JVM（HotSpot）架构分析

[一.初识HotSpot: 1](#_Toc19226456)

[二.HotSpot魔盒的打开： 4](#_Toc19226457)

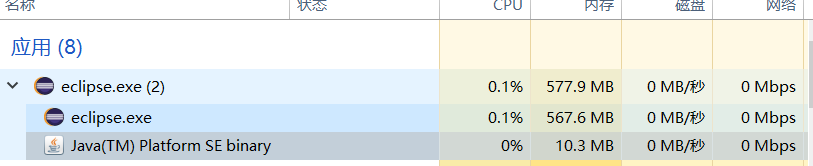
[三.顺藤摸瓜--类加载机制： 13](#_Toc19226458)

[四.身临其境—JVM内存大观园 31](#_Toc19226459)

[五.环保机制—JVM垃圾回收 41](#_Toc19226460)

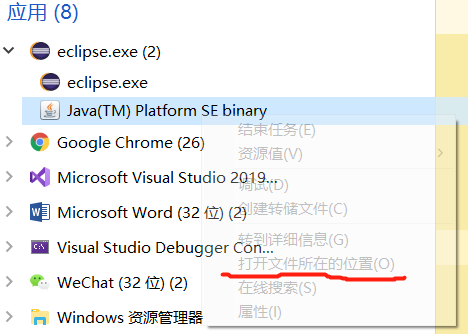
# 一.初识HotSpot:

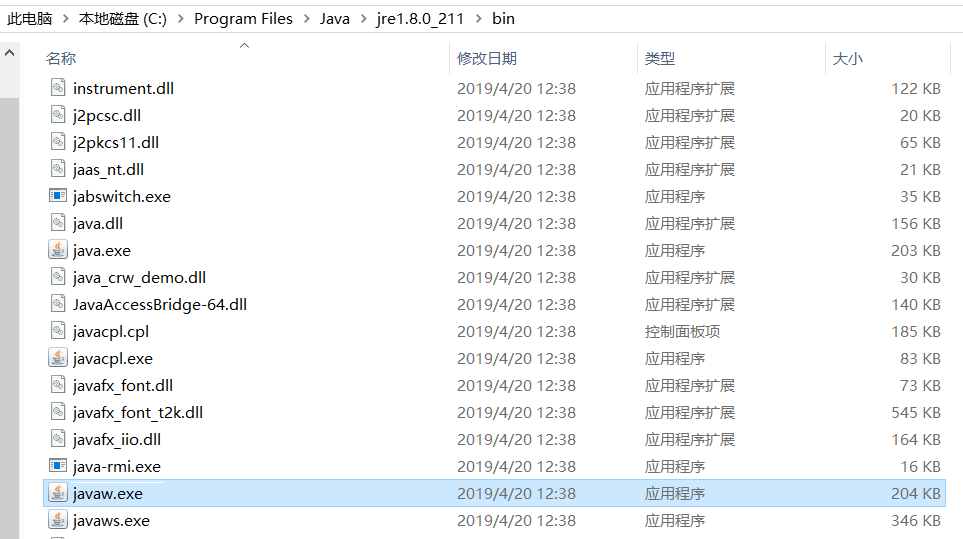
既然是软件，先把你启动起来再说（Windows）：





它在哪儿？

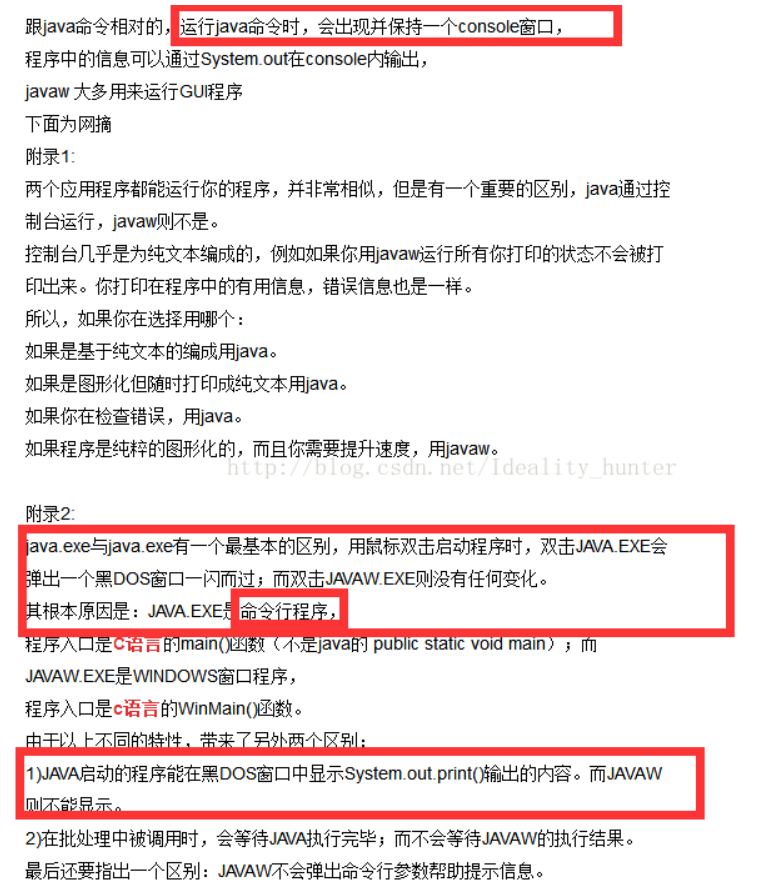




位于Java\jre1.8.0\_211\bin下的javaw.exe

暂且认为java.exe/javaw.exe都是启动虚拟机的入口,同可以运行javac后的class文件

两者区别如下：

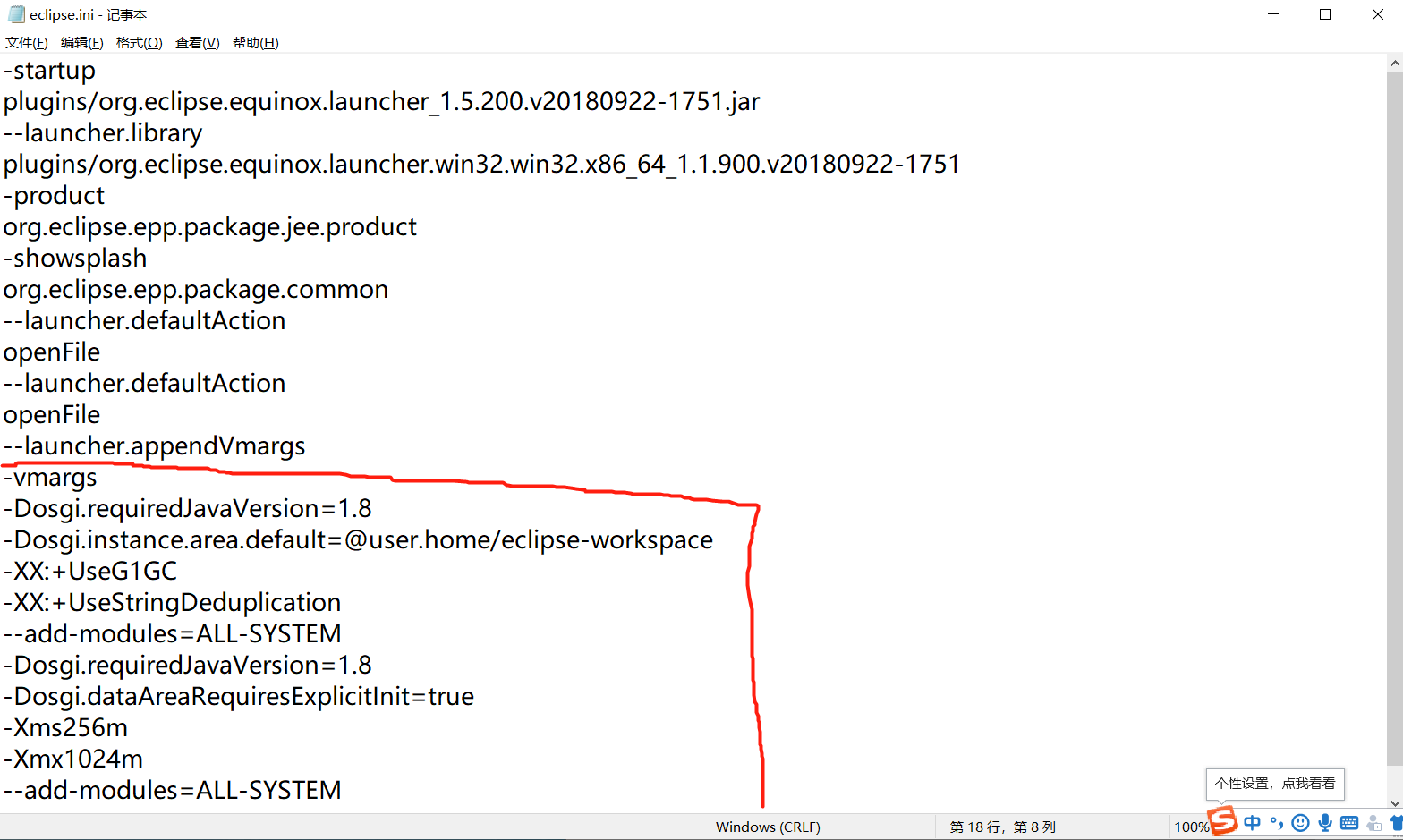


# 二.HotSpot魔盒的打开：

1.HotSpot启动时都发生了什么：

①**命令行参数处理(传说中的jvm虚拟机优化？别被吓倒，让我们一探究竟)**

**来看看eclipse启动虚拟机时的参数设置直接上图：**

****

**vmargs?虚拟机参数？好吧，应该就是你了！大概来看看都是什么意思。**

HotSpot虚拟机中有大量的可影响性能的命令行（参数）属性，这些选项可分为三个主要的类别：标准选项，非标准选项，开发者选项。

**-X**开头的选项:非标准选项（不保证所有JVM虚拟机的实现均支持，后续的JAVA SDK更新也不保证会对它进行通知）

**-XX**开头的：开发者选项（一般需要特定的系统环境，以保证实现正确的操作和足量的权限，以访问系统配置参数）

**+**：代表该属性值为true

**-**：代表该属性值为false

**这下参数啥意思基本能看了，无非就是默认开启了什么属性和属性初始化的值**

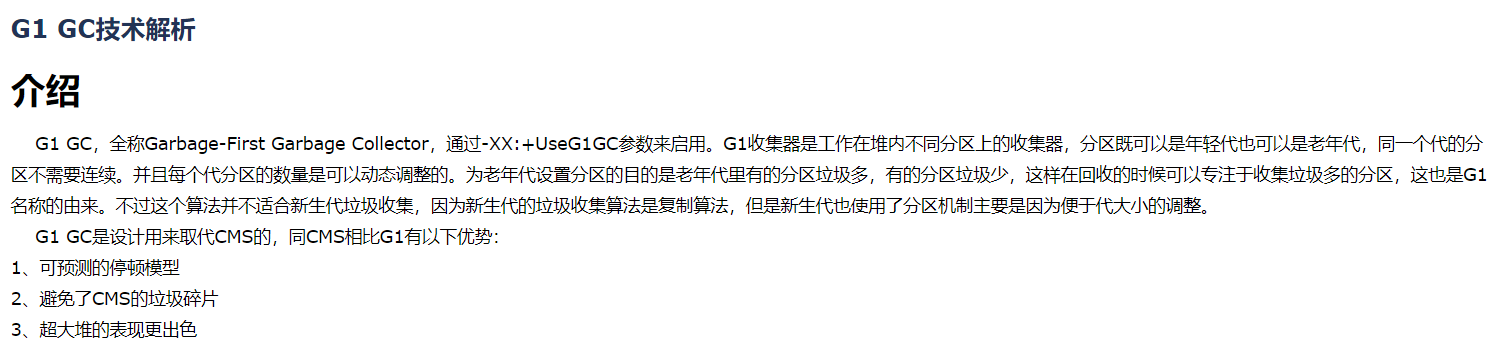
**挑几个重点的看看：**

**-Dosgi.requiredJavaVersion=1.8 //最低要求Java版本为1.8**

**-Dosgi.instance.area.default=@user.home/eclipse-workspace //eclipse默认工作区间**

**-XX:+UseG1GC //看见了GC！好吧肯定是选择了某种垃圾回收机制**

**关于G1查询资料如下：**





**-XX:+UseStringDeduplication //只用于G1垃圾收集器的**字符串去重

据说java应用内存里面的字符串有13.5%是重复的，直接导致这么多内存被浪费！

例如：

String string1 = new String("Hello World");

String string2 = new String("Hello World");

而：string1.equals(String2)=true

string1==string2=false

所以string1和string2就是所谓的重复对象

也可代码层优化：

public static final String HELLO\_WORLD = "Hello World";

String string1 = HELLO\_WORLD;

String string2 = HELLO\_WORLD;

**-Xms256m -Xmx1024m** //HotSpot虚拟机启动初始化堆内存大小256M最大值1024M

**-Xss** //为栈内存分配参数 每个线程都有独立的栈空间，所以栈内存分配的空间都很小 通常只有几百k

java.lang.StackOverflowError 这个异常太熟悉了

较深的递归函数容易发生栈溢出 可通过设置栈区大小来放缓异常的发生

避免java.lang.StackOverflowError异常的其它方式：

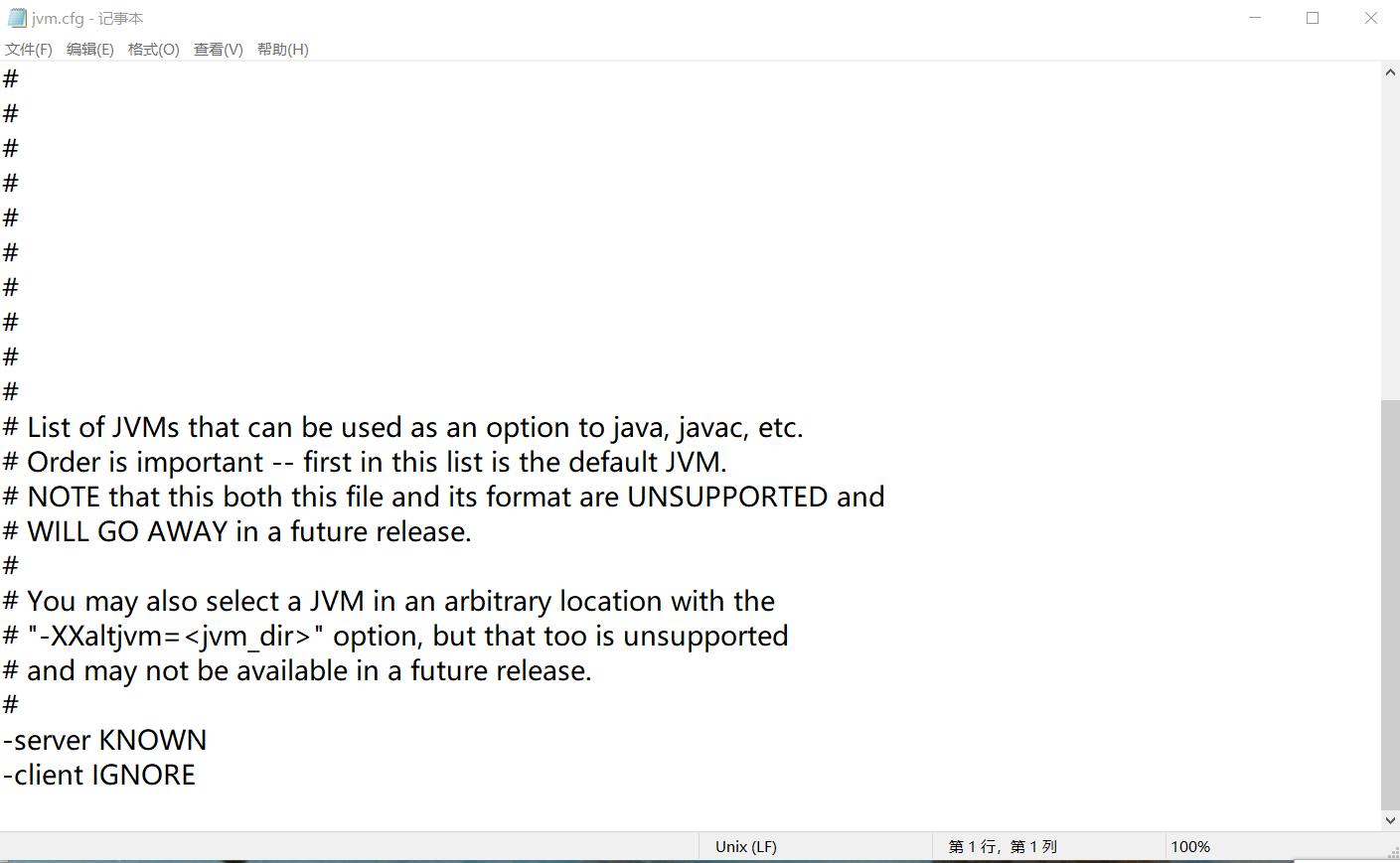
减少在递归函数中的局部变量数、内部参数数量，使得栈内存中的栈帧的局部变量表、以及成员变量参数空间压力小一些

**②虚拟机运行周期**

执行器：HotSpot虚拟机有几种Java标准版的执行器：Unix/Linux(java命令) windows(java.exe/javaw.exe)

执行步骤：

1. 解析命令行选项（如-client/-server用来决断加载合适的vm库，其他的属性则作为虚拟机初始化参数(JavaVMInitArgs)传递给vm）---如果未明确指定选项，执行器来确定堆的大小和编译器类型（是client还是server）---确立如LD\_LIBRARY\_PATH 和 CLASSPATH等环境变量---如果未在命令行中明确指定主类，执行器会从jar文件清单中找出主类名称



1. 执行器会在一个新创建的线程（非原生线程）中使用**JNI\_CreateJavaVM**来创建虚拟机实例（重点）.
2. 一旦vm创建并初始化成功，加载主类成功，执行器可从主类中得到main方法的属性，然后使用CallStaticVoidMethod执行主方法并以命令行参数为它的方法入参
3. 当java主方法执行完成时，检查和清理任何可能已发生的挂起的异常，返回退出状态。它会使用ExceptionOccurred来清理异常，方法如果执行成功，它会给调用进程返回一个0值，否则为其他值
4. 使用DetachCurrentThread解除主线程的关联，这样减少了线程的数量，保证可安全调用**DestroyJavaVM**，它也能保证线程不在vm中执行操作，栈中不再有存活的栈桢

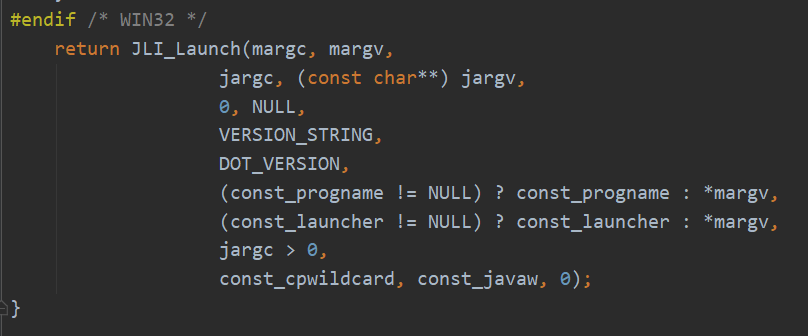
重点描述**JNI\_CreateJavaVM**和**DestroyJavaVM**两个方法

**JNI\_CreateJavaVM：**调用栈大概如下(本人基于windows平台目录结构和下图稍有差别不过出入不大)：

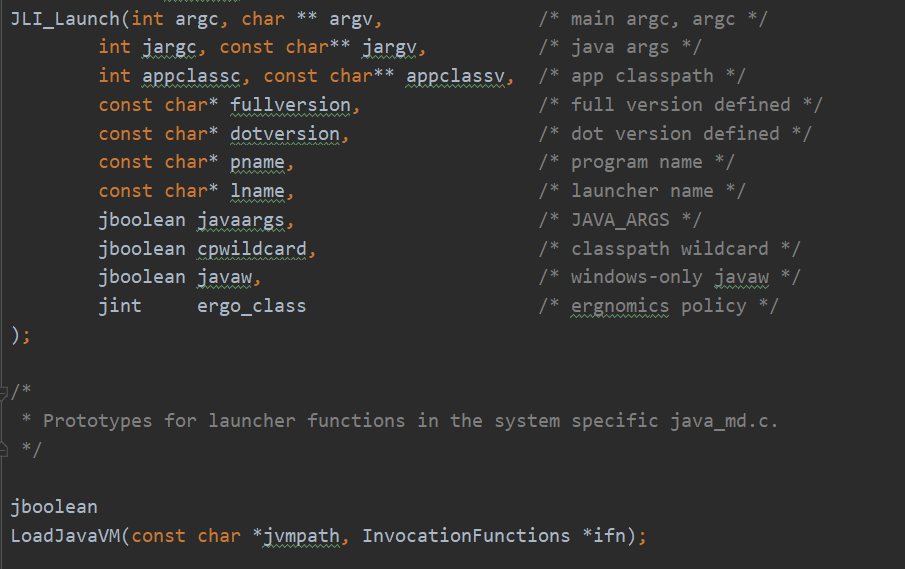


**Main.c** ：





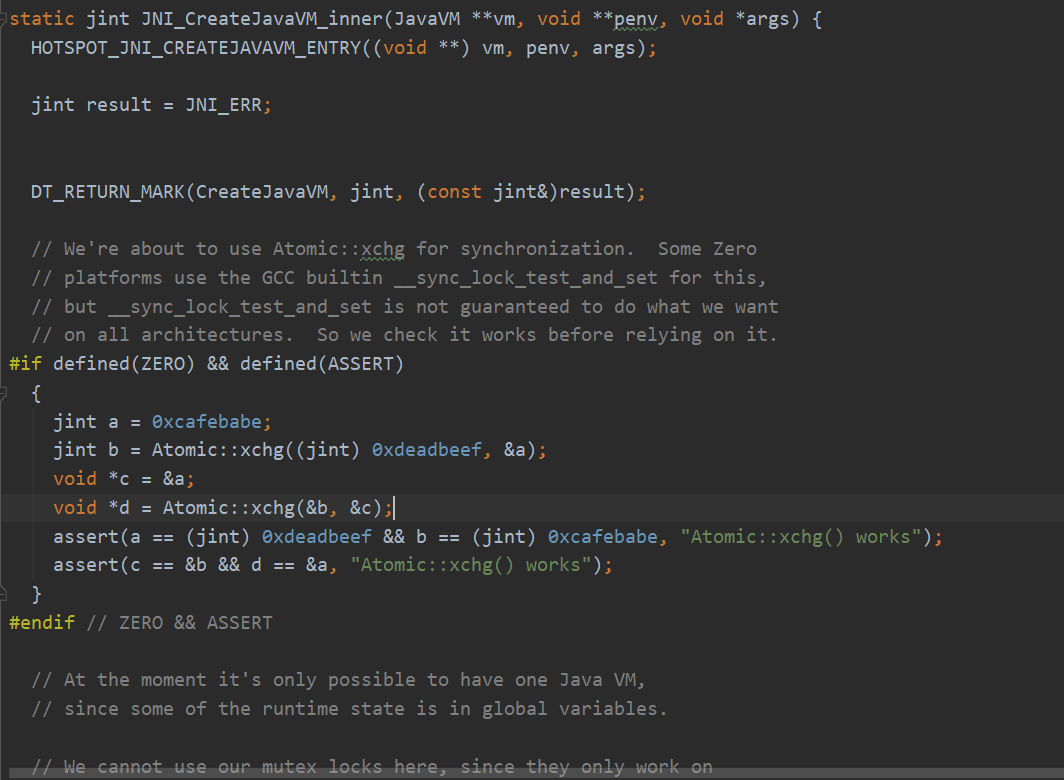
**传说中的 main 函数的真身，代码一时半会儿看不明白不过可以看出最后是返回调用了JLI\_Launch函数**

**JLI\_Launch:**  


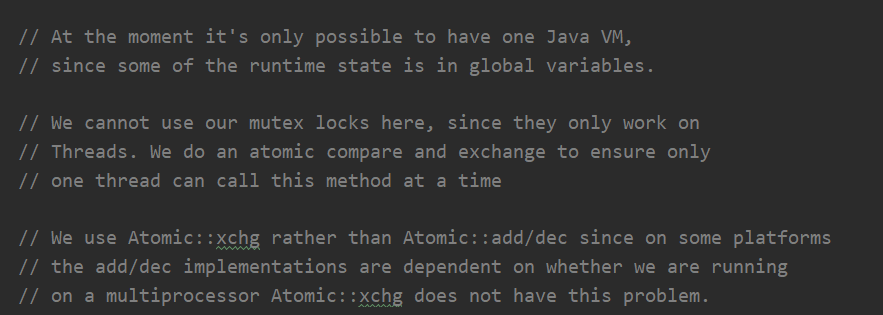
可以看到JLI\_Lanuch函数各个参数的含义，仅接着LoadJavaVM方法加载虚拟机动态链接库，并初始化 ifn 中的函数指针，HotSpot虚拟机就是这样向启动器 java 提供功能

直接跳到创建虚拟机的函数吧：

**JNI\_CreateJavaVM：**



可以看到这段代码下的注释：



我们可以看出**JNI\_CreateJavaVM**方法执行的操作及要点事项，大体为下：



绪，可服务新的JNI请求

**JNI\_CreateJavaVM**方法执行大体总结如下：

①只有单个线程（创世纪之前的那个线程）调用此方法创建VM实例，初始化到‘不可返回的点’，此时VM已初始化完毕并不可逆。

↓

②初始化操作系统模块，初始化库（例如libzip，libhpi，libjava，libthread）

↓

③开始创建Java版本的主线程并绑定到一个当前操作系统线程上（此时这个线程还不能被Threads线程列表感知）

↓

④加载主线程创建后的内部其他模块：启动类加载器（BootClassLoader），代码缓存（CodeCache），解释器（Interpreter），编译器（Compiler），JNI，系统字典（SystemDictionary），Universe（此时到达‘不可返回的点’，不能在此进程地址空间中再创建一个VM）

↓

⑤java主线程被加入到线程列表，Thread\_Lock加锁处理，对Universe（全局数据结构）进行健全检察，创建执行所有重要vm函数的VMThread

↓

⑥加载初始化一些类,例 :

java.lang.String

java.lang.System

java.lang.Thread

java.lang.ThreadGroup

java.lang.reflect.Method

java.lang.ref.Finalizer

java.lang.Class,以及系统类中的其他成员

（此时已经完成初始化并且可操作，但是并未具备完整的功能）

↓

⑦信号处理器线程，编译器线程和CompileBroker线程，以及StatSampler和WatcherThreads等辅助线程开启，此时vm具备完整功能，生成JNIEnv信息并返回给调用者，此时的vm已经准备就绪，可服务新的JNI请求

**到此，Java虚拟机终于启动起来了！ 终于可以服务我们的java代码（将Class字节码加载到内存）开始工作了！！**

**好了，接下来让我们摧毁它(手动狗头。。)**

**DestroyJavaVM：**

**①**等待到自身成为唯一一个正在运行的非守护线程时

**↓**

**②**调用java.lang.Shutdown.shutdown()方法，它会执行java级别的关闭勾子方法，调用before\_exit()，停止剖析器（Profiler），采样器（StatSampler），Watcher和GC线程

**↓**

**③**调用JavaThread的exit方法，释放JNI句柄块，移除栈保护页，把此线程从线程列表中移除，从这个点起，任何java代码不可被执行。  
终止vm线程，它会把当前的vm带到安全点并终止编译器线程。在安全点，应注意任何可能会在安全点阻塞的功能都不可使用

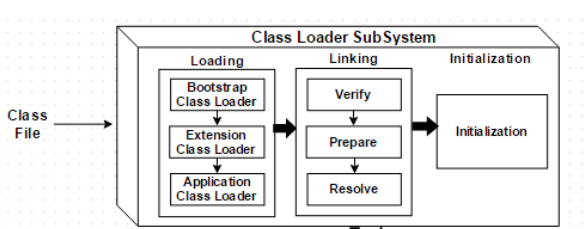
**好了，摧毁就说这么多，刚创建就摧毁，不说这些不吉利的。咱们看看虚拟机被创建后，怎么服务我们的Java源程序！**

# 三.顺藤摸瓜--类加载机制：

类加载到底是什么？

Java源文件想要执行，首先经过前端编译器(javac)编译成字节码文件

类加载就是：将.class字节码文件加载到jvm内部，虚拟机将一个类或接口的名称映射到一个jvm内部内存模型的一个class对象(java.lang.Class)的过程



由图可看出，类的加载大体分为三个阶段：

**①加载（Loading）**

**↓**

**②链接（Linking）**

**↓**

**③初始化（Initilization）**

**①加载：**

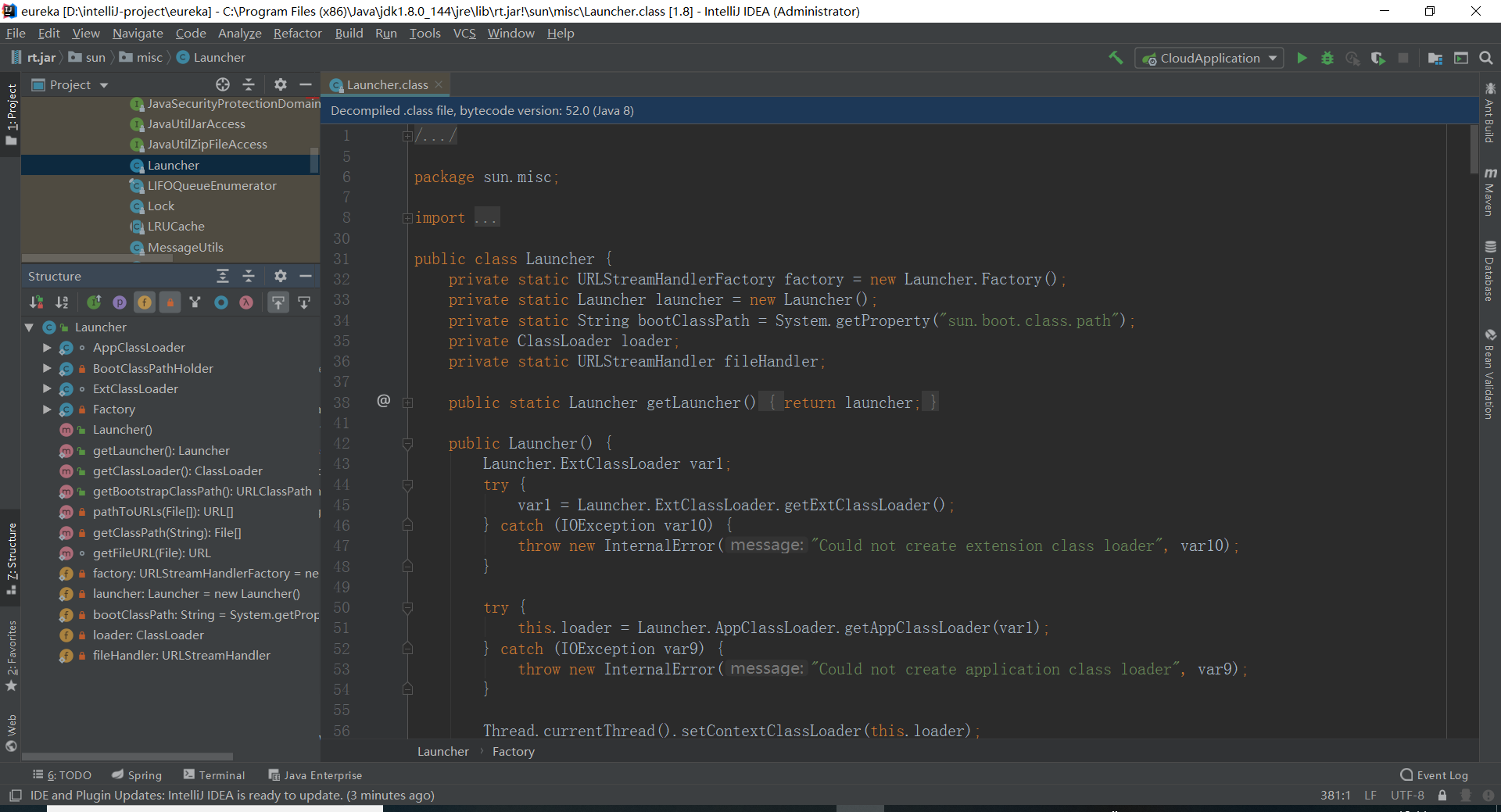
类将通过以下组件加载：

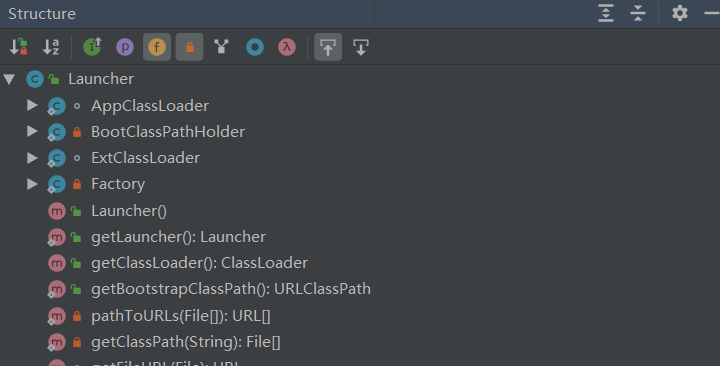
BootStrap Class Loader

Extension Class Loader

Application Class Loader

查看源码：





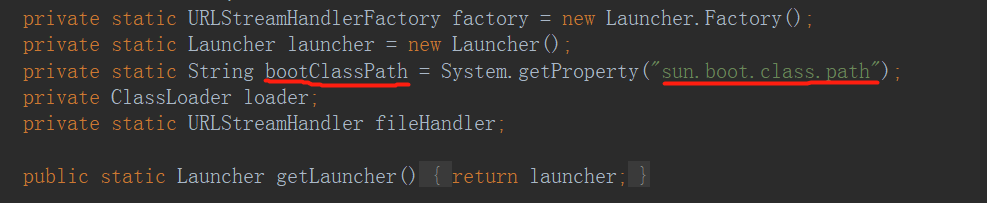
**可以看到Lancher里存在三个静态内部类即为这三个类加载器**

（1）BootStrap ClassLoader——负责加载来自于Bootstrap类路径的类，就是rt.jar。此加载程序将给予最高优先级。

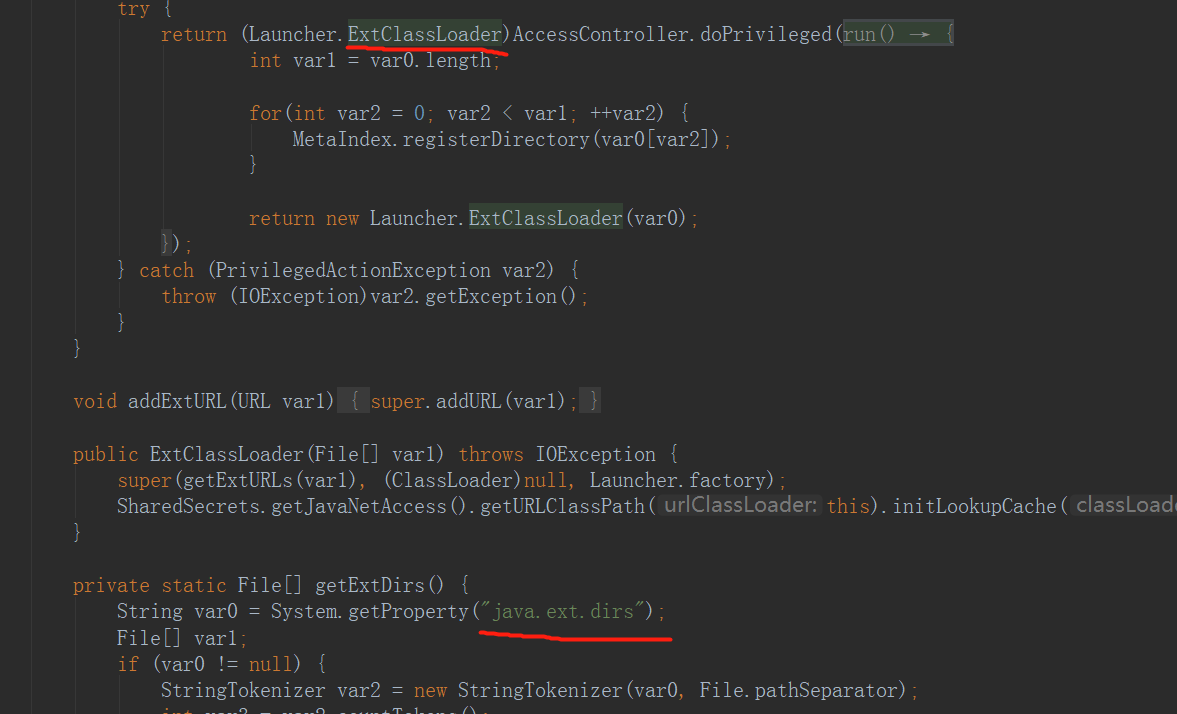
（2）Extension Class Loader——负责加载在ext文件夹（jre lib）内的类。

（3）Application Class Loader——负责加载应用程序级类路径，路径提到环境变量等

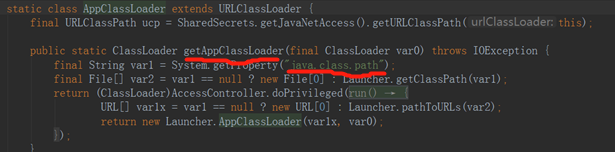
源码如下：



BootstrapClassLoader加载器负责加载的路径为：sun.boot.class.path(也就是jre/lib/rt.jar下的类)

****

**ExtClassLoader加载器负责加载类的路径为：**java.ext.dirs(也就是jre/lib/ext下的类)

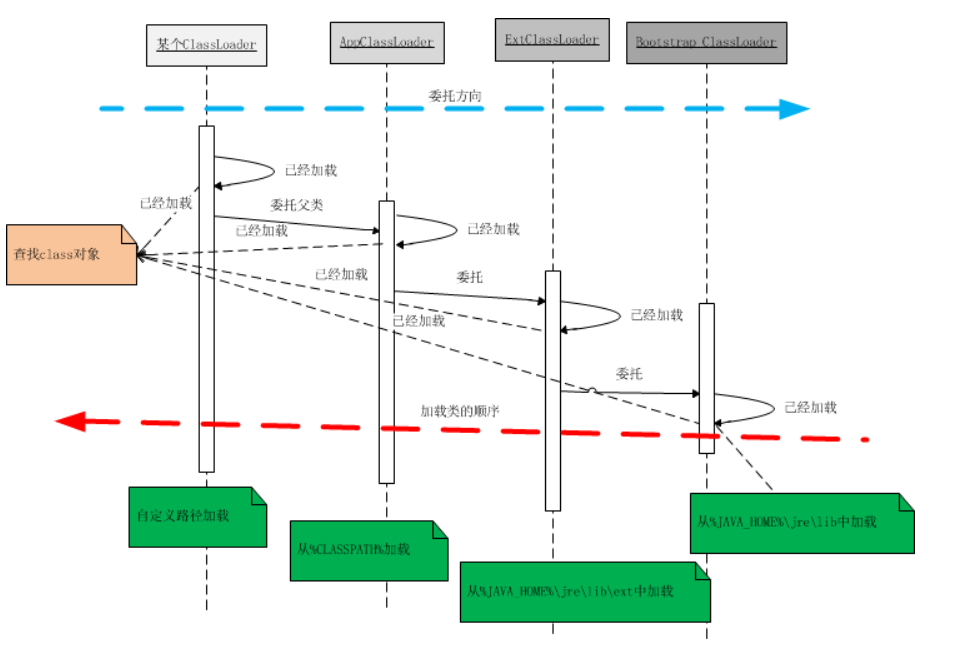
****

**AppClassLoader加载器负责加载类路径为：**java.class.path(应用程序级别，环境变量)

JVM的类加载组件采取何种机制加载类？

**JVM的类加载机制通过双亲委托机制**

**何为双亲委托机制？ 看下图：**



**双亲委托什么意思？**

一个类加载器查找class和resource时，是通过“委托模式”进行的，它首先判断这个class是不是已经加载成功，如果没有的话它并不是自己进行查找，而是先通过父加载器，然后递归下去，直到Bootstrap ClassLoader，如果Bootstrap classloader找到了，直接返回，如果没有找到，则一级一级返回，最后到达自身去查找这些对象。这种机制就叫做双亲委托

通俗大白话：即每个儿子都不愿意干活，每次有活就丢给父亲去干，直到父亲说这件事我也干不了时，儿子才自己想办法去完成

为什么要采取这种（双亲委托）机制？好处在哪里？

①提高效率：

 采用双亲委派模式的是好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系，通过这种层级关可以避免类的重复加载，当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次。

②安全！：

为什么采用这种双亲委托的类加载模式就安全呢？

假设恶意黑客从网络传来一个java.lang.String的类，通过双亲委托模式传递到启动类加载器（顶级的Bootstrap ClassLoader），而启动类加载器在核心Java API(核心API就在rt.jar下边，而顶级加载器在虚拟机启动的时候就是负责加载此路径下的所有类，即核心类)发现这个名字的类，发现该类已被加载，并不会重新加载网络传递的过来的java.lang.String，而直接返回已加载过的String.class，这样便可以防止核心API库被随意篡改。

大概了解了类加载的相关组件和机制，可是！ 还是虚无缥缈！

Talk is cheap , show me the code !

**顺藤摸瓜→刨根挖底…**

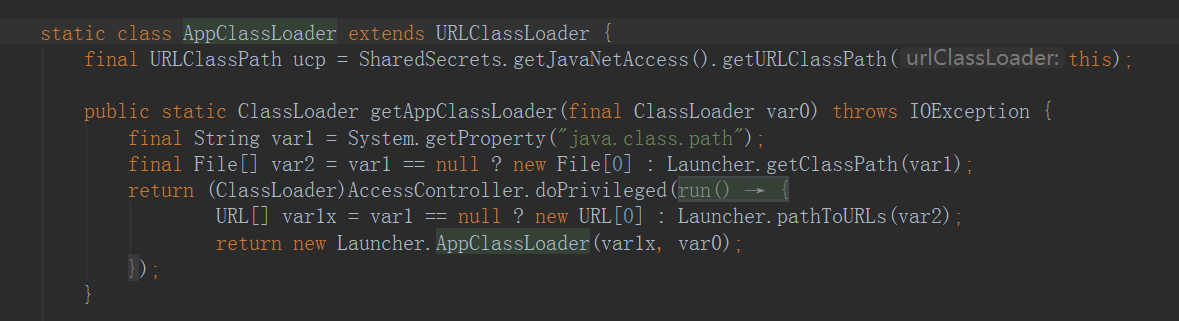
HotSpot启动自身有个类加载（加载一些核心类库，exp:Object , Thread..等等） 这个咱不说了。咱就关注一下离咱们最近的JAVASE类(上文提到的那三个类加载组件)加载库是如何加载我们平时写的java类的….

先从离咱最近的

**AppClassLoader入手：**

你在哪？

jre/lib/rt.jar/sun/misc/Launcher



可以看出AppClassLoader是Launcher类里的静态内部类继承自URLClassLoader

先暂停一下。这里有几个疑问点：

疑问点①为什么Launcher放在了rt.jar/sun…这个路径下？

不要忘了上文说的 HotSpot启动时虚拟机也会加载类（核心类）

而加载我们自己java类的：

**BootStrapClassLoader（顶级）**

**ExtClassLoader**

**AppClassLoader**

**这三大组件肯定是需要先被加载啊！**

而BootStrapClassLoader为顶级类加载器

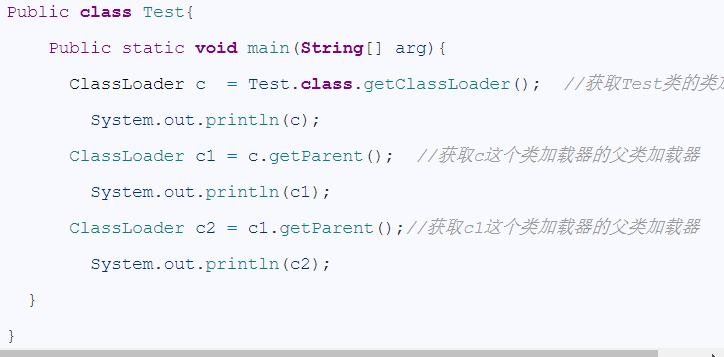
我们再回想BootStrapClassLoader加载的路径是什么？

正是rt.jar ！

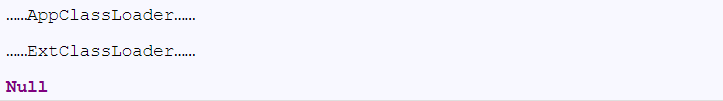
问题又来了：

疑问②那谁来加载我这个顶级类加载器（**BootStrapClassLoader**）呢？

为了一探究竟我做了一下实验：



输出结果：



可以看出：

可以看出Test是由**AppClassLoader**加载器加载的

**AppClassLoader**的Parent 加载器是 **ExtClassLoader**

但是ExtClassLoader的Parent（即**Bootstrap Loader**）为 null ？

为什么？

因为**Bootstrap Loader**是用C++语言写的，依java的观点来看，逻辑上并不存在**Bootstrap Loader**的类实体，所以在java程序代码里试图打印出其内容时，我们就会看到输出为null

好，现在我们知道老大是**Bootstrap Loader了**

**Bootstrap Loader负责加载的类路径我们知道是rt.jar**

**而rt.jar下边的类基本就是我们整个jdk了**

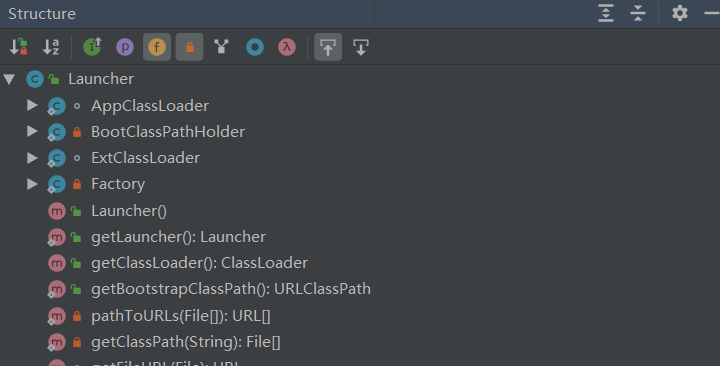
**所以对于jdk来说：万物始于Bootstrap Loader（还记得OS里的Bootstrap程序吗，特意查了下这个单词 ‘Bootstrap’：引导程序（仿佛发现了什么！））**

好，那咱接着刨**Bootstrap Loader！刨刨刨！！！**

**怎么刨呢？**（OpenJDK那么大我怎么去找！）我停下来思考了一下**：**

首先我已经在上边翻出了**Bootstrap**这个类是rt.jar/sun/misc/Launcher

**类里的一个私有静态内部类：**



**（AppClassLoader和ExtClassLoader他俩就不是私有，仿佛又印证了什么。）**

**既然Bootstrap Loader在Launcher类里，所以创生了Launcher类，Bootstrap也就创生了，接下来jdk的一切再交由Bootstrap去创生加载（好像是行得通）**

**那接着刨！Launcher怎么创生！？（这里不可能在依赖类加载机制了，肯定是native代码）**

**我在OpenJDK里边找啊找，最终我来到了这儿：**

**OpenJDK/src/java.base/share/native/launcher**

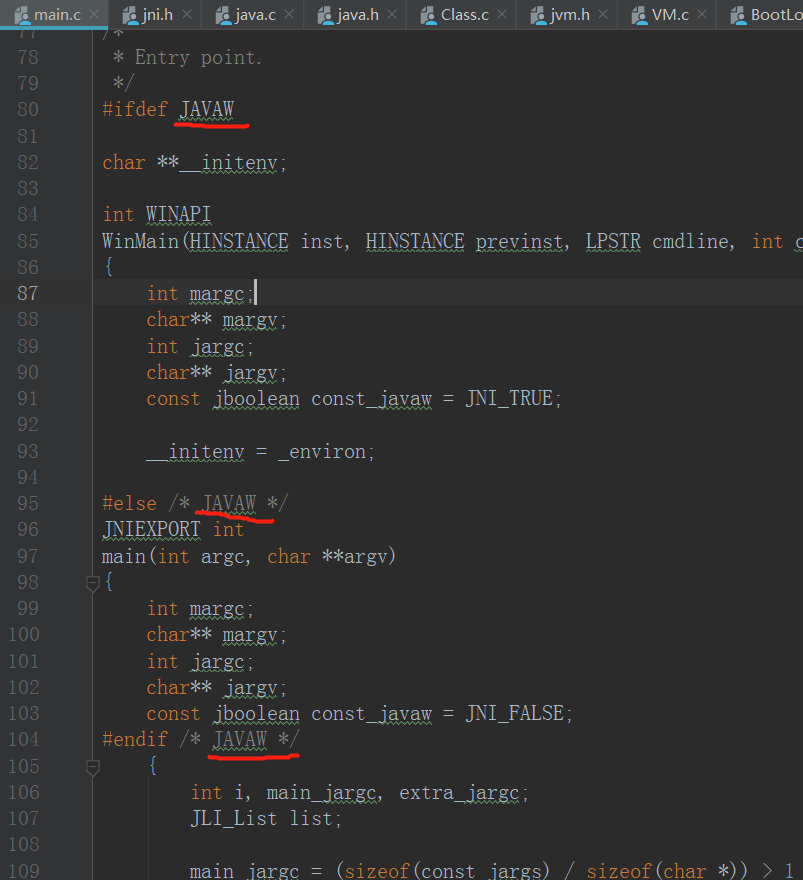
**launcher? 这是我要的那个launcher吗？**

**我打开一看：**



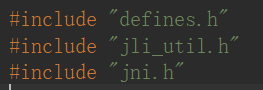
**main.c? 这是我之前分析虚拟机启动时的那个混沌入口吗？**

**打开main.c:**

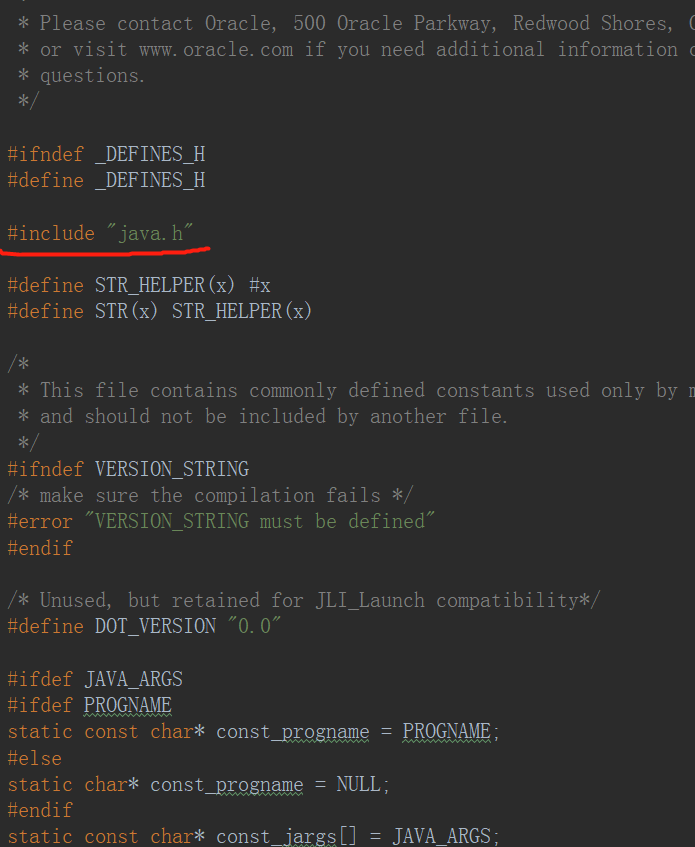
****

**javaw!这就是之前的混沌入口啊 !**

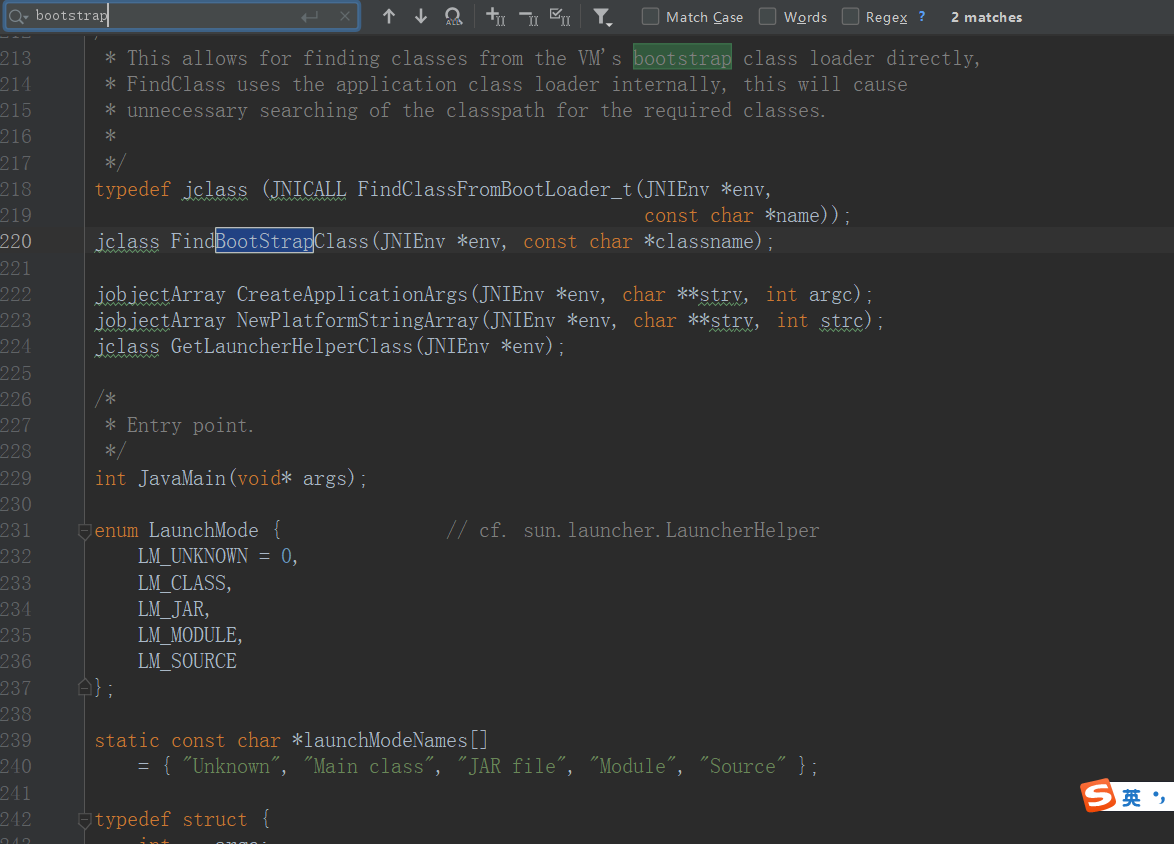
**此主函数包含了一个叫”defines.h”的头文件:**



**我们进去看看：**

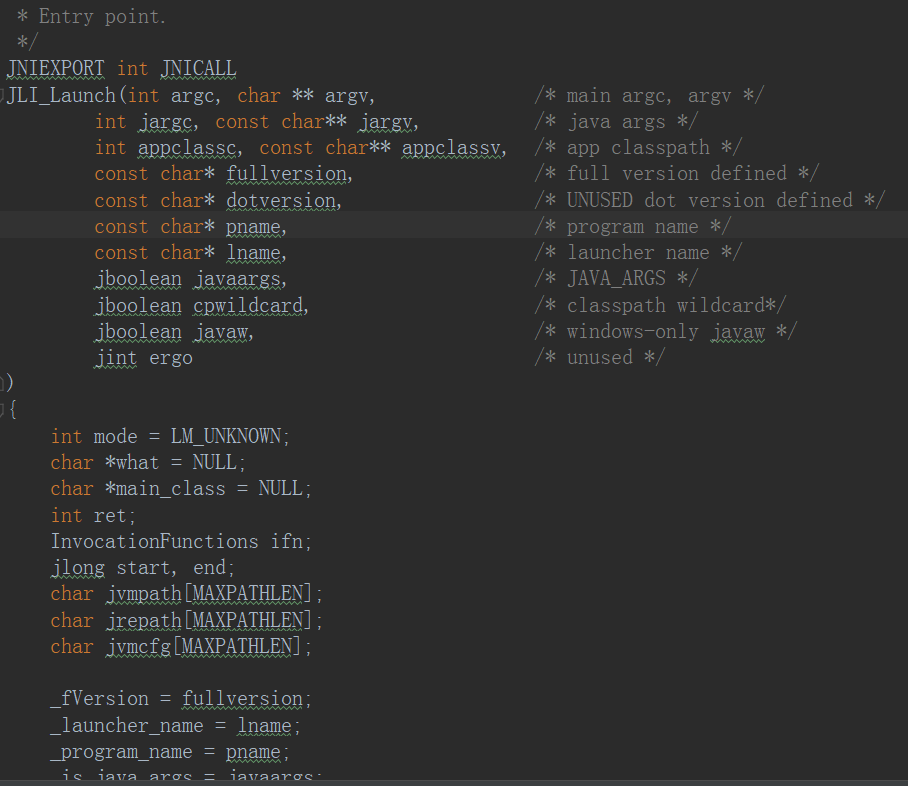
****

**这个头文件里只引入了一个”java.h”的头文件 接着刨！**



**果不其然我看见了loader !**

**然后我打开了此头文件的源代码’java.c’:**



**我又看到了什么？！**

**JLI\_Launch()方法？ 是不是似曾相识？**

**这不是我之前分析虚拟机启动时的那个方法吗？**

**JLI\_Launch返回调用JNI\_CreateJavaVM方法创建虚拟机 ! ! !**

**最终我们刨到了这个创世纪的方法，又回到了之前的原点：**

****

**从native代码一路上来，BootClassLoader（顶级加载器）终于诞生了**

**我们再回到java层代码，还记得BootClassLoader在哪吗？**

**在这个路径下：**

jre/lib/rt.jar/sun/misc/Launcher

除了misc其它单词都很一目了然，这个misc?额。Music? 不能吧

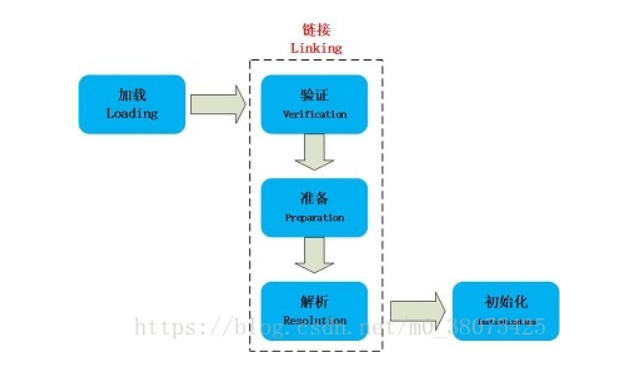
查了一下:



**万物有根，起点朦胧，此乃混沌！**

**好了，native层混沌世界太朦胧庞大（知道它怎么创生的就行）**

**我们还是跳到java层吧 ，接着回来：**



**①加载：**

**加载什么？（前文提到过）：**

加载指的是将类的class文件读入到内存，并为之创建一个java.lang.Class对象，也就是说，当程序中使用任何类时，系统都会为之建立一个java.lang.Class对象。

流程：

1.通过一个类的全限定名来获取定义此类的二进制字节流。

2． 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

3． 在Java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口。

**加载时机：**

创建类的实例，也就是new一个对象

访问某个类或接口的静态变量，或者对该静态变量赋值

调用类的静态方法

反射（Class.forName("com.csd.load")）

初始化一个类的子类（会首先初始化子类的父类）

JVM启动时标明的启动类，即文件名和类名相同的那个类

**类加载的二进制数据来源:**

从本地文件系统加载class文件，绝大部分程序的类加载方式。

从JAR包加载class文件，这种方式也是很常见的，exp:JDBC编程时用到的数据库驱动类就放在JAR文件中，JVM可以从JAR文件中直接加载该class文件。

通过网络加载class文件。

把一个Java源文件动态编译，并执行加载

**类加载器通常无须等到“首次使用”该类时才加载该类，Java虚拟机规范允许系统预先加载某些类**

**②链接**

**验证：**验证阶段用于检验被加载的类是否有正确的内部结构，并和其他类协调一致，确保Class文件的字节流中包含信息符合当前虚拟机要求，不会危害虚拟机自身安全

**1．文件格式验证:** 主要验证字节流是否符合Class文件格式规范，并且能被当前的虚拟机加载处理。例如：主，次版本号是否在当前虚拟机处理的范围之内。常量池中是否有不被支持的常量类型。指向常量的中的索引值是否存在不存在的常量或不符合类型的常量

**2.  元数据验证:** 对字节码描述的信息进行语义的分析，分析是否符合java的语言语法的规范

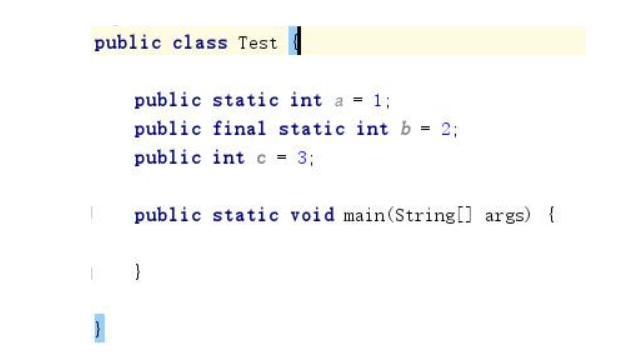
**3.**  **字节码验证：**最重要的验证环节，分析数据流和控制，确定语义是合法的，符合逻辑的。主要的针对元数据验证后对方法体的验证。保证类方法在运行时不会有危害出现

**4.**  **符号引用验证:** 主要是针对符号引用转换为直接引用的时候，是会延伸到第三解析阶段，主要去确定访问类型等涉及到引用的情况，主要是要保证引用一定会被访问到，不会出现类等无法访问的问题

**准备：**类准备阶段负责为类的静态变量分配内存，并设置默认初始值

准备到底如何为变量分配内存及设置初始值？

请看一下代码解析：



可以看到这个类里有三个变量

static int a //普通类静态变量

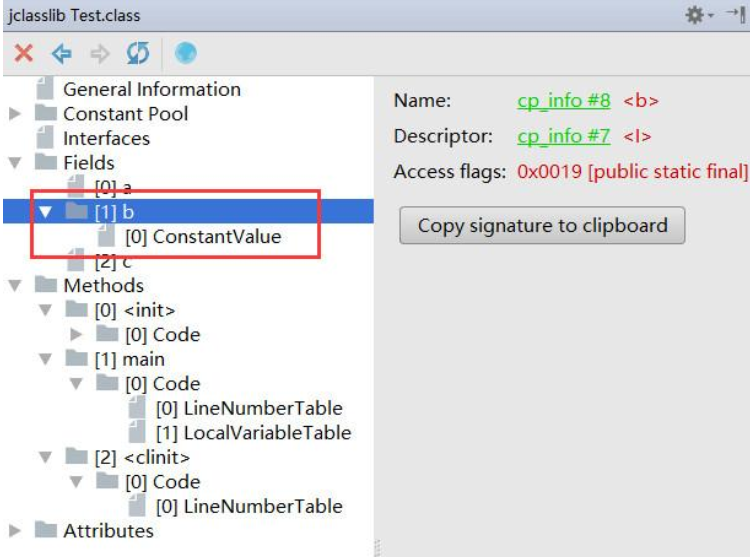
final static int b （和static final int b 一样）//不可修改的类静态变量

int c //普通变量

准备阶段是为类的静态变量分配内存，主要任务是在方法区中为类变量（仅static修饰变量，不包含实例变量）分配内存并设置类变量初始化的阶段

那么变量a和变量b以及变量c到底是怎么分配的呢？

将上述代码编译后用jclasslib（windows下的一个字节码分析工具）打开：



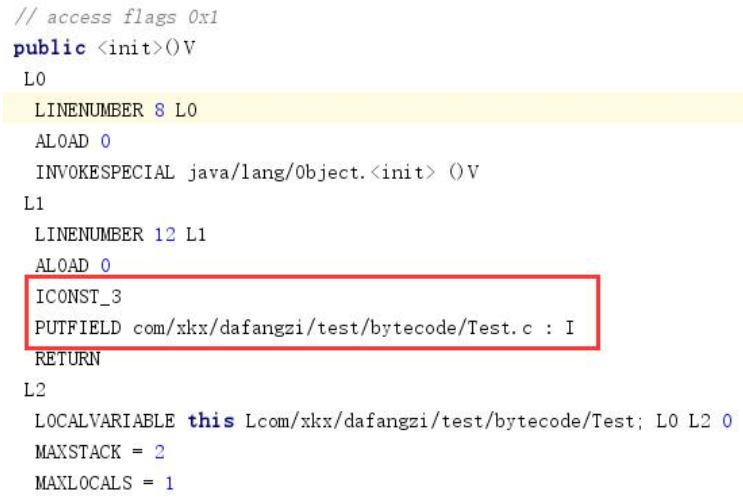
可以看出b对应的是一个ConstantValue常量，而不是一个引用，对应的，在准备阶段，虚拟机就会将常量值2赋给b



由上图可看出：而对于只有static修饰的变量a，在准备阶段，将初始化为0。如上图所示，在执行构造器方法clinit()的时候才会把把1复制给a（把1赋给a，这个是在后边要讲的初始化阶段）

到了变量c这里，则又不一样。它在准备阶段是不会进行任何操作的。如下图所示，到了对象构造方法init()中，才会把值3赋给c。

类的普通变量在对象初始化的时候随着对象分配到Java堆中。如下图：



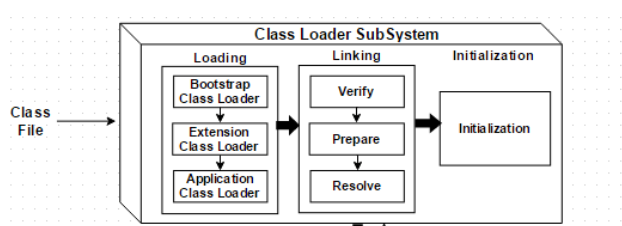
**解析**：将类的二进制数据中的符号引用替换成直接引用。说明一下：符号引用：符号引用是以一组符号来描述所引用的目标，符号可以是任何的字面形式的字面量，只要不会出现冲突能够定位到就行。布局和内存无关。直接引用：是指向目标的指针，偏移量或者能够直接定位的句柄。该引用是和内存中的布局有关的，并且一定加载进来的

**③初始化**

初始化是为类的静态变量赋予正确的初始值，准备阶段和初始化阶段看似有点矛盾，其实是不矛盾的，如果类中有语句：private static int a = 10，它的执行过程是这样的，首先字节码文件被加载到内存后，先进行链接的验证这一步骤，验证通过后准备阶段，给a分配内存，因为变量a是static的，所以此时a等于int类型的默认初始值0，即a=0,然后到解析（后面在说），到初始化这一步骤时，才把a的真正的值10赋给a,此时a=10

**类的整个加载过程差不多就是这样**

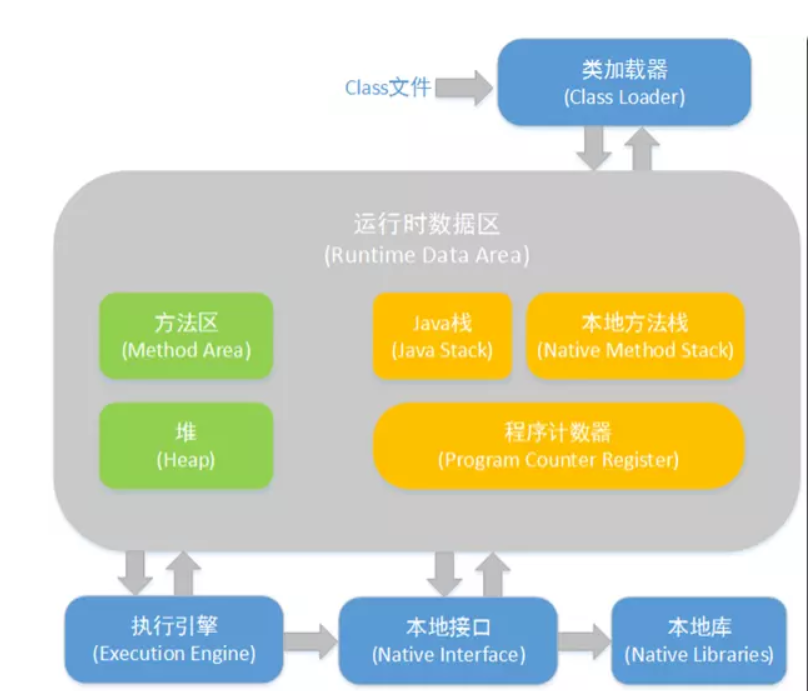
**再把最开始那个大图贴出来（理一理整个流程）：**



# 四.身临其境—JVM内存大观园

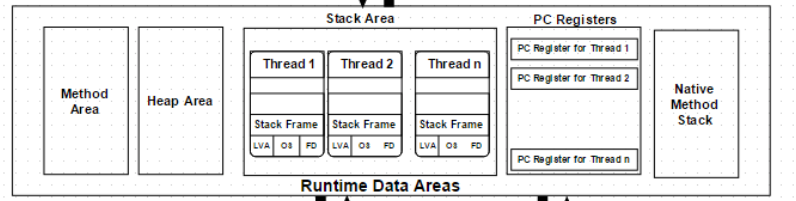
在学习JVM内存模型(JMM)之前，先回顾下之前类加载与内存模型之间存在的关系

我先上个图：



类加载器将Class文件读取后，放到运行时数据区，然后执行引擎执行或调用本地接口、本地库

JVM内存模型大体如下图：



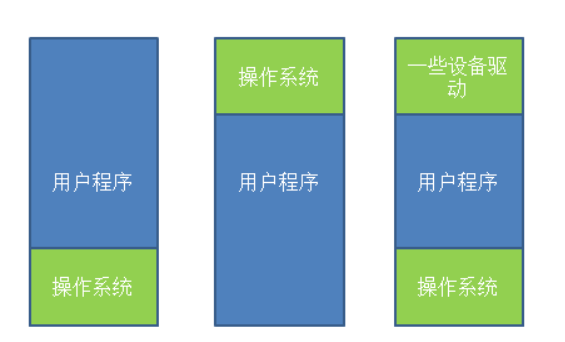
既然叫jvm**内存**模型，那这个模型肯定是在计算机的内存中！（废话）

内存是非常重要的系统资源，是硬盘和CPU的中间仓库及桥梁，承载着操作系统和应用程序的实时运行，JVM内存布局规定了Java在运行过程中内存申请、分配、管理的策略，保证了JVM的高效稳定运行

所以咱们来看看当初sun公司的那帮神仙们是怎么设计jvm内存区域的！

说到这JVM的内存划分，不得提及一个概念：**多线程**！

在计算机组成与结构（COA）里我们知道：内存就是一整块用来暂时存储信息的物理介质（晶体管），它里边没有任何的划分。然后到操作系统这儿，操作系统要管理内存，然后OS大哥又搞出了什么MMU(内存管理单元)和PMMU(分页内存管理单元)通过这种虚拟映射把内存好像是分配了一下（但也只是映射，并没有实质性的将内存区域明显划分开）因为远古时期，CPU也没有什么时间片轮转及相应的调度算法，就是只有一个程序单独的在执行，除了操作系统占用的内存除外，其余内存全交给程序，如下图：



第一种情况操作系统存于RAM中，放在内存的低地址，第二种情况操作系统存在于ROM中，存在内存的高地址，一般老式的手机操作系统是这么设计的。

    如果这种情况下，想要操作系统可以执行多进程的话，唯一的解决方案就是和硬盘搞交换，当一个进程执行到一定程度时，整个存入硬盘，转而执行其它进程，到需要执行这个进程时，再从硬盘中取回内存，只要同一时间内存中只有一个进程就行，这也就是所谓的交换（Swapping）技术。但这种技术由于还是直接操作物理内存，依然有可能引起进程的崩溃。

    所以，通常来说，这种内存操作往往只存在于一些洗衣机，微波炉的芯片中，因为不可能有第二个进程去征用内存

因为CPU和主存之间的IO速率差距的天壤之别，CPU实在忍受不了内存的，慢了! 后来CPU开始实现了调度，在和内存IO等待的时候我去执行其它程序，到时候之前读取指令的操作完了再给CPU发中断请求我再回到之前的现场执行之前的指令，这样大大的提高了程序执行的效率（可以运行多个程序了）。多进程的出现不得不让OS大哥管理进程的内存分配了（不然不同的进程混淆使用同一内存地址空间那就麻烦大了！）

多进程时代OS大哥是怎么管理内存分配的呢？

**内存抽象：**

在现代的操作系统中，同一时间运行多个进程是再正常不过的了。为了解决直接操作内存带来的各种问题，引入的地址空间(Address Space),这允许每个进程拥有自己的地址。这还需要硬件上存在两个寄存器，基址寄存器(base register)和界址寄存器(limit register),第一个寄存器保存进程的开始地址，第二个寄存器保存上界，防止内存溢出。在内存抽象的情况下，当执行

mov reg1,20

  这时，实际操作的物理地址并不是20，而是根据基址和偏移量算出实际的物理地址进程操作，此时操作的实际地址可能是:

mov reg1,16245

  在这种情况下，任何操作虚拟地址的操作都会被转换为操作物理地址。而每一个进程所拥有的内存地址是完全不同的，因此也使得多进程成为可能。

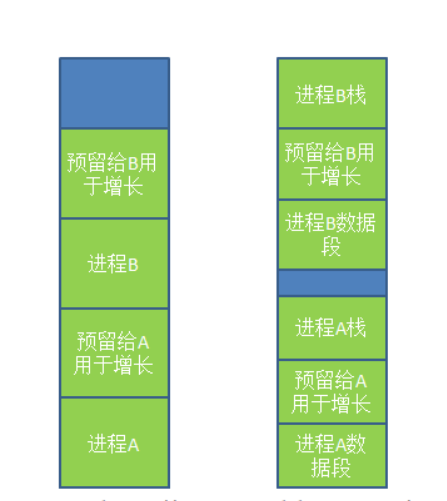
有了基址寄存器和界址寄存器，虽然可以运行多个程序了，但是当有很多个进程都放在内存里的时候，内存也肯定发不下，这又引出了交换技术（和上文说的类似），通过移动进程在内存中的地址，使得这些外部碎片空间（食之不够弃之可惜）被填满，还有就是如内存整理软件，原理是申请一块超大的内存，将所有进程置换出内存，然后再释放这块内存，从而使得从新加载进程，使得外部碎片被消除。这也是为什么运行完内存整理会狂读硬盘的原因

然后操作系统为了进程在运行期间占用内存的动态性，又在进程内存分配这儿搞了两点新特色：

①一个解决方法是在进程创建的时候，比进程实际需要的多分配一点内存空间用于进程的增长

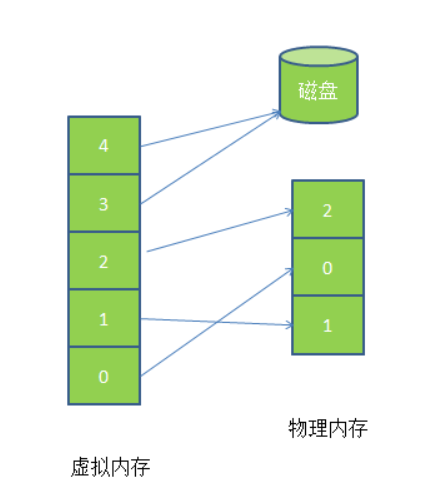
②另一种是将增长区分为数据段和栈（用于存放返回地址和局部变量）

如下图所示：



**虚拟内存：** 虚拟内存是现代操作系统普遍使用的一种技术。前面所讲的抽象满足了多进程的要求，但很多情况下，现有内存无法满足仅仅一个大进程的内存要求(比如很多游戏，都是10G+的级别)。在早期的操作系统曾使用覆盖(overlays)来解决这个问题，将一个程序分为多个块，基本思想是先将块0加入内存，块0执行完后，将块1加入内存。依次往复，这个解决方案最大的问题是需要程序员去程序进行分块，这是一个费时费力让人痛苦不堪的过程。后来这个解决方案的修正版就是虚拟内存。

虚拟内存的基本思想是，每个进程有用独立的逻辑地址空间，内存被分为大小相等的多个块,称为页(Page).每个页都是一段连续的地址。对于进程来看,逻辑上貌似有很多内存空间，其中一部分对应物理内存上的一块(称为页框，通常页和页框大小相等)，还有一些没加载在内存中的对应在硬盘上，如下图所示：

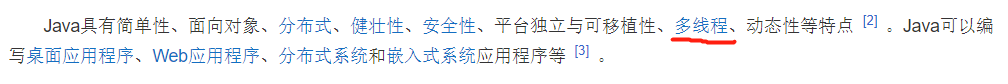


这后边又要牵扯出什么页面置换算法了。。这里就再不啰嗦了。

接着回到咱之前的话题：

**多线程**以及sun公司的那帮神仙是怎么设计jvm内存区域的！

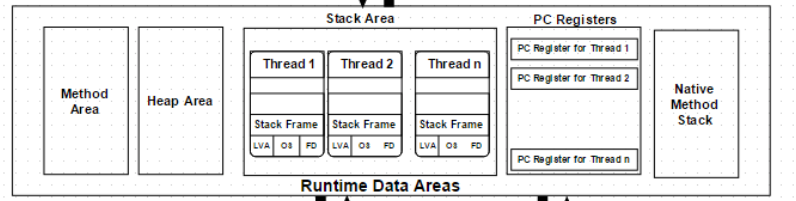
众所周知Java是一种支持多线程编程的语言：



说白了如果java不支持多线程编程，那JVM大可不必要划分内存模型，因为你要单是作为一个进程，那直接交给操作系统大哥就好了，由它给你分配管理内存，所以，正是因为其多线程的特点，JVM在它这一层划分了很多区域，用来支撑Java的多线程程序

好，那sun公司的大佬们为了这个所谓的线程（当然内存模型的划分不光是为了线程，但线程在内存模型中占了很重要的地位）它到底是怎么给java程序划分内存区域的呢？

再次贴图：



Jvm内存模型总共就划分为了五大块：

1. **方法区**(也称元数据区)：所有的类级别数据将存储在这里，包括静态变量。每个JVM只有一个方法区，并且它是一个共享资源
2. **堆区域**：所有对象及其对应的实例变量和数组将存储在这里。每个JVM也有一个堆区域，由于方法和堆区域共享多个线程的内存，因此所存储的数据非线程安全。
3. **堆栈区**：对于每个线程，将创建一个单独的运行时栈。对于每个方法调用，将在堆栈存储器中产生一个条目，称为堆栈帧。所有局部变量将在堆栈内存中创建。堆栈区域是线程安全的，因为它不是共享资源。堆栈帧分为三个子元素：

①局部变量数组：与方法相关，涉及局部变量，并在此存储相应的值

②操作数堆栈：如果需要执行任何中间操作，操作数堆栈将充当运行时工作空间来执行操作

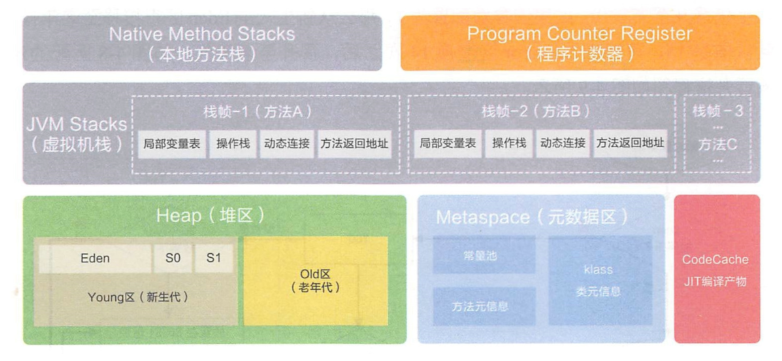
③帧数据：对应于方法的所有符号存储在此处。在任何异常的情况下，捕获的区块信息将被保持在帧数据中

1. **PC寄存器**：每个线程都有单独的PC寄存器，用于保存当前执行指令的地址，一旦指令执行，PC寄存器将更新到下一条指令
2. **本地方法堆栈**：本地方法堆栈保存本地方法信息。对于每个线程，将创建一个单独的本地方法堆栈

线程共享：方法区，堆

线程独享：堆栈区，PC计数寄存器，本地方法栈

好，上边咱们全局纵观了一下JVM的五大内存模块，再贴一个图接着看：

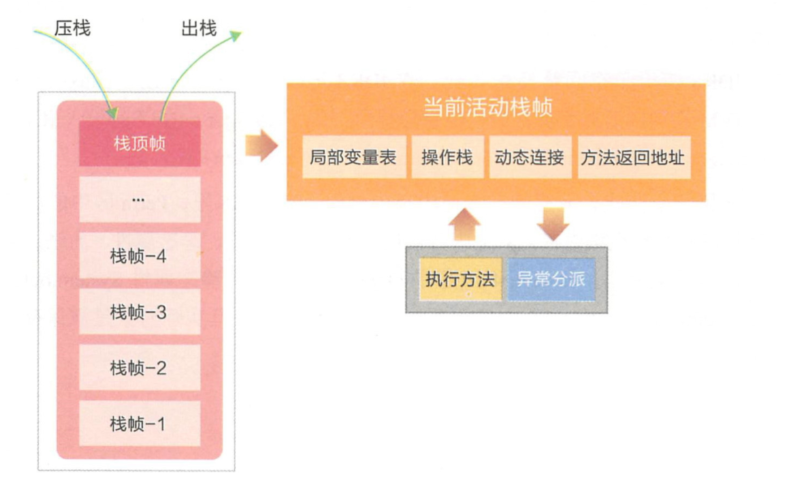


基于以上大局观，再分析一边：

1. 方法区（上图元数据区）：线程共享的运行时内存区，它存储每一个类的实例信息，运行时常量池，字段和方法数据，构造函数和普通方法的字节码等内容。还有一些特殊方法。方法区是堆的逻辑组成部分，也在JVM启动时创建，简单的JVM可以不实现这个区域的垃圾收集。  
   方法区也可固定大小和动态分配与堆一样，内存空间不够，那么JVM抛出OutOfMemoryError异常。
2. 堆区: 线程共享的运行时存储区域，也是供所有类的实例和数组对象分配内存的区域。堆在JVM启动的时候创建，堆所存储的就是被GC所管理的各种对象。  
   堆也是可以固定大小和动态调整的。实际所需的堆超过的GC所提供的最大容量，那么JVM抛出OutOfMemoryError异常(垃圾回收机制后边重点说)
3. 栈：JVM中的虚拟机栈是描述Java方法执行的内存区域，它是线程私有的，栈中的元素用于支持虚拟机进行方法调用，每个方法从开始调用到执行完成的过程，就是栈帧从入栈到出栈的过程。

在活动线程中，只有位于栈顶的帧才是有效的，称为当前栈帧  
正在执行的方法称为当前方法  
栈帧是方法运行的基本结构

在执行引擎运行时，所有指令都只能针对当前栈帧进行操作  
StackOverflowError表示请求的栈溢出，导致内存耗尽，通常出现在递归方法中  
JVM能够横扫千军，虚拟机栈就是它的心腹大将，当前方法的栈帧，都是正在战斗的战场，其中的操作栈是参与战斗的士兵，看下图：

  
虚拟机栈通过压/出栈的方式，对每个方法对应的活动栈帧进行运算处理，方法正常执行结束，肯定会跳转到另一个栈帧上  
在执行的过程中，如果出现异常，会进行异常回溯，返回地址通过异常处理表确定  
栈帧在整个JVM体系中的地位颇高,包括局部变量表、操作栈、动态连接、方法返回地址等

 局部变量表  
存放方法参数和局部变量  
相对于类属性变量的准备阶段和初始化阶段来说，局部变量没有准备阶段，必须显式初始化  
如果是非静态方法，则在index[0]位置上存储的是方法所属对象的实例引用，随后存储的是参数和局部变量  
字节码指令中的STORE指令就是将操作栈中计算完成的局部变量写回局部变量表的存储空间内

 操作栈  
操作栈是一个初始状态为空的桶式结构栈  
在方法执行过程中，会有各种指令往栈中写入和提取信息  
JVM的执行引擎是基于栈的执行引擎，其中的栈指的就是操作栈  
字节码指令集的定义都是基于栈类型的,栈的深度在方法元信息的stack属性中

* 动态连接  
  每个栈帧中包含一个在常量池中对当前方法的引用，目的是支持方法调用过程的动态连接
* 方法返回地址  
  方法执行时有两种退出情况
  + 正常退出  
    正常执行到任何方法的返回字节码指令，如RETURN、IRETURN、ARETURN等
  + 异常退出

无论何种退出情况，都将返回至方法当前被调用的位置。方法退出的过程相当于弹出当前栈帧

退出可能有三种方式:

* 返回值压入，上层调用栈帧
* 异常信息抛给能够处理的栈帧
* PC计数器指向方法调用后的下一条指令

特点：局部变量表的创建是在方法被执行的时候,随着栈帧的创建而创建.  
而且表的大小在编译期就确定,在创建的时候只需分配事先规定好的大小即可.  
在方法运行过程中,表的大小不会改变

Java虚拟机栈会出现两种异常

* **StackOverFlowError**  
  若Java虚拟机栈的内存大小不允许动态扩展,那么当线程请求的栈深度大于虚拟机允许的最大深度时(但内存空间可能还有很多),就抛出此异常
* **OutOfMemoryError**  
  若Java虚拟机栈的内存大小允许动态扩展,且当线程请求栈时内存用完了,无法再动态扩展了,此时抛出OutOfMemoryError异常

**这两个错误的区别 ! 牢记 !**

Java虚拟机栈也是线程私有的,每个线程都有各自的Java虚拟机栈,而且随着线程的创建而创建,随着线程的死亡而死亡.

1. **程序计数寄存器（Program Counter Register）：**Register 的命名源于CPU的寄存器，CPU只有把数据装载到寄存器才能够运行  
   寄存器存储指令相关的现场信息，由于CPU时间片轮限制，众多线程在并发执行过程中，任何一个确定的时刻，一个处理器或者多核处理器中的一个内核，只会执行某个线程中的一条指令。这样必然导致经常中断或恢复，如何保证分毫无差呢?  
   每个线程在创建后，都会产生自己的程序计数器和栈帧，程序计数器用来存放执行指令的偏移量和行号指示器等，线程执行或恢复都要依赖程序计数器。程序计数器在各个线程之间互不影响，此区域也不会发生内存溢出异常
2. **本地方法栈：**本地方法栈和Java虚拟机栈实现的功能与抛出异常几乎相同，只不过虚拟机栈是为虚拟机执行Java方法(也就是字节码)服务,本地方法区则为虚拟机使用到的Native方法服务。

在JVM内存布局中，也是线程对象私有的,但是虚拟机栈“主内”，而本地方法栈“主外”  
这个“内外”是针对JVM来说的，本地方法栈为Native方法服务  
线程开始调用本地方法时，会进入一个不再受JVM约束的世界  
本地方法可以通过JNI(Java Native Interface)来访问虚拟机运行时的数据区，甚至可以调用寄存器,具有和JVM相同的能力和权限  
当大量本地方法出现时,势必会削弱JVM对系统的控制力,因为它的出错信息都比较黑盒.  
对于内存不足的情况，本地方法栈还是会拋出native heap OutOfMemory

典型的native方法：

System.currentTimeMillis()…

# 五.环保机制—JVM垃圾回收



先来个梗：说在食堂里吃饭，吃完饭把餐盘端走清理的是C++程序员，吃完直接走的，是java程序员（狗头。。）

①什么东西需要回收呢？

当然是没有用的垃圾啊！在Java里边就是那些没有用的垃圾对象喽~

②在哪儿回收？（jvm内存模型里的那块区域）

有对象的地方回收喽！ 那对象放在jvm的那个内存区域呢？

我们知道线程独享的有：堆栈区，PC寄存器，本地方法栈，这三个区域都是线程独享的，随着线程的灭亡而灭亡，因此这几个区域的内存分配和回收都具备确定性，就不需要过多考虑回收的问题，因为方法结束或者线程结束时，内存自然就跟随着回收了。

线程共享的内存区域有：方法区和堆区，而我们知道，方法区放的都是和类有关的信息（static变量..），而对象实例都放在堆区，堆区(Heap)是OOM故障最主要的发源地，它存储着几乎所有的实例对象，通常情况下，它占用的空间是所有内存区域中最大的，但如果无节制地创建大量对象，也容易消耗完所有的空间,所以垃圾回收重点在堆区进行回收。（方法区我们在后边也叫**持久代**，也会在此区域进行GC，不过因为方法区存放的都是关于类的一些信息，静态文件，java类，方法等，所以持久代对垃圾回收没有显著影响

知道了jvm主要是在堆区进行GC，那根据GC，又可将整个jvm划分为那些区呢?

③堆区划分？（根据GC划分）

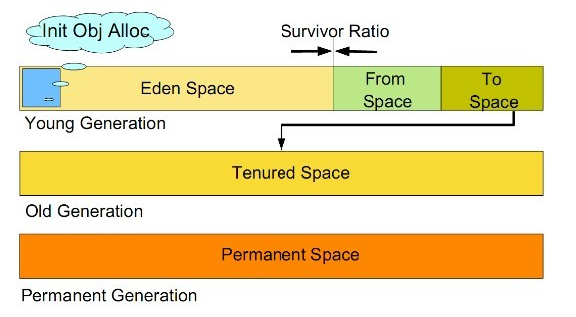
堆分成三大块:

**年轻代（Young Generation）**

**老年代(Old Generation)**

**//持久代(Permanent Generation)（方法区）**

贴图更直观：



**常用的垃圾收集算法**

**①引用计数法：**

每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可用回收。此方法简单，无法解决对象相互循环引用（定义两个对象，相互引用，置空各自的声明引用）的问题，且每次对对象复制时均要维护引用计数器，且计数器本身也有一定的消耗（JVM一般不采用这种方式）

**②复制算法：**

①**年轻代（Young Generation）**的回收

a) 所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。

b) 新生代内存按照8:1:1的比例分为一个eden区和两个survivor(survivor0,survivor1)区。一个Eden区，两个 Survivor区(一般而言)。大部分对象在Eden区中生成。回收时先将eden区存活对象复制到一个survivor0区，然后清空eden区，当这个survivor0区也存放满了时，则将eden区和survivor0区存活对象复制到另一个survivor1区，然后清空eden和这个survivor0区，此时survivor0区是空的，然后将survivor0区和survivor1区交换，即保持survivor1区为空， 如此往复(survivor1其实充当了临时中转的作用)。

c) 当survivor1区不足以存放 eden和survivor0的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次Full GC（完全GC），也就是新生代、老年代都进行回收。

d) 新生代发生的GC也叫做Minor GC（小型GC），MinorGC发生频率比较高(不一定等Eden区满了才触发)。

②**年老代（Old Generation）**的回收

a) 在年轻代中经历了N（由JVM参数MaxTenuringThreshold决定，这个参数默认是15）次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

b) 内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发Major GC(大型GC)即Full GC，Full GC发生频率比较低，老年代对象存活时间比较长，存活率标记高。

③**持久代（Permanent Generation）(**也称为方法区**)**的回收

用于存放静态文件，如Java类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，例如Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。

**③标记清除算法：**

它的做法是当堆中的有效内存空间(available memory)被耗尽的时候，就会让整个程序stop然后进行标记再清除  
标记：标记的过程其实就是，遍历所有的GC Roots，然后将所有的GC Roots可达的对象标记为存活的对象。  
清除：清除的过程将遍历堆中所有的对象中没有标记的对象全部清除掉  
缺点：  
1、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。  
2、这种方式清理出来的空闲内存是不连续的。我们的死亡对象都是随机的出现在内存的各个角落，把它们都清除之后，内存的布局自然会乱七八糟，为了应付这一点，JVM就不得不维持一个内存的空闲列表，这又是一个开销。而且在分配数组对象的，寻找连续的内存不好找

**④标记整理：**

分为标记和整理两个阶段：首先标记出所有需要回收的对象，让所有存活的对象都向一端移动，然后直接清理掉边界意外的内存。  
优点：不会产生空间碎片，但是整理会需要一点时间  
地方：适合老年代进行垃圾收集，parallel Old（针对parallel scanvange gc的） gc和Serial old收集器就是采用该算法进行回收的。

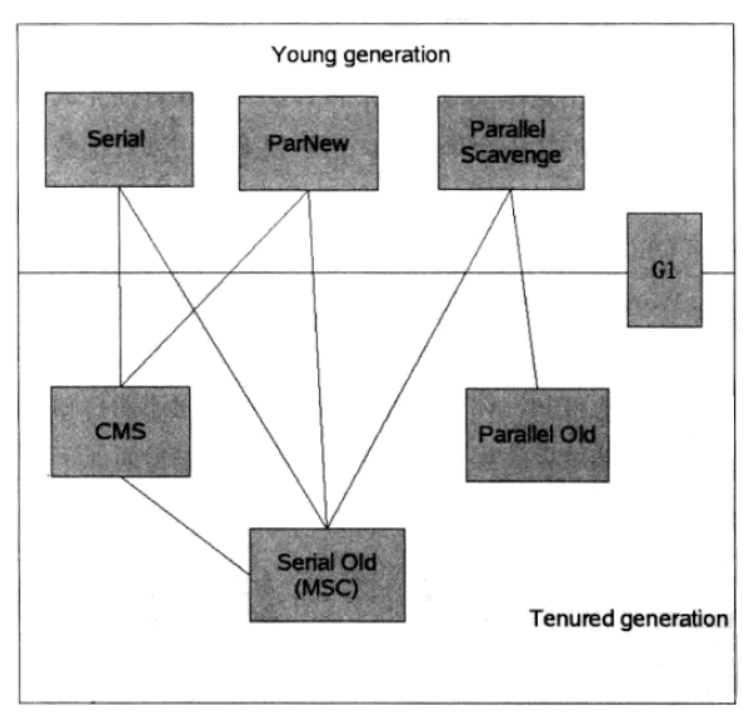
**新生代：“朝生夕死”，存活率低，使用复制算法。**

**老年代：存活率较高，使用“标记-清除”算法或者“标记-整理”算法**

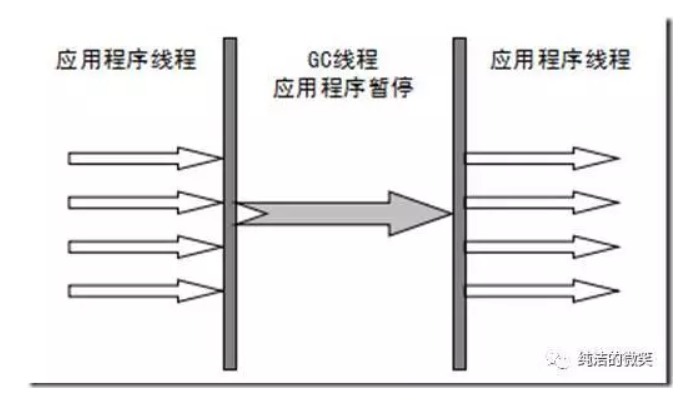
**常用的垃圾收集器**

**如果说收集算法是内存回收的方法论，垃圾收集器就是内存回收的具体实现**

**见下图：**

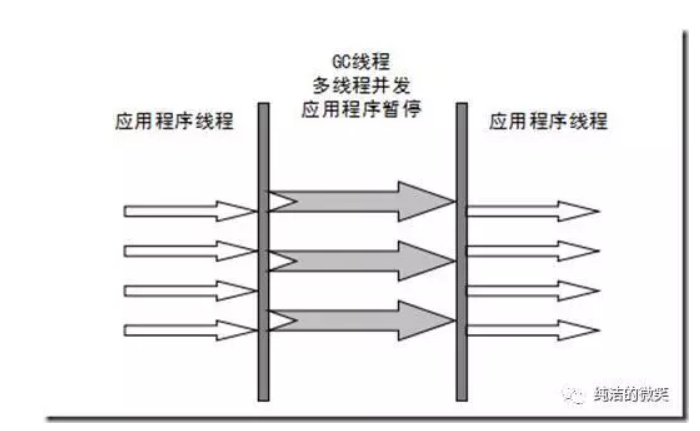


**Serial收集器**  
串行收集器是最古老，最稳定以及效率高的收集器，可能会产生较长的停顿，只是用一个线程去回收。新生代、老年代使用串行回收；新生代复制算法、老年代标记 - 压缩；垃圾收集的过程中会Stop The World(服务暂停)  
参数控制： -XX:+UseSerialGC 串行收集器



**ParNew 收集器**

ParNew收集器其实就是Serial收集器的多线程版本。新生代并行，老年代串行；新生代复制算法、老年代标记-压缩  
参数控制：  
-XX:+UseParNewGC ParNew收集器  
-XX:ParallelGCThreads 限制线程数量



**Parallel收集器**

Parallel Scavenge收集器类似ParNew收集器，Parallel收集器更关注系统的吞吐量。可以通过参数来打开自适应调节策略，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量；也可以通过参数控制GC的时间不大于多少毫秒或者比例；新生代复制算法、老年代标记-压缩  
参数控制： -XX:+UseParallelGC 使用Parallel收集器+ 老年代串行

**Parallel Old 收集器**

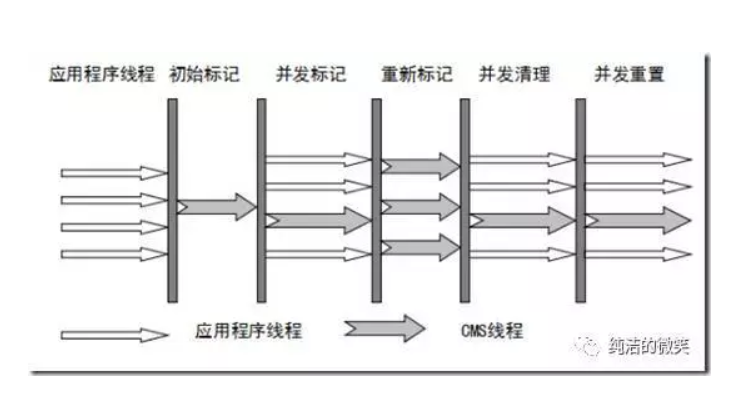
Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记－整理”算法。这个收集器是在JDK 1.6中才开始提供  
参数控制： -XX:+UseParallelOldGC 使用Parallel收集器+ 老年代并行

**CMS收集器**

CMS（Concurrent Mark Sweep）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应用都集中在互联网站或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。  
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要更复杂一些，整个过程分为4个步骤，包括：

* 初始标记（CMS initial mark）
* 并发标记（CMS concurrent mark）
* 重新标记（CMS remark）
* 并发清除（CMS concurrent sweep）

其中初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC Roots Tracing的过程，而重新标记阶段则是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。  
由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以总体上来说，CMS收集器的内存回收过程是与用户线程一起并发地执行。老年代收集器（新生代使用ParNew）  
优点: 并发收集、低停顿  
缺点: 产生大量空间碎片、并发阶段会降低吞吐量  
参数控制：  
-XX:+UseConcMarkSweepGC 使用CMS收集器  
-XX:+ UseCMSCompactAtFullCollection Full GC后，进行一次碎片整理；整理过程是独占的，会引起停顿时间变长  
-XX:+CMSFullGCsBeforeCompaction 设置进行几次Full GC后，进行一次碎片整理  
-XX:ParallelCMSThreads 设定CMS的线程数量（一般情况约等于可用CPU数量）



**G1收集器**

### GcRoot

所谓"GC roots" 或者说tracing GC的"根集合"就是一组必须活跃的引用。  
基本思路就是通过一系列名为"GC Roots"的对象作为起始点，从这个被称为GC Roots的对象开始向下搜索，如果一个对象到GC Roots没有任何引用链相连时，则说明此对象不可用。也即给定一个集合的引用作为根出发，通过引用关系遍历对象图，能被遍历到的(可到达的)对象就被判定为存活；没有被遍历到的就自然被判定为死亡

**哪些可以作为GC Roots的对象：**  
虚拟机栈(栈帧中的局部变量区，也叫做局部变量表)中引用的对象  
方法区中的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(Native方法)引用的对象

**GC是什么时候触发的（面试常问）**

由于对象进行了分代处理，因此垃圾回收区域、时间也不一样。GC有两种类型：Scavenge GC和Full GC

**Scavenge GC**

一般情况下，当新对象生成，并且在Eden申请空间失败时，就会触发Scavenge GC，对Eden区域进行GC，清除非存活对象，并且把尚且存活的对象移动到Survivor区。然后整理Survivor的两个区。这种方式的GC是对年轻代的Eden区进行，不会影响到年老代。因为大部分对象都是从Eden区开始的，同时Eden区不会分配的很大，所以Eden区的GC会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使Eden去能尽快空闲出来

Full GC

对整个堆进行整理，包括Young、Tenured和Perm。Full GC因为需要对整个堆进行回收，所以比Scavenge GC要慢，因此应该尽可能减少Full GC的次数。在对JVM调优的过程中，很大一部分工作就是对于Full GC的调节。有如下原因可能导致Full GC：

a) 年老代（Tenured）被写满

b) 持久代（Perm）被写满

c) System.gc()被显示调用

d) 上一次GC之后Heap的各域分配策略动态变化