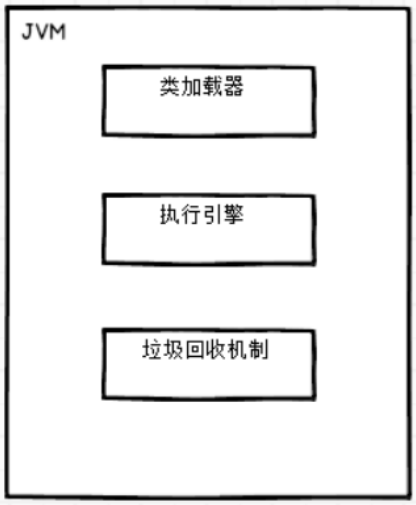
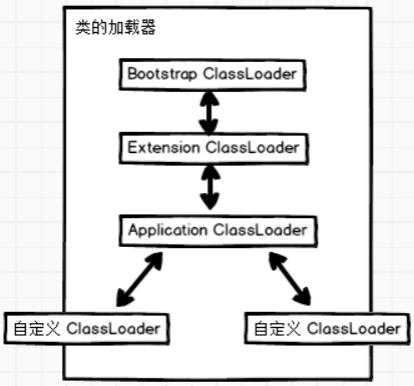


JVM理解
我们可以把JVM看成一台虚拟的机器，这台机器可以按需加载可执行二进制文件（字节码文件），然后由虚拟机执行引擎解释执行字节码，将其翻译成cpu可以识别的指令。在jvm的逻辑地址空间中，有方法区（用来存放可执行文件），堆（用于存放对象和数组，jvm垃圾回收器动态分配和回收该区域的内存空间），栈（保存线程的方法调用关系，数据），常量池（存放常量）等。因此，一个Java虚拟机实例在运行过程中有三个子系统来保障它的正常运行，分别是类加载器， 执行引擎和垃圾回收机制。
执行引擎包括字节码解释器和JIT（just-in-time）及时编译器，解释器将字节码文件一行一行边解释边执行（解释成cpu能识别的指令），而JIT编译器则是将整个字节码文件编译成cpu能够识别的指令，也就是在执行前全部被翻译为机器码，这样做的好处是将热点代码缓存起来，下次使用的时候cpu直接执行，而不用再逐行解释。

通俗的解释:字节码文件相当于食物，类加载器相当于嘴，执行引擎相当于胃，垃圾回收器相当于排泄系统。



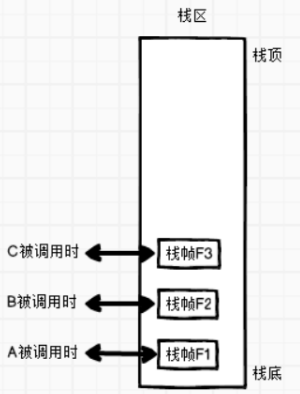
JVM理解之类的加载器理解

JVM自带加载器
启动类加载器(Bootstrap) C++提供
扩展类加载器(Extension) Java提供
应用程序加载器(Application) Java也叫系统类加载器,加载当前引用的classPath的所有类
用户自定义加载器 Java.lang.ClassLoader的子类,用户可以定制类的加载方式

1.启动类加载器：这个类加载器负责放在<JAVA_HOME>\lib目录中的，或者被-Xbootclasspath参数所指定的路径中的，并且是虚拟机识别的类库。用户无法直接使用。(就相当于是为什么我们可以通过new创建出来对象)
2.扩展类加载器：这个类加载器由sun.misc.Launcher$AppClassLoader实现。它负责<JAVA_HOME>\lib\ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库。用户可以直接使用。
3.应用程序类加载器：这个类由sun.misc.Launcher$AppClassLoader实现。是ClassLoader中getSystemClassLoader()方法的返回值。它负责用户路径（ClassPath）所指定的类库。用户可以直接使用。如果用户没有自己定义类加载器，默认使用这个。
4.自定义加载器：用户自己定义的类加载器。(只有做框架才会用到)

JVM理解之方法区
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。

存放在方法区的:静态变量+常量+类信息(构造方法/接口定义)+运行时常量池存在方法区中



JVM理解之栈

栈中存储什么: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；
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

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……

ps:栈:先进后出(FILO) 队列:先进先出(FIFO)

StackOverflowError

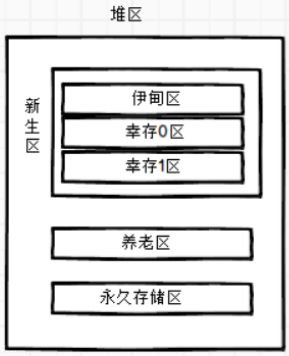
StackOverflowError:抛出这个错误是因为递归太深.其实真正的原因是因为Java线程操作是基于栈的，当调用方法内部方法也就是进行一次递归的时候就会把当前方法压入栈直到方法内部的方法执行完全之后，就会返回上一个方法，也就是出栈操作执行上一个方法

ps:这是栈中最经典的错误，StackOverflow也是一个网站里面有提出问题解决问题的网站

JVM理解之堆

堆和栈可能是在JVM问的比较多的问题,堆栈有各自的会有方式,栈区是系统回收而堆区需要的就是GC

除了基本的概念之外,那么需要将堆进行剖析



堆内存逻辑上分为三部分：新生+养老+永久

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： 伊甸区和幸存者区，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)即轻量垃圾回收，将伊甸园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。

养老区:
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MajorGC（FullGC)即重量垃圾回收，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"。

永久区:
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。

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。原因有二：
Java虚拟机的堆内存设置不够，可以通过参数-Xms、-Xmx来调整。
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）。
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

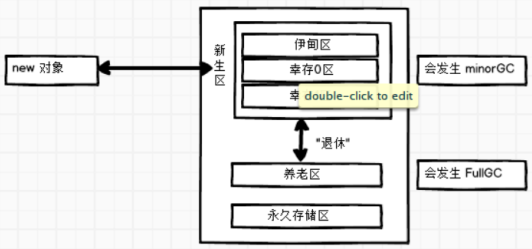
JVM理解之GC

堆中的介绍时,已经介绍出了GC的概念,所以需要对GC进行说明

什么时候"即就是GC触发的条件。GC触发的条件有两种。
1.程序调用System.gc时可以触发；
2.系统自身来决定GC触发的时机。系统判断GC触发的依据：根据新生区和老年区的内存大小来决定。当内存大小不足时，则会启动GC线程并停止应用线程。

"对什么东西"笼统的认为是Java对象。但是准确来讲，GC操作的对象分为：通过可达性分析法无法搜索到的对象和可以搜索到的对象。对于搜索不到的方法进行标记。

"做了什么"最浅显的理解为释放对象。但是从GC的底层机制可以看出，对于可以搜索到的对象进行复制操作，对于搜索不到的对象，调用finalize()方法进行释放。



GC是分代收集算法,频繁收集新生区,较少收集老年区,基本不动永久区(元空间)
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(普通/轻GC)
指发生在新生代的垃圾收集动作，因为 Java 对象大多都具备朝生夕灭的特性，所以 Minor GC 非常频繁，一般回收速度也比较快

当 JVM 无法为一个新的对象分配空间时会触发 Minor GC，比如当 伊甸园 区满了。所以分配率越高，越频繁执行 Minor GC。内存池被填满的时候，其中的内容全部会被复制，指针会从0开始跟踪空闲内存。伊甸园 和 幸存 区进行了标记和复制操作，取代了经典的标记、扫描、压缩、清理操作。所以 伊甸园 和 幸存 区不存在内存碎片。写指针总是停留在所使用内存池的顶部。执行 Minor GC 操作时，不会影响到永久代(元空间)。

majorGC 和 Full GC(全局/重GC)
指发生在老年代的 GC，出现了 Major GC，经常会伴随至少一次的 Minor GC。MajorGC 的速度一般会比 Minor GC 慢 10倍以上。

许多 Major GC 是由 Minor GC 触发的，所以很多情况下将这两种 GC 分离是不太可能的。另一方面，许多现代垃圾收集机制会清理部分永久代空间(元空间)Full GC的触发老年代空间不足或方法去空间不足或通过Minor GC后进入老年代的平均大小大于老年代的可用内存

介绍GC4大算法
1.复制算法 -->年轻代中使用的是MinorGC,这种GC算法采用的是复制算法
2.标记清除 -->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
3.标记压缩 -->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
4.标记清除压缩 -->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