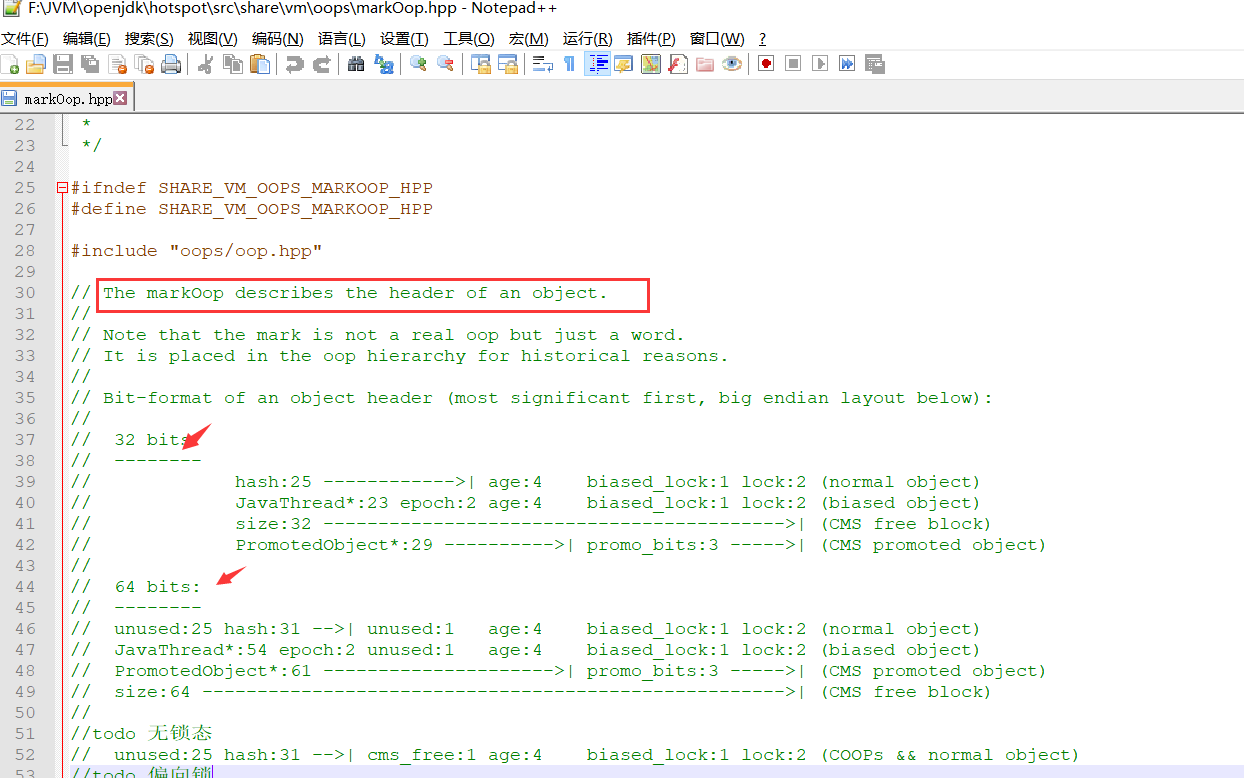
在HotSpot虚拟机中，对象在内存中存储的布局可以分为3块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

对象头包括两部分信息，第一部分用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等。

对象头的另外一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

如果对象是一个java数组，那么在对象头中还有一块用于记录数组长度的数据。

第三部分对齐填充并不是必然存在的，也没有特别的含义，它仅仅起着占位符的作用。由于HotSpot VM的自动内存管理系统要求对对象的大小必须是8字节的整数倍。当对象其他数据部分没有对齐时，就需要通过对齐填充来补全。



# 对象的访问定位

建立对象是为了使用对象，我们的Java程序需要通过栈上的reference数据来操作堆上的具体对象。目前主流的访问方式有使用句柄和直接指针两种。

## 句柄

如果使用句柄访问的话，那么Java堆中将会划分出一块内存来作为句柄池，reference中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。

## 直接指针

如果使用直接指针访问， reference中存储的直接就是对象地址。

这两种对象访问方式各有优势，使用句柄来访问的最大好处就是reference中存储的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而reference本身不需要修改。

使用直接指针访问方式的最大好处就是速度更快，它节省了一次指针定位的时间开销，由于对象的访问在Java中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本。

对Sun HotSpot而言，它是使用直接指针访问方式进行对象访问的。

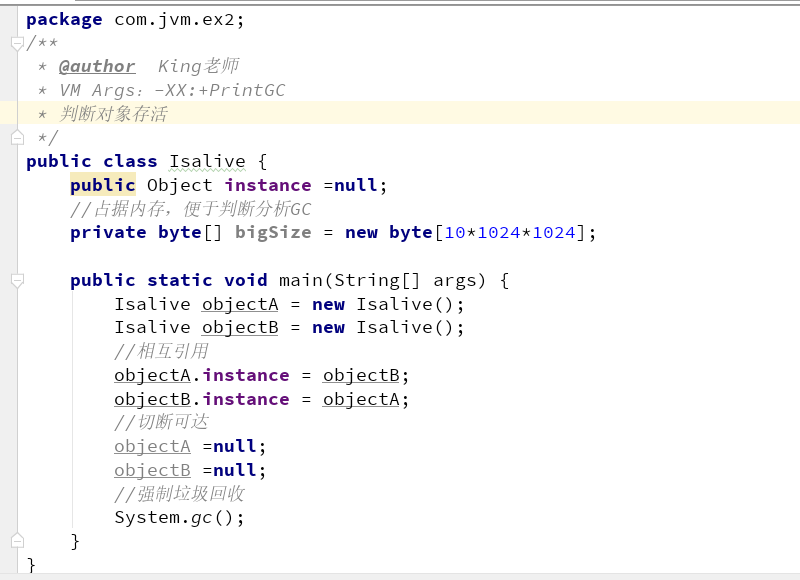
# 判断对象的存活

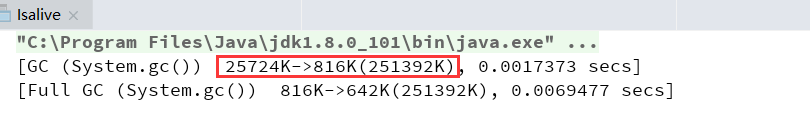
在堆里面存放着几乎所有的对象实例，垃圾回收器在对对进行回收前，要做的事情就是确定这些对象中哪些还是“存活”着，哪些已经“死去”（死去代表着不可能再被任何途径使用得对象了）

## 引用计数法

在对象中添加一个引用计数器，每当有一个地方引用它，计数器就加1，当引用失效时，计数器减1.

Python在用，但主流虚拟机没有使用，因为存在对象相互引用的情况，这个时候需要引入额外的机制来处理，这样做影响效率，





在代码中看到，只保留相互引用的对象还是被回收掉了，说明JVM中采用的不是引用计数法。

## 可达性分析

（**面试时重要的知识点，牢记**）

来判定对象是否存活的。这个算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。

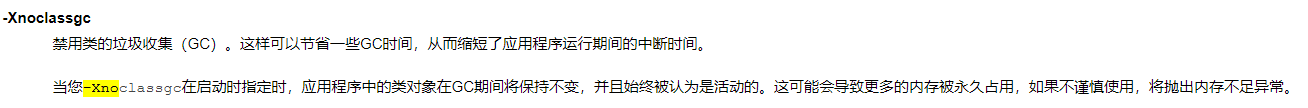
作为GC Roots的对象包括下面几种：

* 虚拟机栈（栈帧中的本地变量表）中引用的对象。
* 方法区中类静态属性引用的对象。
* 方法区中常量引用的对象。
* 本地方法栈中JNI（即一般说的Native方法）引用的对象。
* JVM的内部引用（class对象、异常对象NullPointException、OutofMemoryError，系统类加载器）。
* 所有被同步锁(synchronized关键)持有的对象。
* JVM内部的JMXBean、JVMTI中注册的回调、本地代码缓存等
* JVM实现中的“临时性”对象，跨代引用的对象（在使用分代模型回收只回收部分代时）

以上的回收都是对象，类的回收条件：

**注意Class要被回收，条件比较苛刻，必须同时满足以下的条件（仅仅是可以，不代表必然，因为还有一些参数可以进行控制）：**

1. **该类所有的实例都已经被回收，也就是堆中不存在该类的任何实例。**
2. 加载该类的ClassLoader已经被回收。
3. 该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。
4. 参数控制：



还有一个废弃的常量，这个是对象的回收非常相似，比如：假如有一个字符串“king”进入常量池。

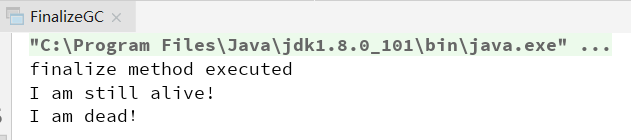
## Finalize方法

即使通过可达性分析判断不可达的对象，也不是“非死不可”，它还会处于“缓刑”阶段，真正要宣告一个对象死亡，需要经过两次标记过程，一次是没有找到与GCRoots的引用链，它将被第一次标记。随后进行一次筛选（如果对象覆盖了finalize），我们可以在finalize中去拯救。

**代码演示：**

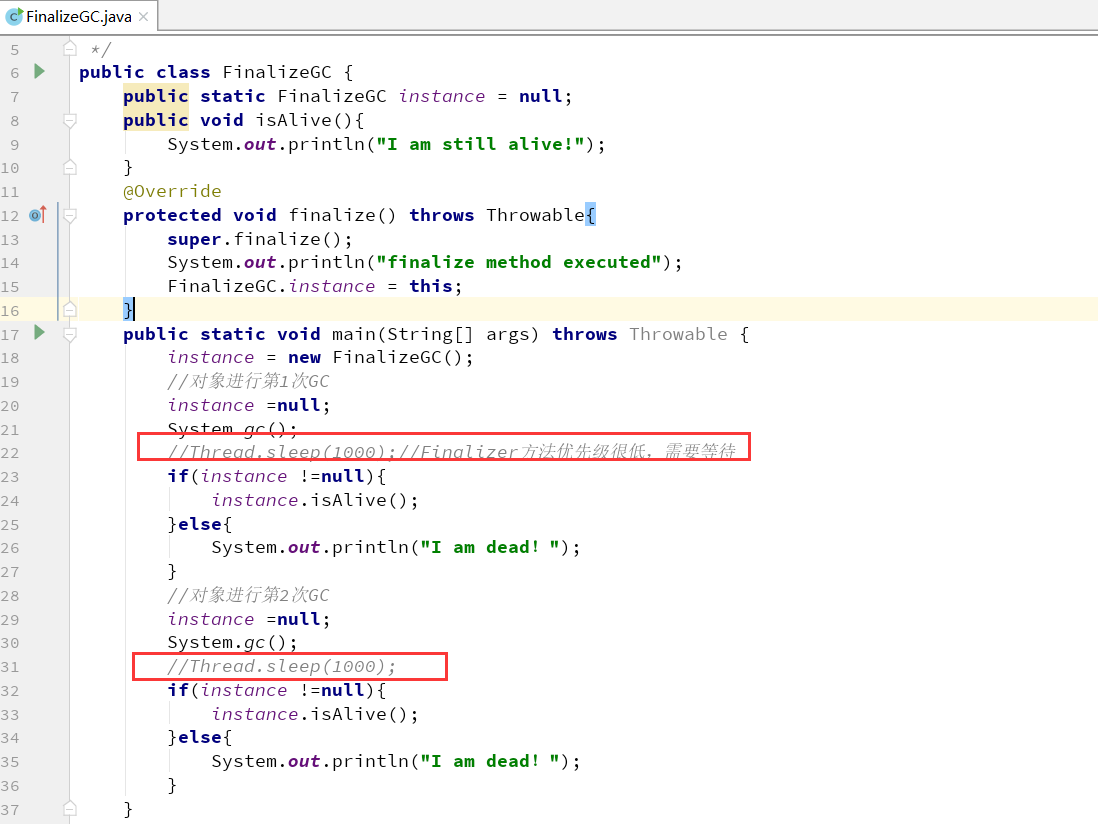
****

**运行结果：**

****

**可以看到，对象可以被拯救一次(finalize执行第一次，但是不会执行第二次)**

**代码改一下，再来一次。**

****

**运行结果：**

****

**对象没有被拯救，这个就是finalize方法执行缓慢，还没有完成拯救，垃圾回收器就已经回收掉了。**

**所以建议大家尽量不要使用finalize，因为这个方法太不可靠。在生产中你很难控制方法的执行或者对象的调用顺序，建议大家忘了finalize方法！因为在finalize方法能做的工作，java中有更好的，比如try-finally或者其他方式可以做得更好**

# 各种引用

## 强引用

一般的Object obj = new Object() ，就属于强引用。在任何情况下，只有有强引用关联（与根可达）还在，垃圾回收器就永远不会回收掉被引用的对象。

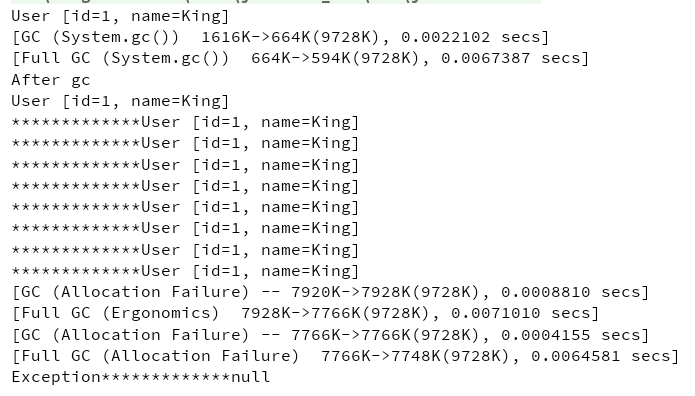
## 软引用 SoftReference

一些有用但是并非必需，用软引用关联的对象，系统将要发生内存溢出（OuyOfMemory）之前，这些对象就会被回收（如果这次回收后还是没有足够的空间，才会抛出内存溢出）。参见代码：

VM参数 -Xms10m -Xmx10m -XX:+PrintGC



运行结果

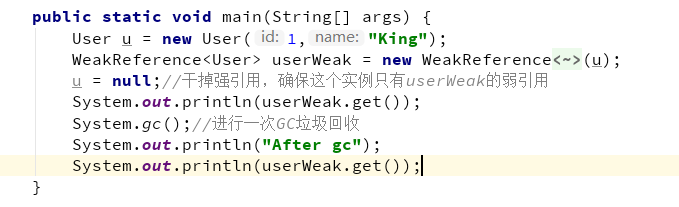


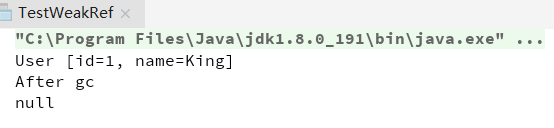
例如，一个程序用来处理用户提供的图片。如果将所有图片读入内存，这样虽然可以很快的打开图片，但内存空间使用巨大，一些使用较少的图片浪费内存空间，需要手动从内存中移除。如果每次打开图片都从磁盘文件中读取到内存再显示出来，虽然内存占用较少，但一些经常使用的图片每次打开都要访问磁盘，代价巨大。这个时候就可以用软引用构建缓存。

## 弱引用 WeakReference

一些有用（程度比软引用更低）但是并非必需，用弱引用关联的对象，只能生存到下一次垃圾回收之前，GC发生时，不管内存够不够，都会被回收。

参看代码：





**注意：**软引用 SoftReference和弱引用 WeakReference，可以用在内存资源紧张的情况下以及创建不是很重要的数据缓存。当系统内存不足的时候，缓存中的内容是可以被释放的。

实际运用（WeakHashMap、ThreadLocal）

## 虚引用 PhantomReference

幽灵引用，最弱（随时会被回收掉）

垃圾回收的时候收到一个通知，就是为了监控垃圾回收器是否正常工作。

# 学习垃圾回收的意义

Java与C++等语言最大的技术区别：自动化的垃圾回收机制（GC）

为什么要了解GC和内存分配策略

1、面试需要

2、GC对应用的性能是有影响的；

3、写代码有好处

栈：栈中的生命周期是跟随线程，所以一般不需要关注

堆：堆中的对象是垃圾回收的重点

方法区/元空间：这一块也会发生垃圾回收，不过这块的效率比较低，一般不是我们关注的重点

# 对象的分配策略

## 栈上分配

### 没有逃逸

即方法中的对象没有发生逃逸。

即方法中的对象没有发生逃逸。

逃逸分析的原理：分析对象动态作用域，当一个对象在方法中定义后，它可能被外部方法所引用，比如：调用参数传递到其他方法中，这种称之为方法逃逸，甚至还有可能被外部线程访问到，例如：赋值给其他线程中访问的变量，这个称之为线程逃逸。

从不逃逸到方法逃逸到线程逃逸，称之为对象由低到高的不同逃逸程度。

如果确定一个对象不会逃逸出线程之外，那么让对象在栈上分配内存可以提高JVM的效率。

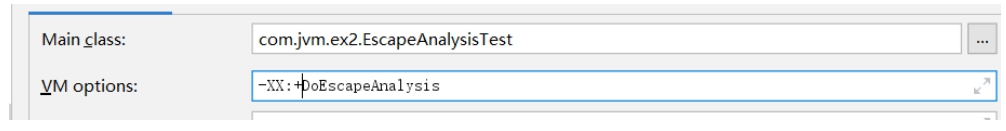
### 逃逸分析代码

public class EscapeAnalysisTest {  
 public static void main(String[] args) throws Exception {  
 long start = System.currentTimeMillis();  
 for (int i = 0; i < 50000000; i++) {  
 allocate();  
 }  
 System.out.println((System.currentTimeMillis() - start) + " ms");  
 Thread.sleep(600000);  
 }  
​  
 static void allocate() {  
 MyObject myObject = new MyObject(2020, 2020.6);  
 }  
​  
 static class MyObject {  
 int a;  
 double b;  
​  
 MyObject(int a, double b) {  
 this.a = a;  
 this.b = b;  
 }  
 }  
}

这段代码在调用的过程中 myboject这个对象属于全局逃逸，JVM可以做栈上分配

然后通过开启和关闭DoEscapeAnalysis开关观察不同。

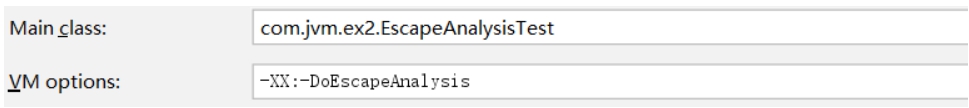
开启逃逸分析（JVM默认开启）



查看执行速度



关闭逃逸分析



查看执行速度

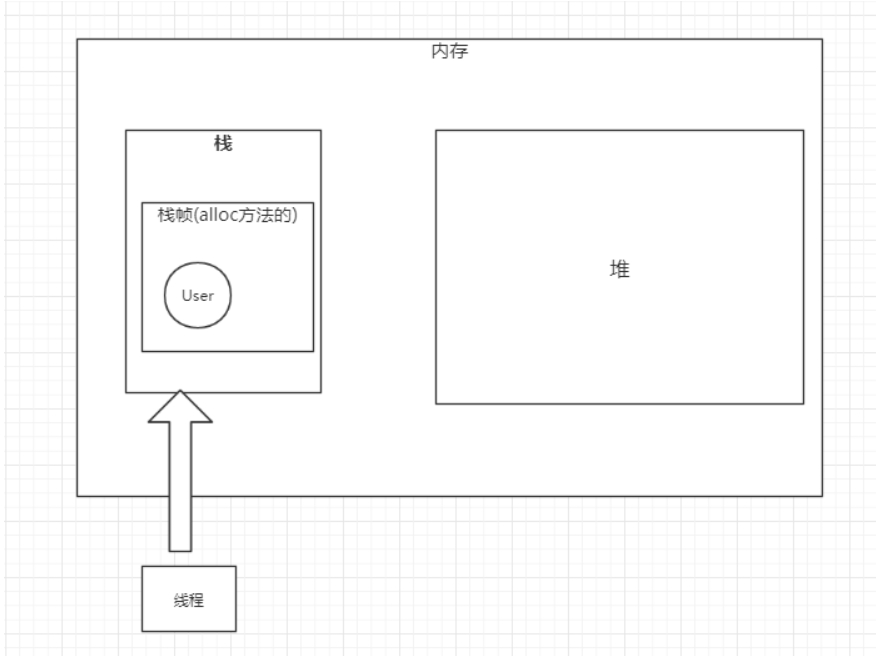


测试结果可见，开启逃逸分析对代码的执行性能有很大的影响！那为什么有这个影响？

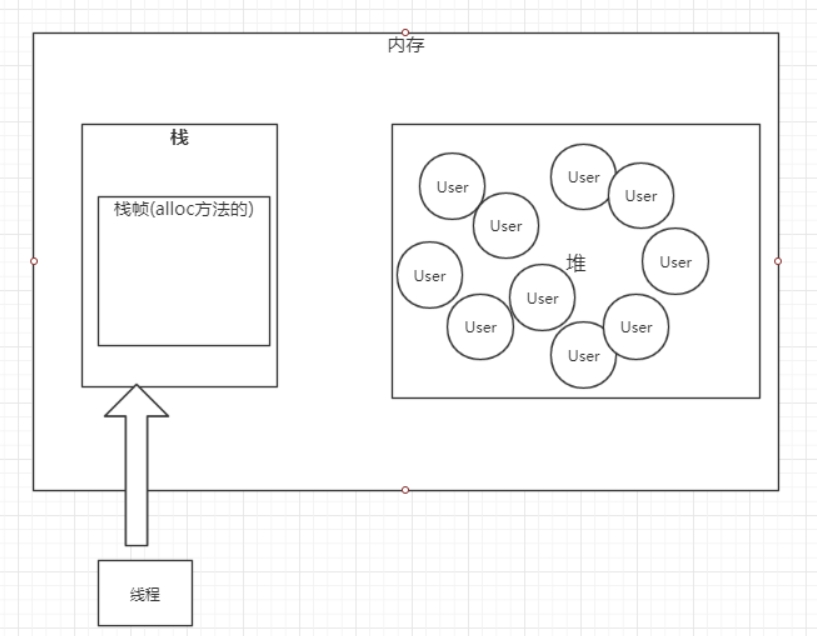
### 逃逸分析

如果是逃逸分析出来的对象可以在栈上分配的话，那么该对象的生命周期就跟随线程了，就不需要垃圾回收，如果是频繁的调用此方法则可以得到很大的性能提高。

采用了逃逸分析后，满足逃逸的对象在栈上分配



没有开启逃逸分析，对象都在堆上分配，会频繁触发垃圾回收（垃圾回收会影响系统性能），导致代码运行慢



**代码验证**

开启GC打印日志

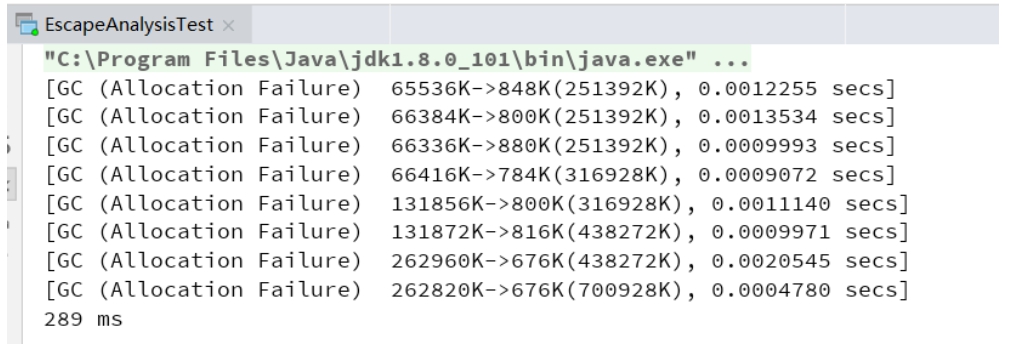
-XX:+PrintGC

**开启逃逸分析**



可以看到没有GC日志

关闭逃逸分析



可以看到关闭了逃逸分析，JVM在频繁的进行垃圾回收（GC），正是这一块的操作导致性能有较大的差别。

## 对象优先在Eden区分配

虚拟机参数：

-Xms20m

-Xmx20m

-Xmn10m

-XX:+PrintGCDetails

-XX:+PrintGCDetails 打印垃圾回收日志，程序退出时输出当前内存的分配情况

注意：新生代初始时就有大小

大多数情况下，对象在新生代Eden区中分配。当Eden区没有足够空间分配时，虚拟机将发起一次Minor GC。

## 大对象直接进入老年代

-Xms20m

-Xmx20m

-Xmn10m

-XX:+PrintGCDetails

-XX:PretenureSizeThreshold=4m

-XX:+UseSerialGC

PretenureSizeThreshold参数只对Serial和ParNew两款收集器有效。

最典型的大对象是那种很长的字符串以及数组。这样做的目的：1.避免大量内存复制,2.避免提前进行垃圾回收，明明内存有空间进行分配。

## 长期存活对象进入老年区

如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，将被移动到Survivor空间中，并将对象年龄设为1，对象在Survivor区中每熬过一次 Minor GC，年龄就增加1，当它的年龄增加到一定程度(并发的垃圾回收器默认为15),CMS是6时，就会被晋升到老年代中。

-XX:MaxTenuringThreshold调整

## 对象年龄动态判定

为了能更好地适应不同程序的内存状况，虚拟机并不是永远地要求对象的年龄必须达到了MaxTenuringThreshold才能晋升老年代，如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到MaxTenuringThreshold中要求的年龄

## 空间分配担保

在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么Minor GC可以确保是安全的。如果不成立，则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次Minor GC，尽管这次Minor GC是有风险的，如果担保失败则会进行一次Full GC；如果小于，或者HandlePromotionFailure设置不允许冒险，那这时也要改为进行一次Full GC。

### 本地线程分配缓冲(TLAB)

具体见章节[分配缓冲](#_分配缓冲)

# 垃圾回收算法

垃圾回收算法的实现设计到大量的程序细节，并且每一个平台的虚拟机操作内存的方式都有不同，所以不需要去了解算法的实现，我们重点讲解分代收集理论和3种算法的思想。

## 分代收集理论

当前商业虚拟机的垃圾收集器，大多遵循“分代收集”的理论来进行设计，这个理论大体上是这么描述的：

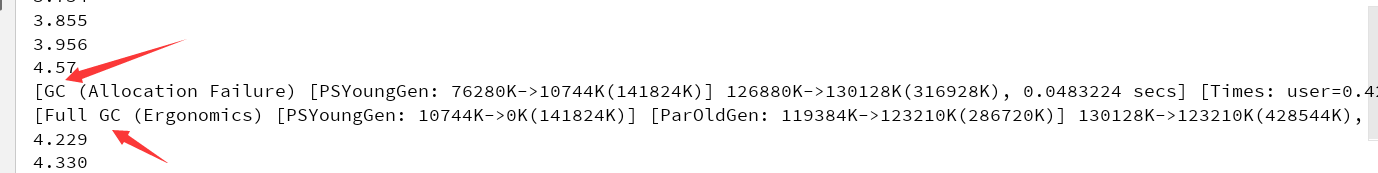
1. 绝大部分的对象都是朝生夕死
2. 熬过多次垃圾回收的对象就越难回收。

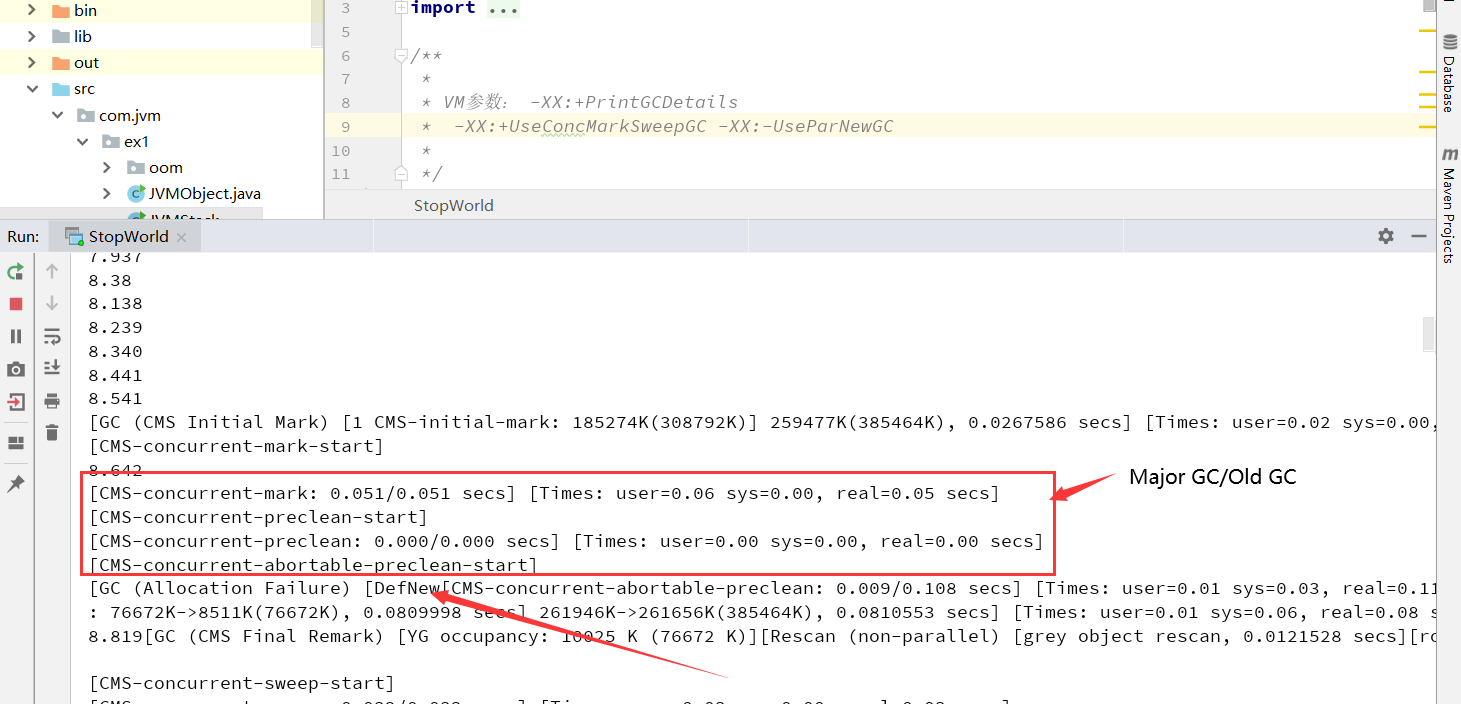
根据以上两个理论，朝生夕死的对象放一个区域，难回收的对象放另外一个区域，这个就构成了新生代和老年代。

### GC种类

市面上发生垃圾回收的叫法很多，我大体整理了一下：

1. 新生代回收（Minor GC/Young GC）:指只是进行新生代的回收。
2. 老年代回收（Major GC/Old GC）:指只是进行老年代的回收。目前只有CMS垃圾回收器会有这个单独的收集老年代的行为。（Major GC定义是比较混乱，有说指是老年代，有的说是做整个堆的收集，这个需要你根据别人的场景来定，没有固定的说法）
3. 整堆收集（Full GC）:收集整个Java堆和方法区(注意包含方法区)





## 复制算法（Copying）

将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为了原来的一半。

注意：内存移动是必须实打实的移动（复制），不能使用指针玩。

复制回收算法适合于新生代，因为大部分对象朝生夕死，那么复制过去的对象比较少，效率自然就高，另外一半的一次性清理是很快的。

### Appel式回收

一种更加优化的复制回收分代策略：具体做法是分配一块较大的Eden区和两块较小的Survivor空间（你可以叫做From或者To，也可以叫做Survivor1和Survivor2）

专门研究表明，新生代中的对象98%是“朝生夕死”的，所以并不需要按照1:1的比例来划分内存空间，而是将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor[1]。当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。

HotSpot虚拟机默认Eden和Survivor的大小比例是8:1，也就是每次新生代中可用内存空间为整个新生代容量的90%（80%+10%），只有10%的内存会被“浪费”。当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于10%的对象存活，当Survivor空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）

## 标记-清除算法（Mark-Sweep）

算法分为“标记”和“清除”两个阶段：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。

回收效率不稳定，如果大部分对象是朝生夕死，那么回收效率降低，因为需要大量标记对象和回收对象，对比复制回收效率很低。

它的主要不足空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

回收的时候如果需要回收的对象越多，需要做的标记和清除的工作越多，所以标记清除算法适用于老年代。复制回收算法适用于新生代。

## 标记-整理算法（Mark-Compact）

首先标记出所有需要回收的对象，在标记完成后，后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。标记整理算法虽然没有内存碎片，但是效率偏低。

我们看到标记整理与标记清除算法的区别主要在于对象的移动。对象移动不单单会加重系统负担，同时需要全程暂停用户线程才能进行，同时所有引用对象的地方都需要更新。

所以看到，老年代采用的标记整理算法与标记清除算法，各有优点，各有缺点。

# JVM中常见的垃圾收集器

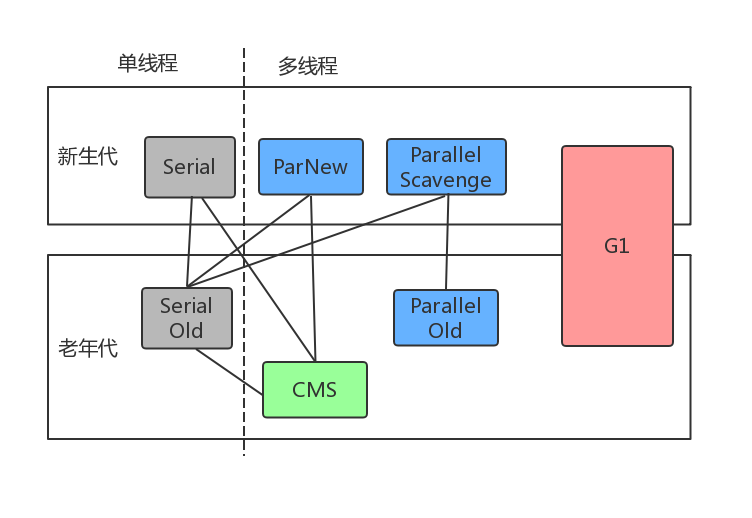
## 分代收集的思想

在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。

而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记—清理”或者“标记—整理”算法来进行回收。

**请记住下图的垃圾收集器和之间的连线关系。**

**具体看官网JVM参数：**<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html>



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **收集器** | **收集对象和算法** | **收集器类型** | **说明** | **适用场景** |
| Serial | 新生代，复制算法 | 单线程 |  | 简单高效；  适合内存不大的情况； |
| ParNew | 新生代，复制算法 | 并行的多线程收集器 | ParNew垃圾收集器是Serial收集器的多线程版本 | 搭配CMS垃圾回收器的首选 |
| Parallel Scavenge  吞吐量优先收集器 | 新生代，复制算法 | 并行的多线程收集器 | 类似ParNew，更加关注吞吐量，达到一个可控制的吞吐量； | 本身是Server级别多CPU机器上的默认GC方式，主要适合后台运算不需要太多交互的任务； |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **收集器** | **收集对象和算法** | **收集器类型** | **说明** | **适用场景** |
| Serial Old | 老年代，标记整理算法 | 单线程 |  | Client模式下虚拟机使用 |
| Parallel Old | 老年代，标记整理算法 | 并行的多线程收集器 | Parallel Scavenge收集器的老年代版本，为了配合Parallel Scavenge的面向吞吐量的特性而开发的对应组合； | 在注重吞吐量以及CPU资源敏感的场合采用 |
| CMS | 老年代，标记清除算法 | 并行与并发收集器 | 尽可能的缩短垃圾收集时用户线程停止时间；缺点在于：  1.内存碎片  2.需要更多cpu资源  3.浮动垃圾问题，需要更大的堆空间 | 重视服务的响应速度、系统停顿时间和用户体验的互联网网站或者B/S系统。互联网后端目前cms是主流的垃圾回收器； |
| G1 | 跨新生代和老年代；标记整理 + 化整为零 | 并行与并发收集器 | JDK1.7才正式引入，采用分区回收的思维，基本不牺牲吞吐量的前提下完成低停顿的内存回收；可预测的停顿是其最大的优势； | 面向服务端应用的垃圾回收器，目标为取代CMS |

并行：垃圾收集的多线程的同时进行。

并发：垃圾收集的多线程和应用的多线程同时进行。

注：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)

垃圾收集时间= 垃圾回收频率 \* 单次垃圾回收时间

# 垃圾回收器工作示意图

## Serial/Serial Old

最古老的，单线程，独占式，成熟，适合单CPU 服务器

-XX:+UseSerialGC 新生代和老年代都用串行收集器

-XX:+UseParNewGC 新生代使用ParNew，老年代使用Serial Old

-XX:+UseParallelGC 新生代使用ParallerGC，老年代使用Serial Old

## ParNew

和Serial基本没区别，唯一的区别：多线程，多CPU的，停顿时间比Serial少

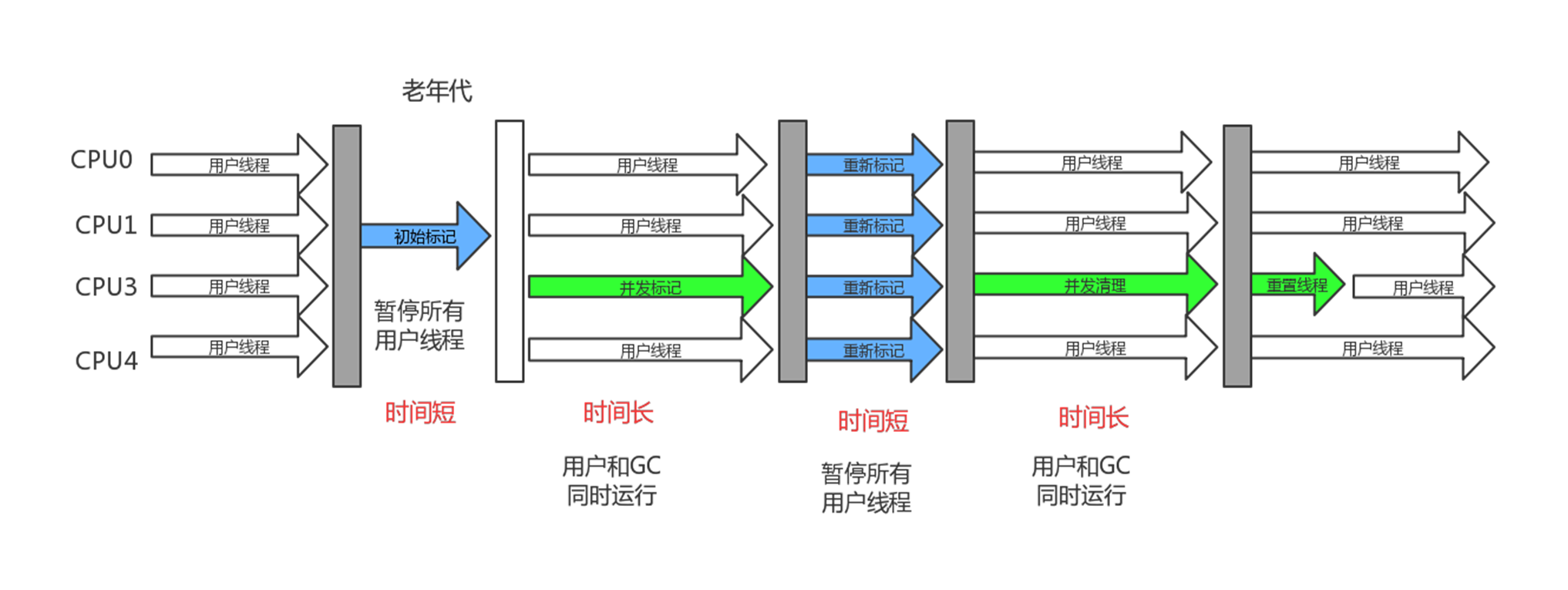
-XX:+UseParNewGC 新生代使用ParNew，老年代使用Serial Old

## Parallel Scavenge（ParallerGC）/Parallel Old

关注吞吐量的垃圾收集器，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。

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%。

## Concurrent Mark Sweep （CMS）



收集器是一种以获取最短回收停顿时间为目标的收集器。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。CMS收集器就非常符合这类应用的需求。

从名字（包含“Mark Sweep”）上就可以看出，CMS收集器是基于“标记—清除”算法实现的，它的运作过程相对于前面几种收集器来说更复杂一些，整个过程分为4个步骤，包括：

* **初始标记**-短暂，仅仅只是标记一下GC Roots能直接关联到的对象，速度很快。
* **并发标记**-和用户的应用程序同时进行，进行GC Roots追踪的过程，标记从GCRoots开始关联的所有对象开始遍历整个可达分析路径的对象。这个时间比较长，所以采用并发处理（垃圾回收器线程和用户线程同时工作）
* **重新标记**-短暂，为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。
* **并发清除**

由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

-XX:+UseConcMarkSweepGC ，表示新生代使用ParNew，老年代的用CMS

**CPU敏感：**CMS对处理器资源敏感，毕竟采用了并发的收集、当处理核心数不足4个时，CMS对用户的影响较大。

**浮动垃圾：**由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”。

由于浮动垃圾的存在，因此需要预留出一部分内存，意味着 CMS 收集不能像其它收集器那样等待老年代快满的时候再回收。

在1.6的版本中老年代空间使用率阈值(92%)

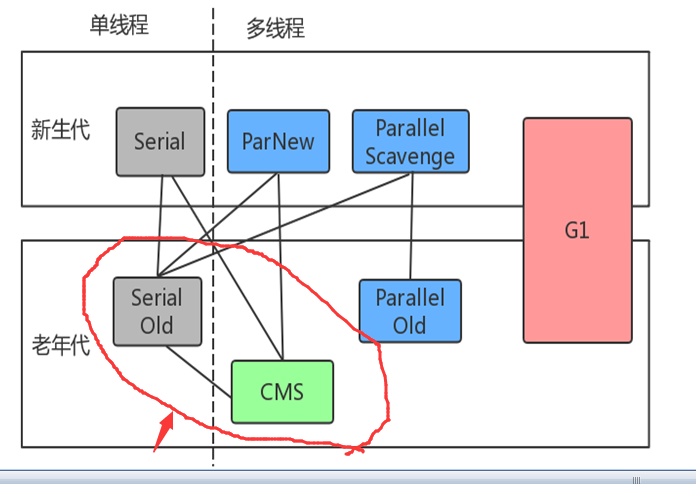
如果预留的内存不够存放浮动垃圾，就会出现 Concurrent Mode Failure，这时虚拟机将临时启用 Serial Old 来替代 CMS。

**会产生空间碎片：**标记 - 清除算法会导致产生不连续的空间碎片

总体来说，CMS是JVM推出了第一款并发垃圾收集器，所以还是非常有代表性。

但是最大的问题是CMS采用了标记清除算法，所以会有内存碎片，当碎片较多时，给大对象的分配带来很大的麻烦，为了解决这个问题，CMS提供一个参数：-XX:+UseCMSCompactAtFullCollection，一般是开启的，如果分配不了大对象，就进行内存碎片的整理过程。

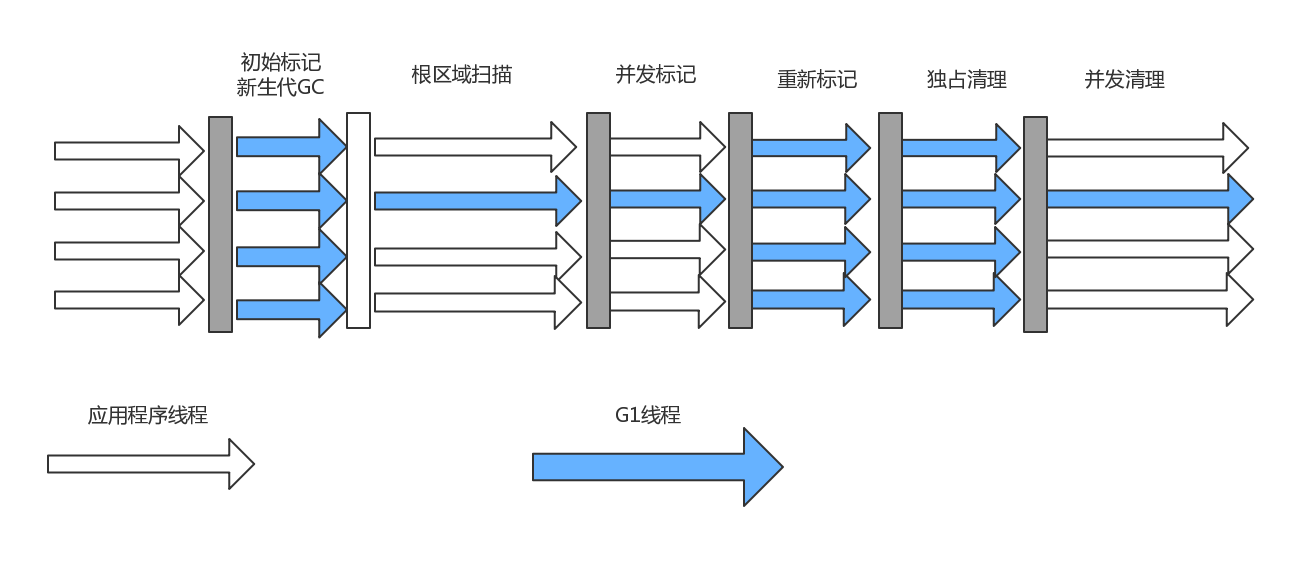
这个地方一般会使用Serial Old ，因为Serial Old是一个单线程，所以如果内存空间很大、且对象较多时,CMS发生这样情况会很卡。



# Stop The World现象

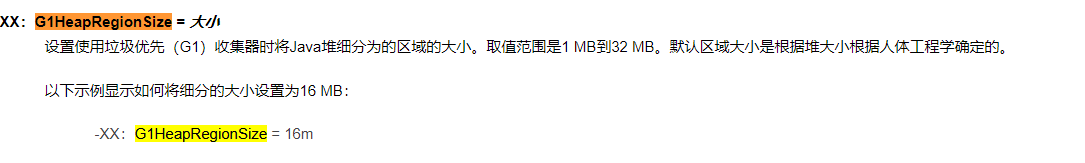
任何的GC收集器都会进行业务线程的暂停，这个就是STW，Stop The World,所以我们GC调优的目标就是尽可能的减少STW的时间和次数。

## G1



-XX:+UseG1GC

**内存布局：**在G1之前的其他收集器进行收集的范围都是整个新生代或者老年代，而G1不再是这样。使用G1收集器时，Java堆的内存布局就与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分Region（不需要连续）的集合。每一个区域可以通过参数-XX:G1HeapRegionSize=size 来设置。

****

Region中还有一块特殊区域Humongous区域，专门用于存储大对象，一般只要认为一个对象超过了Region容量的一般可认为是大对象，如果对象超级大，那么使用连续的N个Humongous区域来存储。

**并行与并发**：G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短Stop-The-World停顿的时间，部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让Java程序继续执行。

**分代收集**：与其他收集器一样，分代概念在G1中依然得以保留。虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但它能够采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次GC的旧对象以获取更好的收集效果。

**空间整合**：与CMS的“标记—清理”算法不同，G1从整体来看是基于“标记—整理”算法实现的收集器，从局部（两个Region之间）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次GC。

**追求停顿时间：**

-XX:MaxGCPauseMillis 指定目标的最大停顿时间，G1尝试调整新生代和老年代的比例，堆大小，晋升年龄来达到这个目标时间。

-XX:ParallerGCThreads：设置GC的工作线程数量。

一般在G1和CMS中间选择的话平衡点在6~8G，只有内存比较大G1才能发挥优势。

# 常量池与String

常量池有很多概念，包括运行时常量池、class常量池、字符串常量池。

虚拟机规范只规定以上区域属于方法区，并没有规定虚拟机厂商的实现。

严格来说是静态常量池和运行时常量池，静态常量池是存放字符串字面量、符号引用以及类和方法的信息，而运行时常量池存放的是运行时一些直接引用。  
运行时常量池是在类加载完成之后，将静态常量池中的符号引用值转存到运行时常量池中，类在解析之后，将符号引用替换成直接引用。  
这两个常量池在JDK1.7版本之后，就移到堆内存中了，这里指的是物理空间，而逻辑上还是属于方法区（方法区是逻辑分区）。

**字面量:**

给基本类型变量赋值的方式就叫做字面量或者字面值

比如：int i=120; long j=10L;

**符号引用**：包括类和方法的全限定名（例如 String 这个类，它的全限定名就是 Java/lang/String）、字段的名称和描述符以及方法的名称和描述符。

**直接引用：**具体对象的索引值。

## String 对象是如何实现的？

了解了 String 对象的实现后，你有没有发现在实现代码中 String 类被 final 关键字修饰了，而且变量 char 数组也被 final 修饰了。我们知道类被 final 修饰代表该类不可继承，而 char[]被 final+private 修饰，代表了 String 对象不可被更改。Java 实现的这个特性叫作 String 对象的不可变性，即 String 对象一旦创建成功，就不能再对它进行改变。

在 Java 中，通常有两种创建字符串对象的方式，

一种是通过字符串常量的方式创建，如 String str=“abc”；

这种方式首先会检查该对象是否在字符串常量池中，如果在，就返回该对象引用，否则新的字符串将在常量池中被创建。这种方式可以减少同一个值的字符串对象的重复创建，节约内存。

另一种是字符串变量通过 new 形式的创建，如 String str = new String(“abc”)。

这种方式，首先在编译类文件时，"abc"常量字符串将会放入到常量结构中，在类加载时，“abc"将会在常量池中创建；其次，在调用 new 时，JVM 命令将会调用 String 的构造函数，同时引用常量池中的"abc” 字符串，在堆内存中创建一个 String 对象；最后，str 将引用 String 对象。

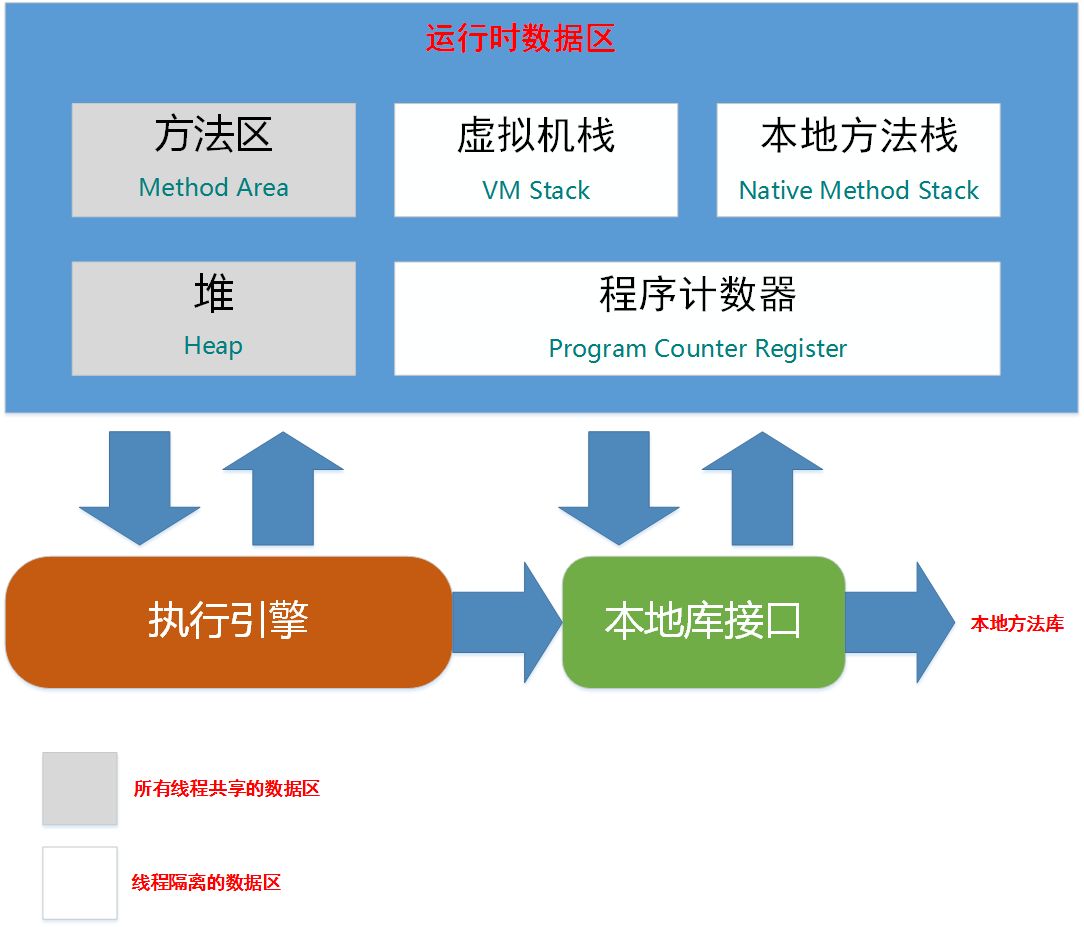
如果调用 intern 方法，会去查看字符串常量池中是否有等于该对象的字符串的引用，如果没有会把首次遇到的字符串的引用添加到常量池中；如果有，就返回常量池中的字符串引用。（这个版本都是基于JDK1.7及以后版本）

# 面试常见问题剖析

## JVM内存结构说一下！

开放式题目，具体可见章节 [运行时数据区域](#_运行时数据区域)

一般从两个维度出发：线程私有和线程共享。到每一个内存区域的细节点。



Java 虚拟机栈是基于线程的。哪怕你只有一个 main() 方法，也是以线程的方式运行的。在线程的生命周期中，参与计算的数据会频繁地入栈和出栈，栈的生命周期是和线程一样的。

栈里的每条数据，就是栈帧。在每个 Java 方法被调用的时候，都会创建一个栈帧，并入栈。一旦完成相应的调用，则出栈。所有的栈帧都出栈后，线程也就结束了。每个栈帧，都包含四个区域：

局部变量表

操作数栈

动态连接

返回地址

本地方法栈是和虚拟机栈非常相似的一个区域，它服务的对象是 native 方法。

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。

堆是 JVM 上最大的内存区域，我们申请的几乎所有的对象，都是在这里存储的。我们常说的垃圾回收，操作的对象就是堆。

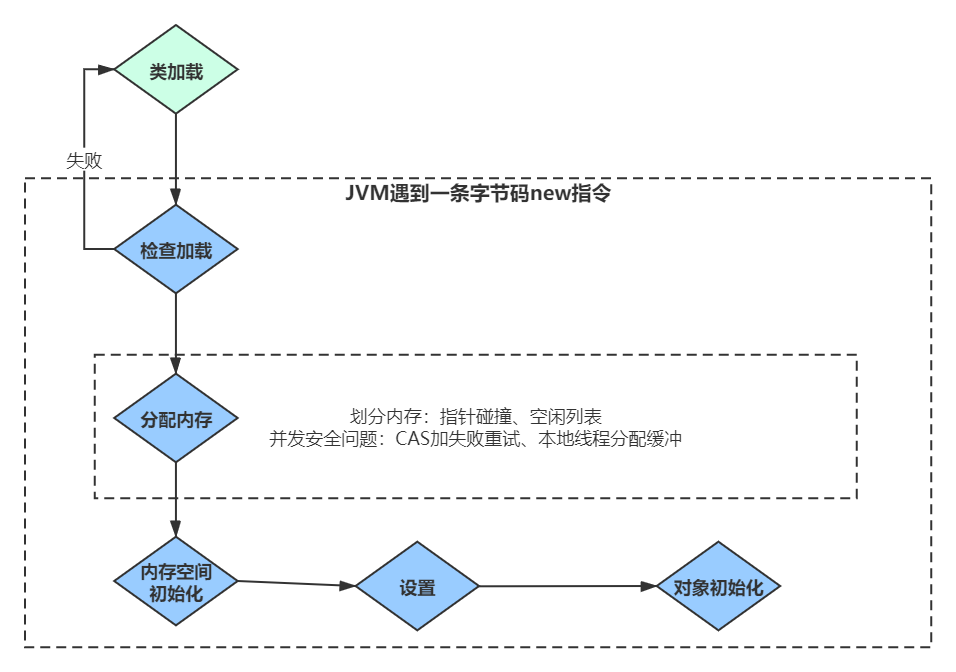
方法区，这个区域存储的内容，包括：类的信息、常量池、方法数据、方法代码就可以了。

## 什么情况下内存栈溢出？

java.lang.StackOverflowError 如果出现了可能会是无限递归。

OutOfMemoryError：不断建立线程，JVM申请栈内存，机器没有足够的内存。

## 描述new一个对象的流程！



具体见章节[对象的分配](#_对象的分配)

## Java对象会不会分配在栈中？

可以，如果这个对象不满足逃逸分析，那么虚拟机在特定的情况下会走栈上分配。

## 如果判断一个对象是否被回收，有哪些算法，实际虚拟机使用得最多的是什么？

引用计数法和根可达性分析两种，用得最多是根可达性分析。

## GC收集算法有哪些？他们的特点是什么？

复制、标记清除、标记整理。复制速度快，但是要浪费空间，不会内存碎片。标记清除空间利用率高，但是有内存碎片。标记整理算法没有内存碎片，但是要移动对象，性能较低。三种算法各有所长，各有所短。

## JVM中一次完整的GC流程是怎样的？对象如何晋级到老年代？

对象优先在新生代区中分配，若没有足够空间，Minor GC；  
大对象（需要大量连续内存空间）直接进入老年态；长期存活的对象进入老年态。

如果对象在新生代出生并经过第一次MGC后仍然存活，年龄+1，若年龄超过一定限制（15），则被晋升到老年态。

## Java中的几种引用关系，他们的区别是什么？

**强引用**

一般的Object obj = new Object() ，就属于强引用。在任何情况下，只有有强引用关联（与根可达）还在，垃圾回收器就永远不会回收掉被引用的对象。

**软引用 SoftReference**

一些有用但是并非必需，用软引用关联的对象，系统将要发生内存溢出（OuyOfMemory）之前，这些对象就会被回收（如果这次回收后还是没有足够的空间，才会抛出内存溢出）。

**弱引用 WeakReference**

一些有用（程度比软引用更低）但是并非必需，用弱引用关联的对象，只能生存到下一次垃圾回收之前，GC发生时，不管内存够不够，都会被回收。

**虚引用 PhantomReference**

幽灵引用，最弱（随时会被回收掉）

垃圾回收的时候收到一个通知，就是为了监控垃圾回收器是否正常工作。

## final、finally、finalize的区别？

在java中，final可以用来修饰类，方法和变量（成员变量或局部变量）

当用final修饰类的时，表明该类不能被其他类所继承。当我们需要让一个类永远不被继承，此时就可以用final修饰，但要注意：

final类中所有的成员方法都会隐式的定义为final方法。

使用final方法的原因主要有两个：

　　(1) 把方法锁定，以防止继承类对其进行更改。

　　(2) 效率，在早期的java版本中，会将final方法转为内嵌调用。但若方法过于庞大，可能在性能上不会有多大提升。因此在最近版本中，不需要final方法进行这些优化了。

final成员变量表示常量，只能被赋值一次，赋值后其值不再改变。

**finally作为异常处理的一部分**，它只能用在try/catch语句中，并且附带一个语句块，表示这段语句最终一定会被执行（不管有没有抛出异常），经常被用在需要释放资源的情况下

**Object中的Finalize方法**

即使通过可达性分析判断不可达的对象，也不是“非死不可”，它还会处于“缓刑”阶段，真正要宣告一个对象死亡，需要经过两次标记过程，一次是没有找到与GCRoots的引用链，它将被第一次标记。随后进行一次筛选（如果对象覆盖了finalize），我们可以在finalize中去拯救。

所以建议大家尽量不要使用finalize，因为这个方法太不可靠。在生产中你很难控制方法的执行或者对象的调用顺序，建议大家忘了finalize方法！因为在finalize方法能做的工作，java中有更好的，比如try-finally或者其他方式可以做得更好

## String s = new String(“xxx”);创建了几个对象？

2个，

1. 在一开始字符串"xxx"会在加载类时，在常量池中创建一个字符串对象。
2. 调用 new时 会在堆内存中创建一个 String 对象，String 对象中的 char 数组将会引用常量池中字符串。