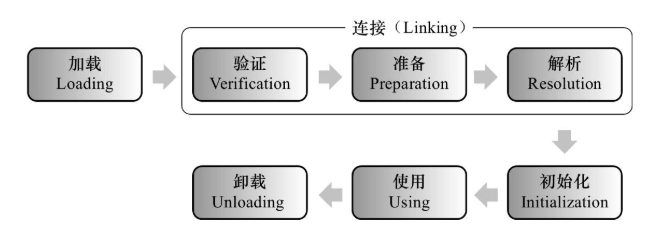
## 类的加载：



加载、验证、准备、初始化、和卸载这 5 个阶段的顺序是确定的。而解析阶段不一定：它在某些情况下可以在初始化阶段之后再开始，这是为了支持 Java 的运行时绑定

加载：

类加载是将类的Class文件中的二进制数据读入到内存中，放在方法区中，然后在堆上创建对应的Class对象，用来封装方法区内的数据结构。Class对象向java程序提供访问方法区数据的接口(类加载器)

加载类的方式：

从本地系统中直接加载

通过网络下载.class文件

从zip，jar等归档文件中加载.class文件

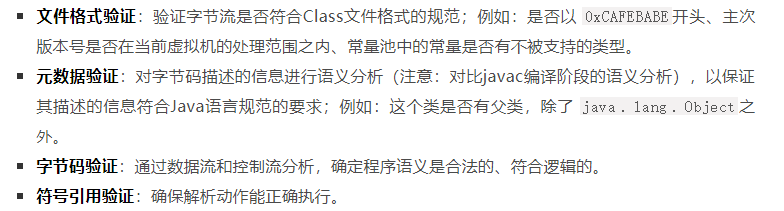
从专有数据库中提取.class文件

将Java源文件动态编译为.class文件

动态代理

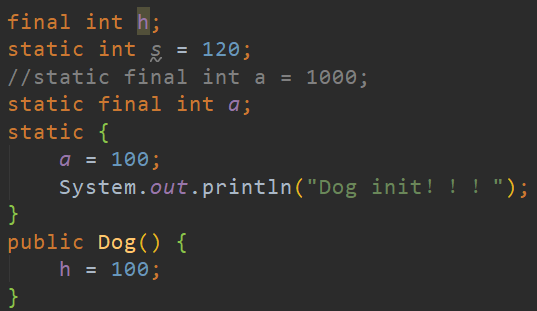
验证： 确保被加载的类的正确性

确保Class文件的字节流中包含的信息符合JVM的要求，并且不会对JVM产生危害，校验很重要，但不是必须的，可以通过-Xverifynone参数关闭校验，缩短JVM类加载的时间



准备： 为类变量分配内存，初始化默认值

为类变量(static)分配内存空间并设置类变量的初始值阶段，即在方法区(类变量从属于类)中分配变量所使用的内存空间(static int i = 10000，实际上i此阶段是0，赋值阶段在类构造器<cilent>方法中，只是i占用的内存为int)，但是static final i = 1000，在准备阶段就是1000了，static属于类，可变，赋值可以在构造函数中进行，而final属于不可变，只能在声明的时候就确定



final从属于对象，final初始化在构造对象之前都可以初始化，

static从属于类，在类能被访问(类被初始化)之前都可以初始化，

System.out.println(Dog.a)：如果a在声明时就初始化，就不会初始化类(执行static代码块)，如果a的初始化在static代码块中，就会初始化代码块，而输出s无论在哪里初始化s都会初始化类；在输出a时，如果a没值，JVM就会初始化

解析：

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程，解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。符号引用就是一组符号来描述目标，可以是任何字面量。直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

初始化：

类加载的最后一个阶段，此阶段之前都是JVM主导(即类已经加载到内存中)，此时才开始执行类中的程序，初始化阶段是执行类构造器<client>方法(不是构造函数)的过程。<client>方法是由编译器自动收集类中的类变量的赋值操作和静态语句块中的语句合并而成的。对类变量进行赋值，解析完后，class文件已经在JVM内存中

触发类初始化的操作：

1. new
2. 访问类的静态变量(准备阶段)
3. 调用静态方法
4. 反射 Class.forName
5. 初始化子类，父类也初始化
6. JVM在启动时被标明为启动类的类

不触发类初始化的操作：

1，子类应用父类静态字段，只会触发父类初始化，不会触发子类

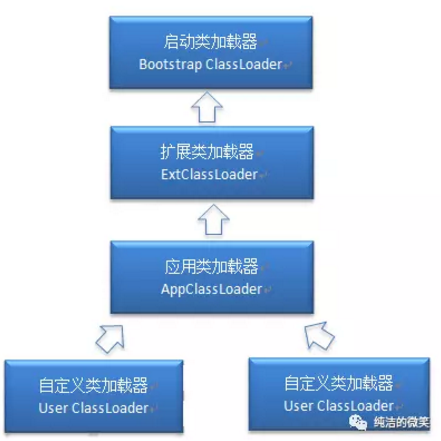
2，同归类名获取Class对象，不会触发类的初始化

3，通过Class.forName加载指定类时，如果指定参数initialize为false时，也不会触发类初始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。

4，通过ClassLoader默认的loadClass方法，也不会触发初始化动作

5，常量在编译期间会存入调用类的常量池中，本质上并没有直接引用定义常量的类，不会触发定义常量所在的类。

类加载器



启动类加载器：

BootstrapClassLoader，负责加载存放在 JDK\jre\lib(JDK代表JDK的安装目录，下同)下，或被 -Xbootclasspath参数指定的路径中的，并且能被虚拟机识别的类库（如rt.jar，所有的java.开头的类均被 BootstrapClassLoader加载）。启动类加载器是无法被Java程序直接引用的(BootstrapClassLoader是由C++实现的)

扩展类加载器：

ExtensionClassLoader，该加载器由 sun.misc.Launcher$ExtClassLoader实现，它负责加载 JDK\jre\lib\ext目录中，或者由 java.ext.dirs系统变量指定的路径中的所有类库（如javax.开头的类），开发者可以直接使用扩展类加载器。

应用程序类加载器：

ApplicationClassLoader，该类加载器由 sun.misc.Launcher$AppClassLoader来实现，它负责加载用户类路径（ClassPath）所指定的类，开发者可以直接使用该类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。它负责将 用户类路径(java -classpath或-Djava.class.path变量所指的目录(即classes目录下)

JVM类加载机制：

全盘负责：当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入

父类委托：先让父类加载器试图加载该类，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类

缓存机制：缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效

类加载的方式：

启动应用程序，由JVM初始化加载

Class.forName方法动态加载

ClassLoader.loadClass方法动态加载

在使用Class.forName加载类时，除了将class文件加载到JVM外，还会对类进行初始化(默认情况下，可以设置initialize为false

[Class.forName(name,initialize,loader)])

而**ClassLoader.loadClass只会加载class文件到JVM**中

类加载和类初始化是两部分，初始化在类加载完成后，加载只是将class加载到内存中，初始化只有在使用的时候才会进行

双亲委派模型：

当类加载器收到类加载请求后，首先不会尝试加载这个类，而是把这个请求委托给父类加载器完成，如果父加载器在它范围内搜索不到，子加载器才尝试加载才类，如果子类加载器也没找到，则抛出ClassNotFoundException

当 AppClassLoader加载一个class时，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器ExtClassLoader去完成。

当 ExtClassLoader加载一个class时，它首先也不会自己去尝试加载这个类，而是把类加载请求委派给BootStrapClassLoader去完成。

如果 BootStrapClassLoader加载失败（例如在 $JAVA\_HOME/jre/lib里

未查找到该class），会使用 ExtClassLoader来尝试加载(ExtClassLoader在$JAVA\_HOME/jre/lib/ext里查找)，如果找到了，会返回对应类的Class instance

若ExtClassLoader也加载失败，则会使用 AppClassLoader来加载，如果 AppClassLoader也加载失败，则会报出异常 ClassNotFoundException。

双亲委派的意义：

系统类防止内存中出现多份同样的字节码

避免有些类别重复加载

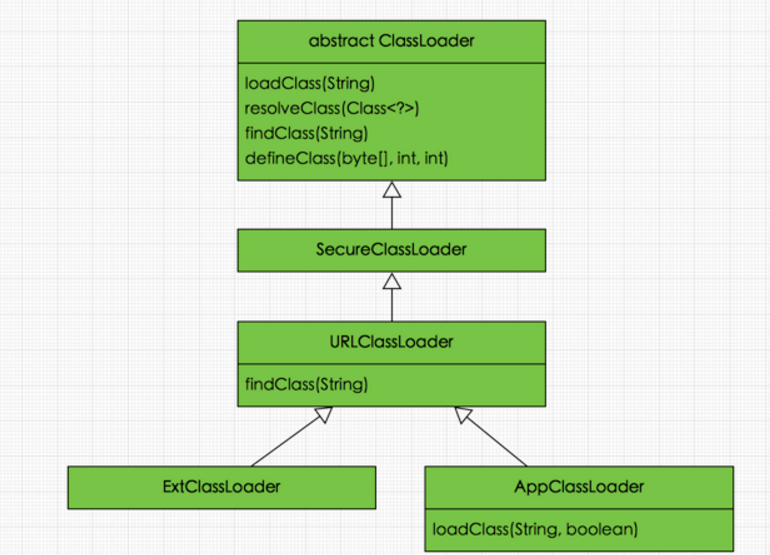
保证Java程序安全稳定运行

双亲委派模型很好的解决了各个类加载器加载基础类的统一性问题。(即越基础的类由越上层的加载器进行加载。)

安全：java核心api不会被随意替换，如自定义一个java.lang.Integer类，双亲委派，会先加载系统(BootStrap ClassLoader)的，就不会加载自定义的了

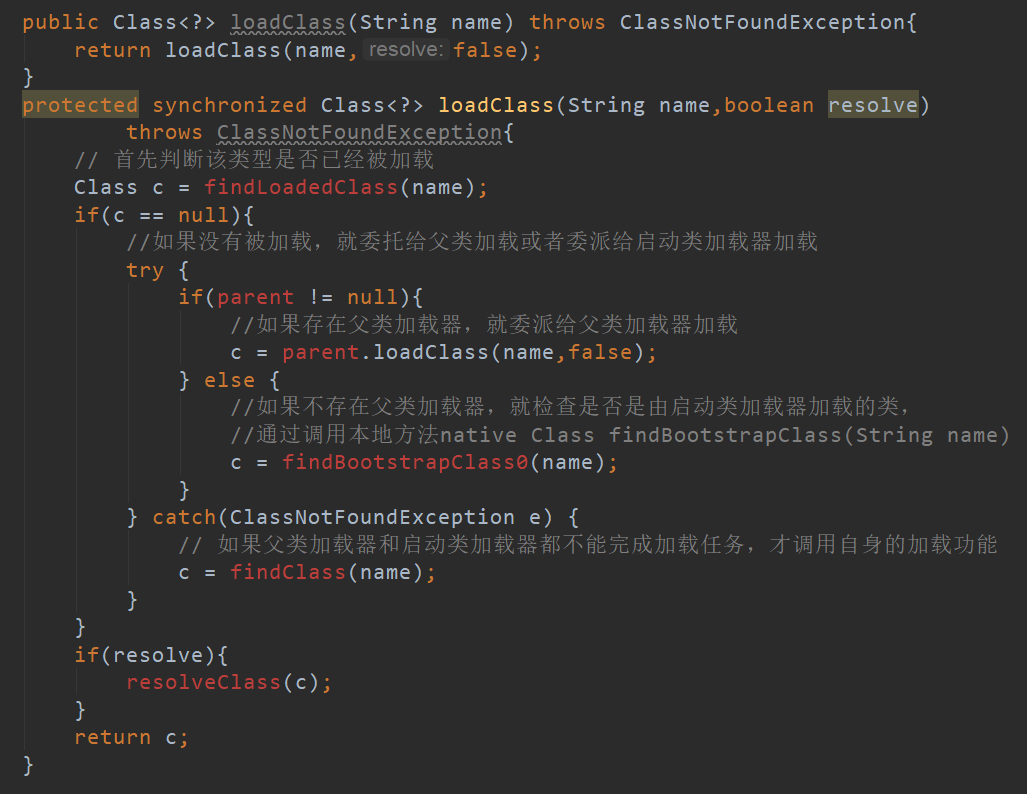
自定义System类(这里的类路径和系统System的完全一样，java.lang.System)，是不会加载自定义的类的，因为双亲委派

ClassLoader结构图



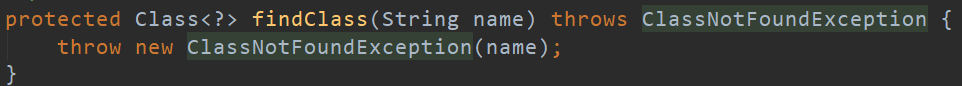
loadClass(String，Boolean)

根据类名称获取对应的Class，此方法实现的就是双亲委派，不建议重写，resolve参数代表是否生成class对象的同时进行解析相关操作



findClass(String)：实现类对findClass重写(URLClassLoader)

在自定义类时，不建议重写loadClass，而建议将自定义类加载器的逻辑在findClass方法中实现，默认findClass是直接抛出异常



defineClass(byte[] b, int off, int len)

将byte子节流解析为JVM能识别的Class对象(ClassLoader中已实现)，一般在自定义类加载器的时候，在自定义的findClass中获取到类的字节码后，使用defineClass转化为Class(这里只是转换，并为解析，解析在准备阶段)



resolveClass(Class c)

该方法对Class对象进行解析(准备阶段)

SecureClassLoader

ClassLoader的扩展，主要新增了对class的访问权限以及证书的验证，一般直接使用即可，更多的是使用其子类URLClassLoader

URLClassLoader

ClassLoader是一个抽象类，实现的方法比较少(loadClass， resolveClass等)，URLClassLoader对ClassLoader中的方法进行实现(findClass，findResource等)，在自定义类加载器的时候，可以继承URLClassLoader(即findClass符合自己的需求)，在URLClassLoader类中有一个URLClassPath类，URLClassPath类负责找到要加载的字节码，再读取成字节流，最后通过defineClass方法创建Class对象，所以必须传入一个URL参数，代表字节码文件的路径。在创建URLClassLoader对象时，会根据URL的路径判断是class文件还是jar，然后创建FileLoader或者JarLoader或默认的Loader类去加载相应路径下的class文件，当JVM调用findClass方式时，将class文件的字节流加载到内存中，最后创建为Class对象；在自定义类加载器的时候，如果继承ClassLoader就需自己实现findClass以及获取字节码流的逻辑

ExtClassLoader/APPClassLoader

这两个类加载器都是URLClassLoader的子类，并且是Launcher类的静态内部类，Launcher被系统用于启动主应用程序，ExtClassLoader/AppClassLoader都被创建

Launcher是单例的，ExtClassLoader/AppClassLoader在JVM中只有一个

同一个类在JVM中出现两个Class对象情况：加载这个类时，使用了两个ClassLoader(使用ExtClassLoader加载类A，使用APPClassLoader加载类B)，但是这种情况一般不会出现，因为有双亲委派的存在，除非直接调用类加载器的findClass方法，跳过loadClass(双亲委派)方法

自定义类加载器：

类不在classpath路径下时(并且必须要不在classpath下，如果在就会调用父类APPClassLoader进行加载)

Class文件是一个网络传输的，可能会涉及对class进行解密的操作

热部署，利用同一个class文件，在不同的类加载器在内存中创建两个class对象(只能直接使用findClass获取)

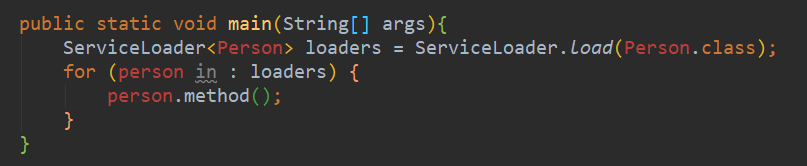
实现自定义类加载器需继承ClassLoader或URLClassLoader，继承ClassLoader需要自己实现findClass和转换字节流的方法(defineClass)

**自定义类加载器时，只需继承ClassLoader，然后重新findClass，找到类的位置，然后做一些操作，然后将二进制文件交给defineClass，转换为Class**

破坏委派双亲：线程上下文类加载器

SPI机制：java提供了很多服务提供者接口(service provider interface，SPI)，第三方可以为这些接口提供实现方法(如JDBC，JNDI等)，第三方类只需要在WEB-INF下指定类的类路径

例如：定义一个借口Person，Person的实现有WhitePerson，BlackPerson，并在Person实现类所在的程序的/META-INF/services/创建一个文件，将Person实现类的类路径写入到此文件中，将此程序打包person.jar，在另一个程序中引入此jar包(Person接口是当前类的，而其他类只是实现Person接口，然后当前类再使用Person接口的实现，然后当前类调用Person接口的方法)

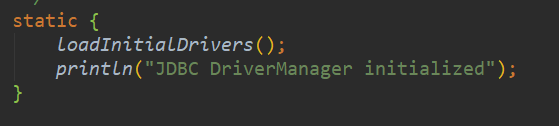


这样就可以引入Person的实现类(和一般的导入包，SPI机制会自动实例化META-INF/services下的类，而通过导入的需要自己手动实例化)

在JDBC4之前，在获取Connection之前需要执行Class.forName加载数据库驱动，而在JDBC4之后，不用Class.forName加载驱动了，而直接使用

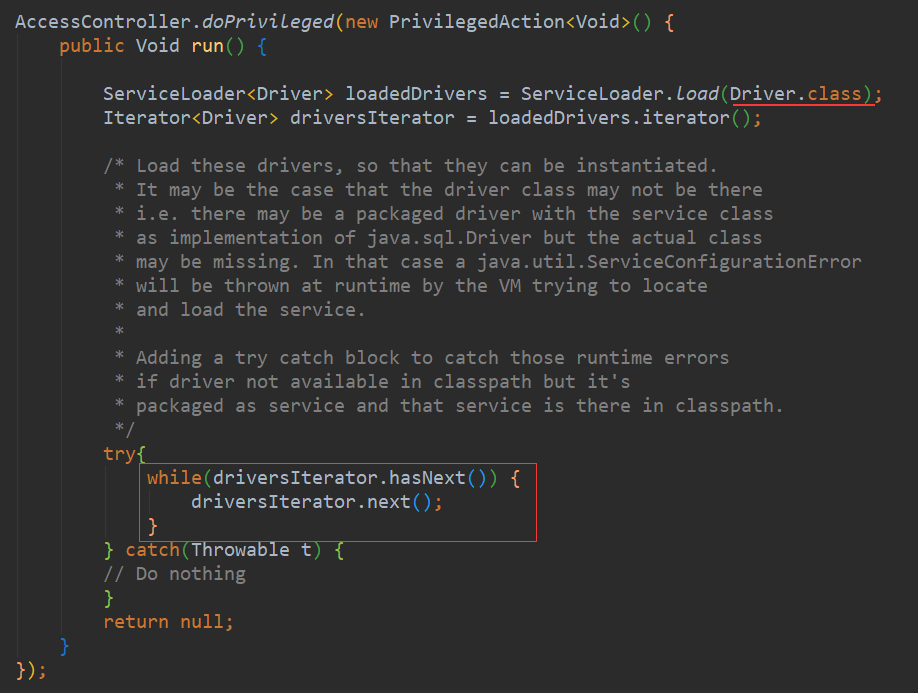
Connection con = DriverManager.getConnection(url,user,password)就可以获取conn，Class.forName的作用是将MySQL驱动的类加载到JVM中(以MySQL为例，连接不同类型数据库只需修改jar包)

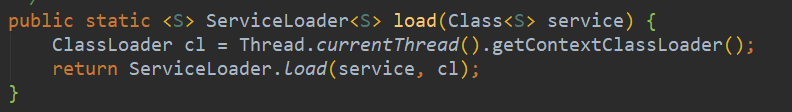
在执行DriverManager.getConnection时，会加载DriverManager(java.sql下的类)，JVM启动的时候会加载(Bootstrap CLassLoader)DriverManager

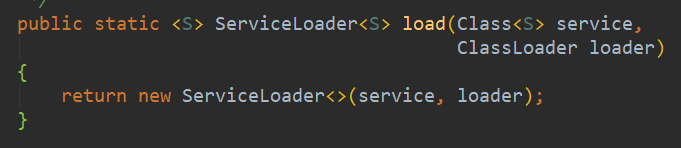


在DriverManager中有一个静态语句块，会执行loadInitialDrivers

在loadInitialDrivers有这么一段方法



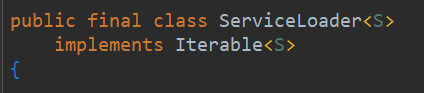




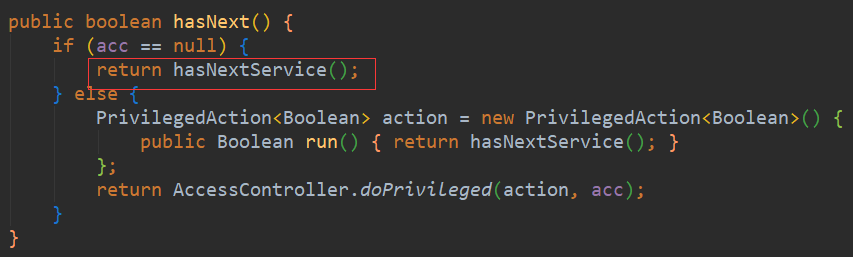
先会创建一个ServiceLoader(Driver，ClassLoader)，Driver是SPI接口，ClassLoader是线程上下文类加载器(AppClassLoader)

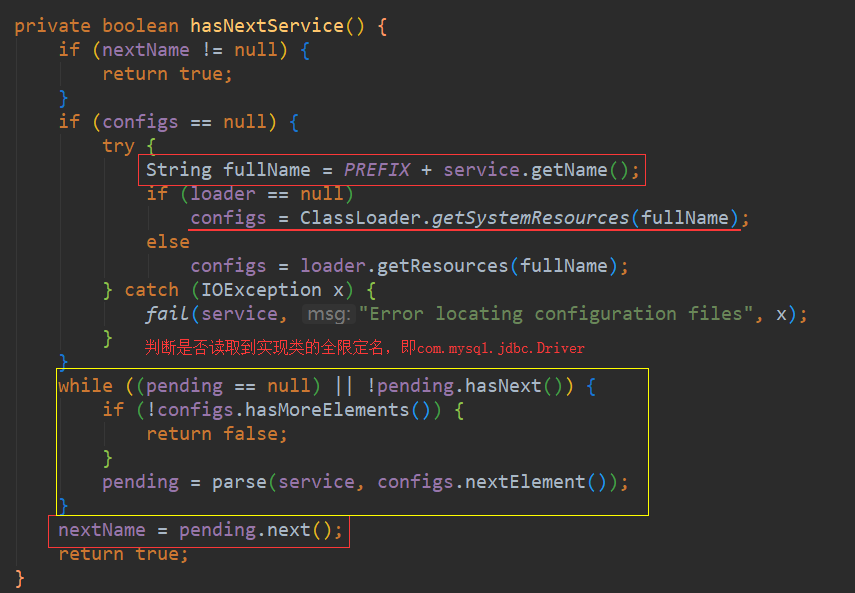
While循环就是查找具体的provider(就是在META-INF/services/\*\*\*.Driver文件中查找具体的实现)

这里循环是迭代器，而又是迭代的ServiceLoader，



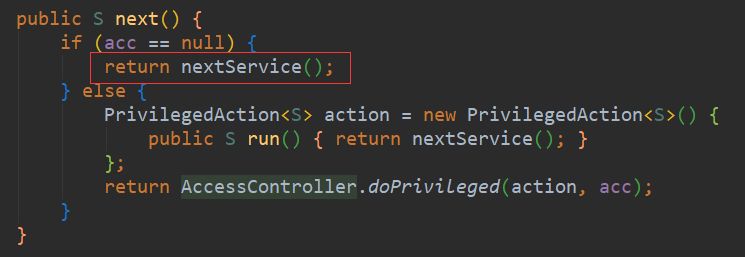
**ServiceLoader**继承了Iterable，并且重写了next和hasNext方法

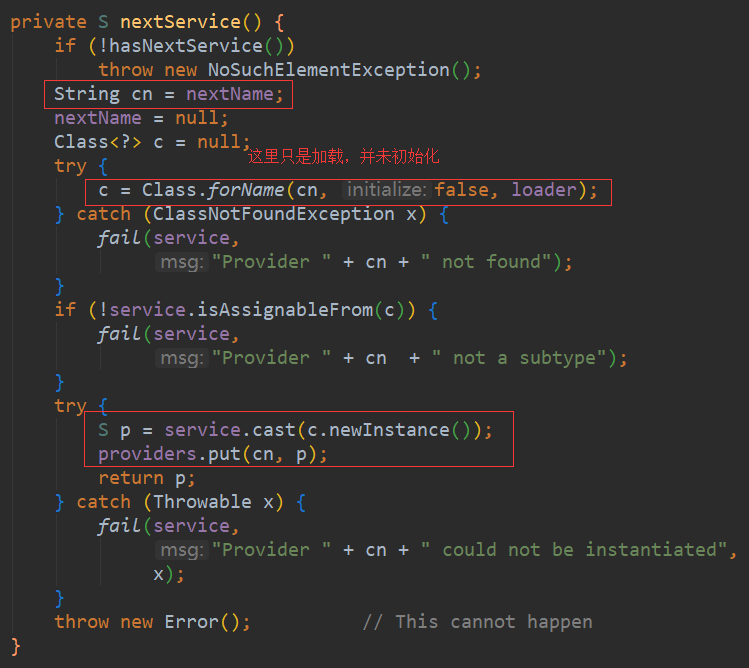






先读取classpath中META-INF目录的文件，也就是读取服务提供者的实现类权限定名，保存到config中，然后将要初始化的类赋值到nextName中





调用c.newInstance()就会实例化对应的提供者(这里MySQL的Driver)，这时提供者提供的对象就加载到JVM中了，通过DriverManager.getConnection，从DriverManager的provider容器中获取

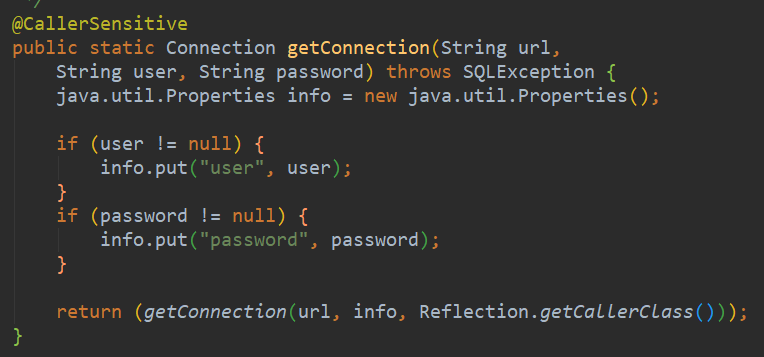
使用TCCL(线程上下文类加载器)校验实例的归属

在调用c.newInstance()后，调用MySQL的静态语句块

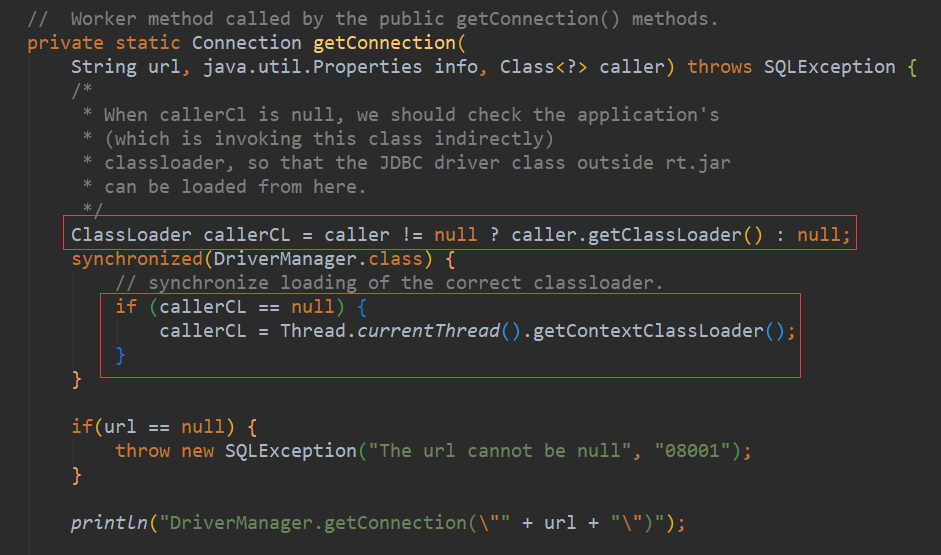


将MySQL的driver注入到java.sql.DriverManager的registeredDrivers成员变量中(registeredDrivers是一个CopyOnWriteArrayList)

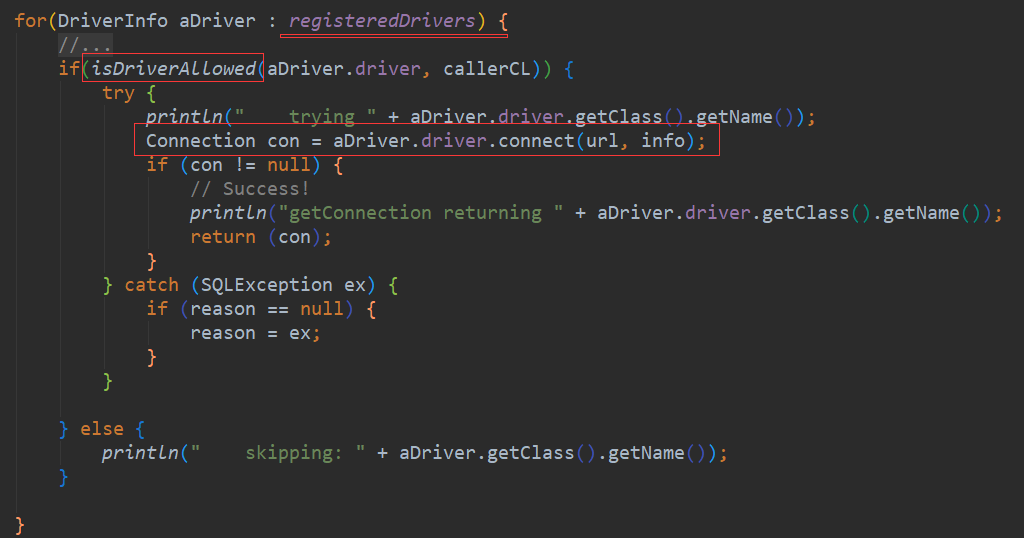
获取时，调用getConnection方法



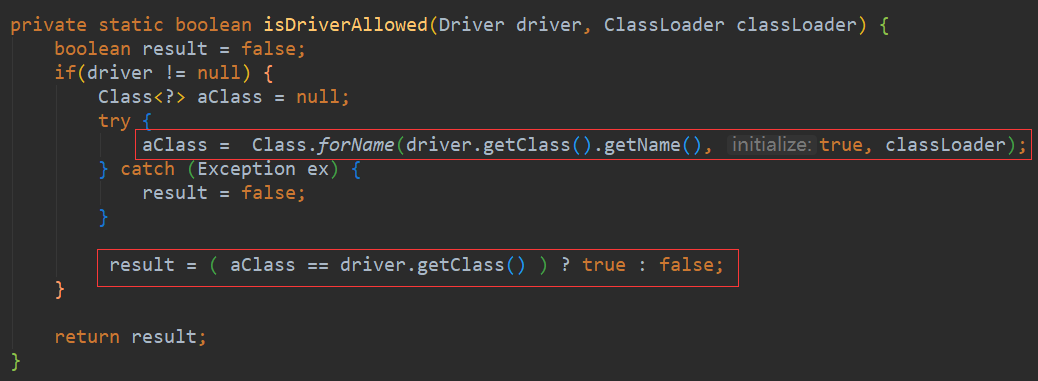
Reflection.getCallerClass()：可获取到调用本方法的Class类，这儿调用者是java.sql.DriverManager（位于/lib/rt.jar中）



caller.getClassLoader()本应得到Bootstrap启动类加载器

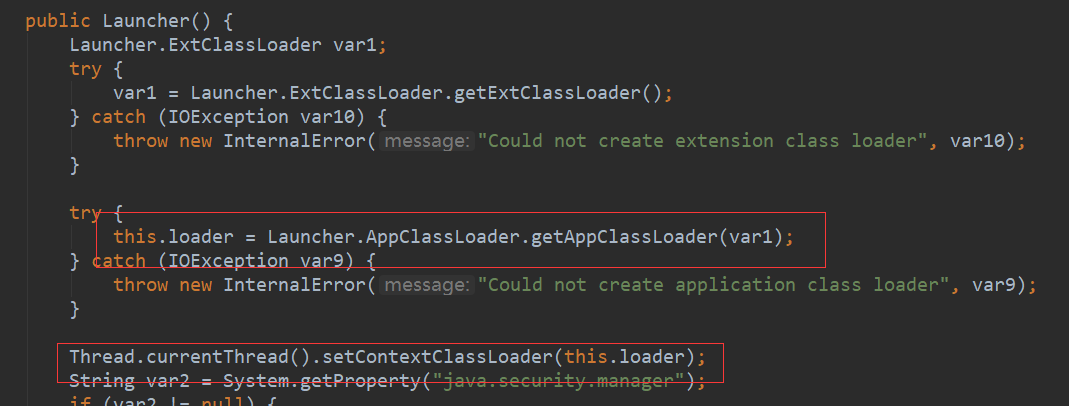


从registeredDrivers中获取Driver，isDriverAllowed校验Driver是否是自己需要的



校验方法：只有同一个类加载器中的Class使用==比较时才会相等，此处就是校验用户注册Driver时该Driver所属的类加载器与调用时的是否同一个

破坏双亲委派：DriverManager是由BootStrapClassLoader加载的，而具体provider的实现类又是由APPCLassLoader加载，所以设置了线程上下文类加载器，在整个过程中都可以调用APPClassLoader(线程上下文的类加载器在**Launcher**初始化的时候创建)，这里使用APPClassLoader加载provider，已经是BootStrapClassLoader加载不了，交由子类APPClassLoader加载了



类加载器之间的关系

启动类加载器，由C++实现，没有父类。

拓展类加载器(ExtClassLoader)，父类加载器为null，实际上就是启动类加载器

系统类加载器(AppClassLoader)，父类加载器为ExtClassLoader

自定义类加载器，父类加载器肯定为AppClassLoader

注意：这里的父子关系不是指继承

加载过程：

在执行main方法时，会调用java.exe完成：

1. 根据JVM内存配置要求，为JVM申请特定大小的内存空间；

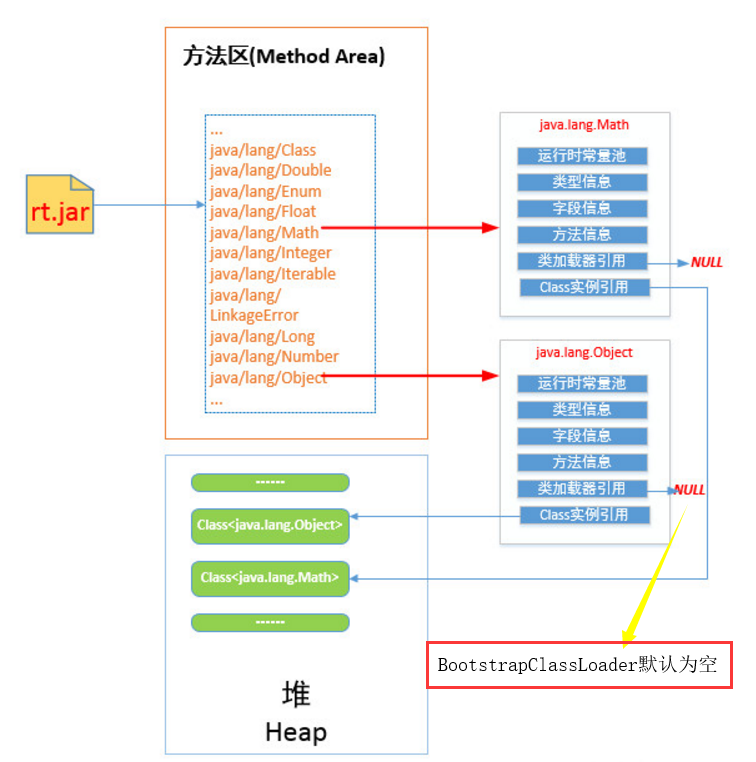
分配堆，栈，方法区

1. 创建一个引导类加载器实例，初步加载系统类到内存方法区区域中；

内存空间分配好后，JVM会创建一个引导类加载器(Bootstrap ClassLoader)实例(C++实现)，加载{JRE\_HOME}/lib下的系统类



引导类加载完后，JVM内存的格局



首先会在方法区创建对应Class的数据结构，再堆中创建对应的Class对象，Class对象作为访问方法区数据的接口

1. 创建JVM 启动器实例 Launcher,并取得类加载器ClassLoader；

现在JVM的基本运行环境准备就绪，开始执行自己的程序，让JVM工作起来

首先JVM会调用已加载到方法区的系统类sun.misc.Launcher的静态方法getLauncher获取Launcher实例(在Launcher下定义了两个类加载器ExtClassLoader和APPClassLoader)，然后调用launcher.getClassLoader()获取到APPClassLoader

4,使用上述获取的ClassLoader实例加载我们定义的 Main类；

获取到ClassLoader后，进行双亲委派加载Main类，在将Main类加载到内存后，对Main的Class文件进解析(解析阶段)时，发现Main的Class文件的常量池(Constant Pool)中有其他类的引用，然后加载其他依赖的类，最后解析Main类，最后在方法区中，不同的类加载器管理着这些class

JVM方法区的类信息区是按照类加载器进行划分的，每个类加载器会维护自己加载类信息；

1. 加载完成时候JVM会执行Main类的main方法入口，执行Main类的main法；
2. 结束，java程序运行结束，JVM销毁。

## Java内存模型：

程序计数器：线程私有

用于记录(保存)当前线程正在执行的字节码的地址

如果正在执行的是native方法，程序计数器为空(native调用的是OS方法，而程序计数器数为java线程服务)

程序计数器每个线程私有，这是一个唯一不会出现OutOfMemoryError的内存区域

Java虚拟机栈：线程私有

java虚拟机栈使用描述java方法运行过程的内存模型

java 虚拟机栈会为每一个即将运行的 Java 方法创建一块叫做 “栈帧” 的区域

用于存储方法的局部变量表，基本数据类型，引用数据类型，返回值等

方法执行完毕，方法对应的栈帧将会出栈，并释放内存空间

java虚拟机栈线程私有

java虚拟机栈的生命周期和线程相同

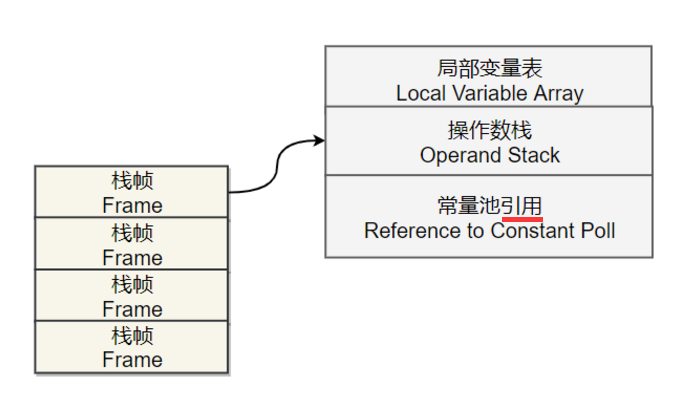
方法所占用的栈是虚拟机栈中局部变量表部分

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）、对象引用（reference 类型，它不等同于对象本身，根据不同的虚拟机实现，它可能是一个指向对象起始地址的引用指针，也可能指向一个代表对象的句柄或者其他与此对象相关的位置）

除了Long和double占用2个局部变量空间(Slot),其余的数据类型都只占用一个，**局部变量表的大小在编译期间就确定**了，当进入一个方法后，这个方法所需在栈中分配多大的局部变量空间不是确定的，栈的大小在编译时就确定了，运行器件是不会改变局部变量表的大小

如果线程请求的栈深度超过JVM允许的深度(方法内部不断的调用方法[死循环])，会抛出StackOverflowError异常(**超过栈的深度**)

当无法申请足够的内存时，会抛出OutOfMemoryError异常(疯狂的创建线程)(**创建栈过多**)



操作数栈：保存方法执行原语过程中的结果

修改JVM栈空间大小：java –Xss512M HackTheJava

本地方法栈：线程私有

和虚拟机栈的功能类似，虚拟机栈执行的是java方法，而本地方法栈则是为java

中native方法提供的栈空间，本地方法栈也会抛出StackOverflowError和OutOfMemorError

Java堆：非线程私有

Java对是java虚拟机所管理内存中最大的一部分，java堆被所以线程共享，堆用于存放对象实例(new Object)，堆是垃圾收集器管理的主要区域。从内存回收角度看，由于现在收集器基本都是采用分代收集算法，所以java堆可以细分为新生代和老年代，再细分点有Eden空间，From Survivor空间，To Survivor空间；从内存分配角度看，线程共享的堆可以划分为多个线程私有的分配缓冲区(Thread Local Allocation Buffer,TLAB)，不过无论如何划分，都与内存内容无关，无论哪个区域，存储的任然是java对象实例，进一步划分是为了更好的回收和分配内存，使用“-Xms”与“-Xmx”控制堆的最小与最大内存，但是扩展动作由虚拟机执行，很消耗性能，所以一般最大和最小内存设为相等

修改JVM堆大小：java –Xms512M –Xmx512M HackTheJava

Java堆的物理内存空间可以不连续，只要逻辑上连续即可，堆内存会抛出OutOfMemoryError异常

**栈是运行时的单位，而堆是存储的单位。**

方法区：

和堆内存一样，线程共享，用于存放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，方法区就是java堆的一部分，只是为了和堆区分，方法区还叫永久代，永久代这个概念只存在于HotSpot虚拟机中(因为HotSpot虚拟机把GC分代收集扩展至方法区)，方法区的内存也是不连续的(不连续分配灵活，但是速度慢，连续分配速度快，但是不灵活)

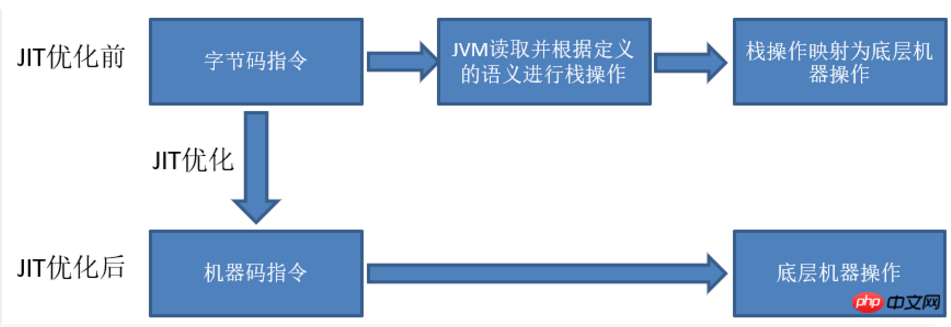
方法区存储的信息：

加载的类字节码：类的字节码加载到JVM中

class/method/field等元数据对象：字节码加载后，JVM会根据字节码(二进制)生成对应的class/method/field等对象，用于描述一个类，在反射中用得较多

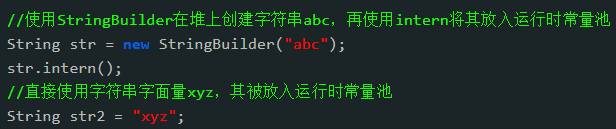
Static-final常量、static变量：JVM会在方法区创建一份数据，所以一个类的static修饰的类成员只有一份

Jit编译器的编译结果：JIT将字节码编译为机器码，通常情况下，JVM执行字节码时，会将其按照预先定好的规则执行栈操作，而栈操作会进一步映射为底层机器操作指令，而通过JIT编译后，可以直接和底层打交道，而JIT编译后的结果会存放到方法区中



相当于是之前每次都已进栈操作，而现在可以换成为对应底层操作

在方法区中还有一个运行时常量池，常量池用于存放字节码中的字面量(如字符串常量)，类加载时，字面量进入运行时常量池存放，在运行时也可以向运行是常量池中写入字面量



Java虚拟机对方法区的现在非常松，除了和堆内存是一个不连续的内存空间和可以固定此区域的大小外，方法区还选择是否实现垃圾收集(就是是否能叫永久代)，方法区进行内存回收的目的是针对常量池的回收和类型的卸载(应该就是移出类信息)。

但是在JDK1.7之后，移出了方法区而使用元数据空间(metaspace)，metaspace占用的是系统内存，而不是JVM的，只要不触碰到系统最大内存，就不会导致OOM，但是还是要对方法区的大小进行限制。对方法区的回收是非常复杂且严格，回收不当会导致类失去元数据和常量信息。但是随着规模的变大，会导致方法区的内存占用越来越大(如cgLib，会不断反射，加载类)，导致OOM。

运行时常量池：方法区的一部分，class文件中除了有类的版本，字段，方法，接口等描述信息外，还有常量池，JVM对运行是常量池没有做任何细节要求，不同版本的JVM可以选择哪个内存区域实现方法区，运行时常量池和class文件常量池相比还具有动态性(可在程序运行时添加新常量，如String的intern方法)

常量池也会抛出OOM

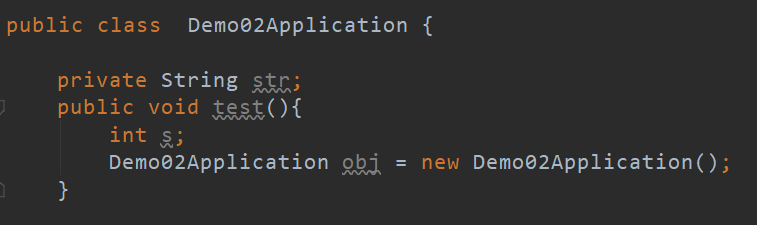
JDK1.8后，常量池从永久代中移出，在堆内存中分配

JDK1.8后，永久代被元数据空间代替，元数据空间占用系统内存

默认情况下，元数据空间的大小受本地内存限制，但是可以通过java –XX：Metaspace初始化元空间的大小，达到该值后，就会触发GC进行类型卸载，同时GC会对该值进行调整：如果释放了大量的空间就适当降低MetaSpace，如果释放少，就适当提升MetaSpace(任小于指定大小，默认没限制，就是系统内存)

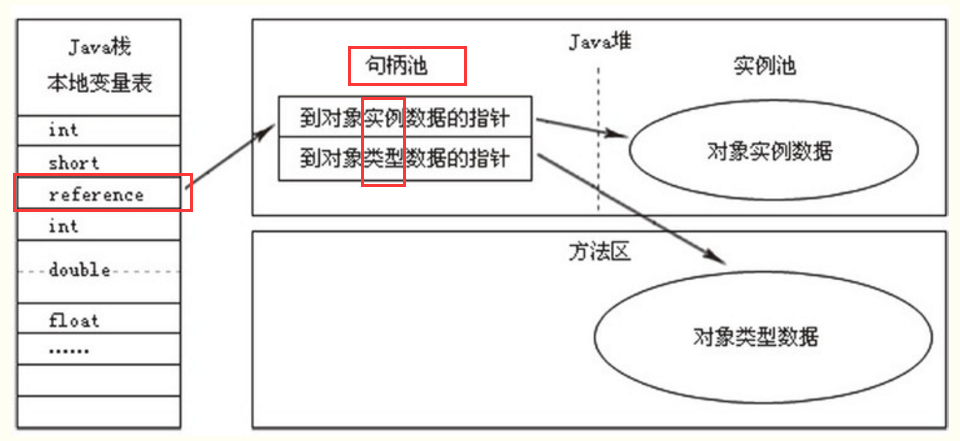
直接内存：

直接内存(Direct Memory)不是JVM运行时的数据区，但是这部分内存也频繁被使用，也会导致OOM，在java4中的NIO是一种缓存IO，它可以使用native函数直接分配堆外内存，然后通过一个存储在java堆中的DirectByteBuffer对象与堆外内存进行操作，避免java堆和Native堆中来回复制数据



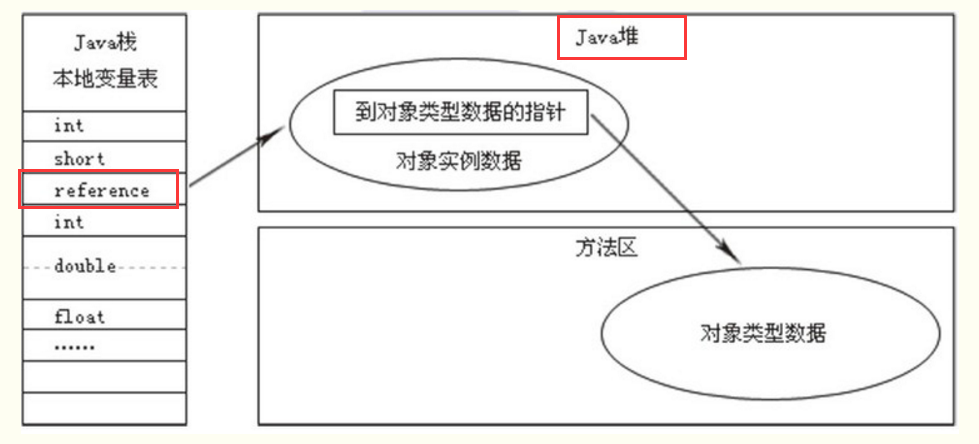
在方法中创建一个对象Demo02Application，Demo02Application类信息存在方法区中，实例存放堆中(堆中包含str属性)，应用reference保存和方法保存在栈中，三者相互关联起来可以采用句柄和直接指针的方式

句柄：



句柄是java堆的划分出来的一部分，句柄池，reference(obj)中存放的是句柄地址，句柄中包含对象实例数据和类型数据的地址

直接指针：



采用直接指针的方式，java堆对象中就会保存对象类型数据的地址，而reference保存的是堆中对象的地址

句柄的好处是reference中存储的是稳定的句柄地址，对象在被移动(垃圾收集是移动对象是普遍行为)时只会改变句柄中的实际数据指针，而reference本身不会被修改

直接指针的好处是速度快，节省指针定位的时间开销，由于对象的访问在java中非常频繁，这类开销也很大(HotSpot采用的是此方式)

各种区间OOM的情况：

堆：疯狂创建对象(new Object)

栈：疯狂的调用方法，创建线程

常量池：疯狂的执行String的intern方法(创建String)

方法区：cgLib疯狂创建动态代理类

直接内存：疯狂向堆外内存写入

Class类信息的目的是为了获取操作指令

-Xms设置堆的最小空间大小。

-Xmx设置堆的最大空间大小。

-XX:NewSize设置新生代最小空间大小。

-XX:MaxNewSize设置新生代最大空间大小。

新生代大小为堆的1/3或1/4

-XX:PermSize设置永久代最小空间大小。

-XX:MaxPermSize设置永久代最大空间大小。

-Xss设置每个线程的堆栈大小。

-XX:SurvivorRatio设置Eden和其中一个Survivor的比值

-XX:+PrintGCDetails在控制台显示GC日志

老年代空间大小=堆空间大小-年轻代大空间大小

堆内存是JVM中最大的一块由年轻代和老年代组成，而年轻代内存又被分成三部分，Eden空间、From Survivor空间、To Survivor空间,默认情况下年轻代按照8:1:1(大量数据计算出来的，80%以上的对象都是朝生夕灭)的比例来分配；永久代(方法区)

MinorGC对新生代进行回收/MajorGC对老年代进行回收

年轻代主要存放新创建的对象，内存大小相对会比较大，垃圾回收会比较频繁。年轻代分成1个Eden Space和2个Suvivor Space（命名为A和B/From Survivor，To Survivor），当对象在堆上创建时，对象占用Eden Space，如果Eden Space内存不够，直接分配到老年代，如果老年代也装不下，触发full gc，还装不下，OOM(只有当老年代装不下时，才会OOM)；新生代会频繁的创建对象，会频繁的调用MiniorGC

当执行MinorGC时，扫描Eden Space和 A Survivor Space，如果对象任存活，则复制到B Survivor Space(并存活的对象年龄+1，当达到老年代的标准后(默认15，通过-XX:MaxTenuringThreshold来设置)，复制到老年代)，清空Eden和A Survivor，最后 A Survivor和B Survivor互换(这样作的目的是确保不存在内存碎片，使得AB的空间连续，采用空间换时间来加速内存中不再被持有的对象尽快被回收)

如果B空间不足，就会复制到老年代，这属于过早提升，说明Survivor的空间小了，需要扩容，过早提升会触发调用MajorGC，而MajorGC比MiniorGC耗时，导致吞吐量下降，延迟上升(解决：直接扩容新生代空间，也可以修改Eden和Survivor的比例，但是Eden减少会增加MinorGC的频率)

Survivor空间计算公式：

survivor空间大小 = -Xmn[value] / (-XX:SurvivorRatio=<ratio> + 2)

新生代对对象的年龄进行标识，采用复制算法对空间进行回收(存活对象复制到一个Survivor，清空另外一个Survivor个Eden)，新生代采用空闲指针的方式触发GC(指针指向最后一个对象的地址，下一次有新对象时会检测空间是否够，当连续分配对象时，对象会直接到survivor，最后到老年代)

老年代比较稳定，不会频繁的执行MajorGC，在执行MajorGC前都会执行一次MiniorGC，使得有新生代进入老年代内存不够，触发MajorGC，当无法找到足够空间分配给较大对象(Eden创建大对象时，直接给老年代)时触发MajorGC

MajorGC采用标记-清除算法，先扫描一次所有老年代，标记出存活的对象，然后回收没有标记的对象。MajorGC的耗时比较长，因为要扫描再回收。MajorGC会产生内存碎片，为了减少内存损耗，需要进行合并或者标记出来方便下次直接分配，老年待采用压缩的方式避免内存碎片(将存活的对象移到内存另外一边，也就是内存整理)，压缩会影响时间，所以有些也没有对老年代进行压缩，当无法分配到连续的内存时，会提前(默认是内存大小不够时)触发一次GC

判断内存是否够用：

用一个标志记录内存可以空间，每次分配时比较

用指针指向最后一个对象的地址

老年代装不下，OOM

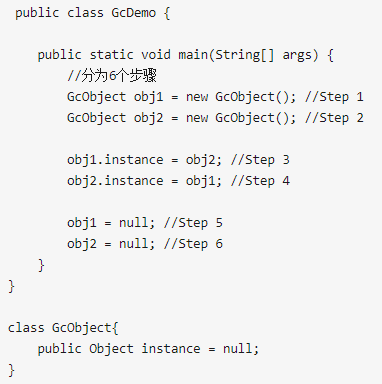
分区目的：新生代大多数都是朝生夕灭，老年代生命力强，分区方便管理，提升效率(新生代频繁GC)，优化JVM性能

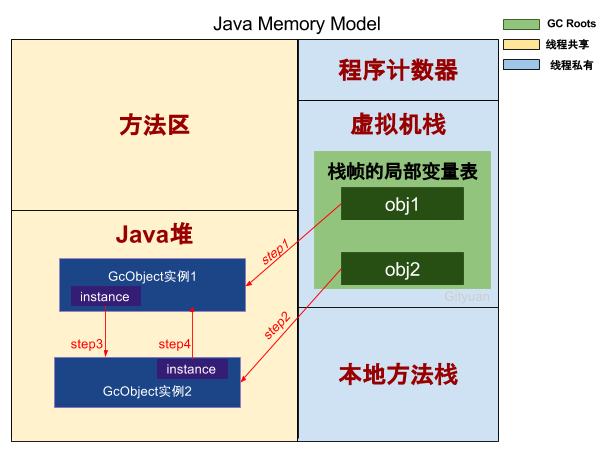
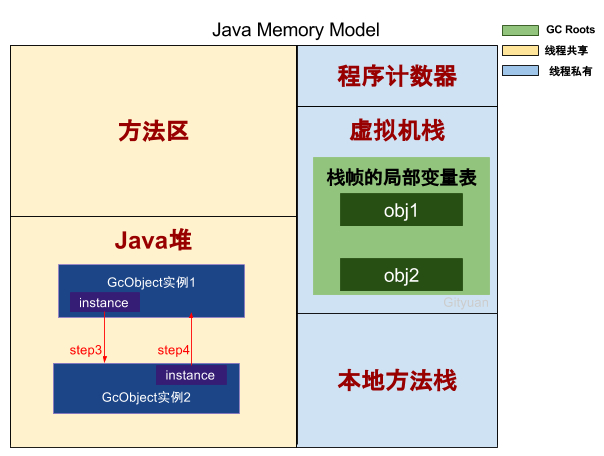
判断对象是否存活的算法：引用计数算法，可达性分析算法

引用计数：每个对象有一个引用计数器，当对象被引用一次则计数器加1，当对象引用失效一次则计数器减1，对于计数器为0的对象意味着是垃圾对象，可以被GC回收。

可达性分析：从GC Roots作为起点开始搜索，那么整个连通图中的对象便都是活对象，对于GC Roots无法到达的对象便成了垃圾回收的对象，随时可被GC回收。

引用计数每次只需判断计数器，计数器为0可回收，可达性分析每次需要遍历，从性能上来看，引用计数快，但是引用计数会导致OOM



Step5/6：栈帧中obj1不再指向Java堆，GcObject实例1的引用计数减1，结果为1；导致OOM

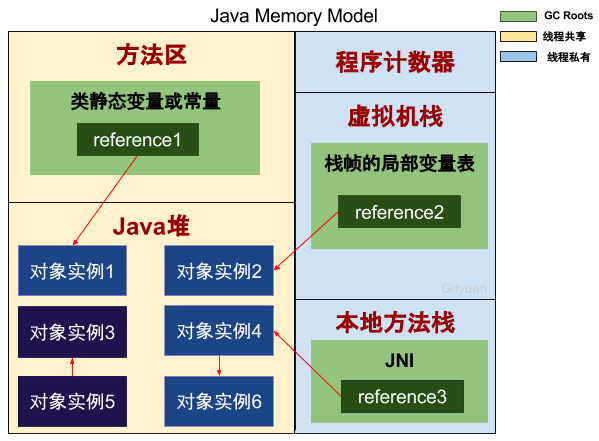
可达性分析：这是目前主流的虚拟机都是采用GC Roots Tracing算法，比如Sun的Hotspot虚拟机便是采用该算法。 该算法的核心算法是从GC Roots对象作为起始点，利用数学中图论知识，图中可达对象便是存活对象，而不可达对象则是需要回收的垃圾内存。

可作为GC Roots的对象：

虚拟机栈的栈帧的局部变量表所引用的对象；

本地方法栈(Native)的JNI所引用的对象；

方法区的静态变量和常量所引用的对象；



实例3,5不可达，所有会被GC

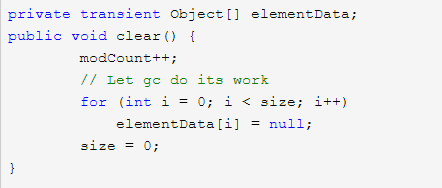
引用：

强引用：

Object obj = new Object();当内存不足时，发生OOM都不会回收强引用(有引用指向对象)，不再使用对象时，可以obj = null；帮助GC回收

在方法中创建对象，方法执行完，出栈(栈帧)，对象被回收，但是如果创建一个局部变量的对象，使用完后需要手动复制为null，否则将永远不会被GC

类似这种情况



强引用的好处，elementData私有，是个强引用，不使用时手动null，但是引用任然存在

软引用：

空间不足时，会回收软引用，空间足时不会回收；软引用可用来实现内存敏感的高速缓存。

实现方式

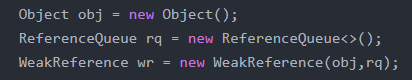


这里的softRef指向的是SoftReference对象，是一个强引用类型，SoftReference对象中引用的str就是一个软引用

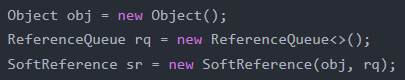
软引用的使用场景：将资源进行缓存，只有空间不足时才会去收集软引用

弱引用：

执行GC就会被回收



引用与ReferenceQueue



在软/弱引用中，实际的软引用是obj，而sr任然是一个强引用，而引用与ReferenceQueue之间的关系是，当弱引用(obj)被会收时，对应的sr将会被存放到ReferenceQueue中(在某个时候就可以去清空ReferenceQueue中的Reference)

软引用在内存空间不足时将会被回收，不管这个引用是否还指向堆中对象

弱引用在执行GC时将会被回收，不管这个引用是否还指向堆中对象

软/弱引用可以用来做缓存，当内存空间不足回收缓存中数据，空间足就不回收

虚引用：

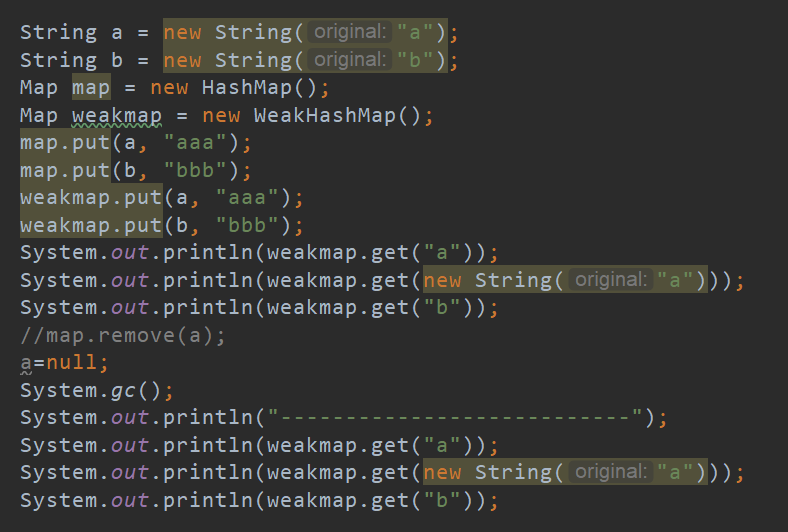
主要用来跟踪对象被GC的活动，当GC在回收对象时，发现有虚引用，就会把虚引用(PhantomReference)加入到关联的引用队列中，可以通过监控队列在引用的对象的内存在被回收前采取必要的行动(Object中finaize就是这样的)

Reference和ReferenceQueue之间的区别

Reference的构造函数可以有ReferenceQueue也可以没有，如果没有，Referenc所引用的对象被GC回收后，Reference也将会被GC回收，如果使用ReferenceQueue时，当Reference所指向的对象被GC后，Reference将被放入到对应ReferenceQueue中(Reference有4个状态：active[Reference被创建时候状态]；pending[被放进ReferenceQueue中时的状态]；enqueued[Reference被GC回收]；inactive[最终状态，对象以不存在]这说的都是Reference指向的对象)，虚引用必须有ReferenceQueue

存放到队列中用处：检查引用对象是否被回收；在ReferenceQueue中检查是是否有对应的Reference，从而判断引用是否被回收取出被回收

判断Reference所引用的对象是否被回收方式：通过Reference的get方法是否能获取到引用的对象;在队列中查看是否有对应的Reference



如果未执行map.remove，weakmap输出结果为aaa，aaa，bbb

如果执行map.remove，weakmap输出结果为null，null，bbb

原因：在执行map.remove之前，有3个指向堆中对象(new String(“a”))，分别是String a，map对象中Entery数组的K，weakmap对象中Entery数组的K，但是weakmap指向new String(“a”)是弱引用，当执行a=null后，指向new String(“a”)的使用两个，未执行map.remove时，虽然weakmap指向的是弱引用，但是任然不会回收new String(“a”)，引用还有强引用map指向，所有并不会回收new String(“a”)，但是，当执行了map.remove后，只有弱引用指向对象，所有会被回收，如果两个弱引用指向对象，然后会被回收

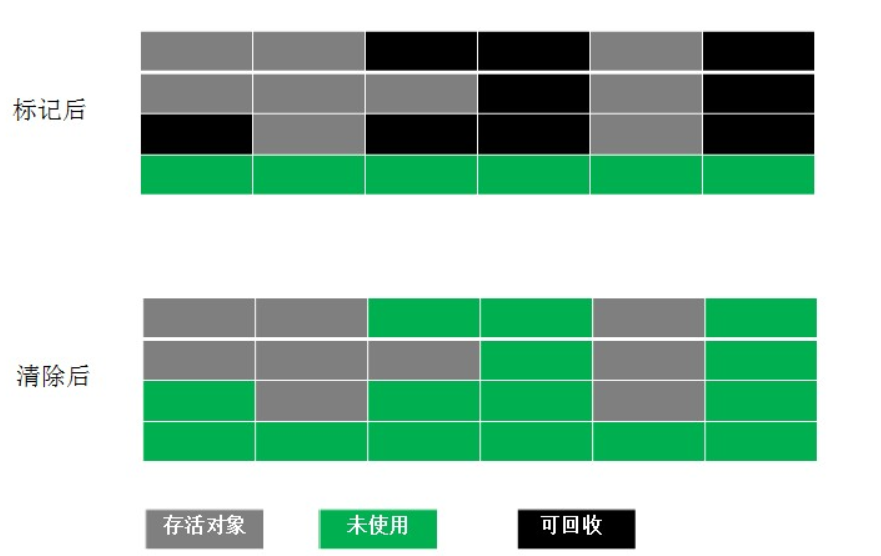
Reference是用来判断是否回收指向的对象，只当对象无强引用指向时，才会被回收

虚引用必须要有ReferenceQueue，引用执行虚引用的get方法获取到的总是null

引用在断开前，要判断是否有强引用执行被引用的对象，而不是弱引用就直接可以断开

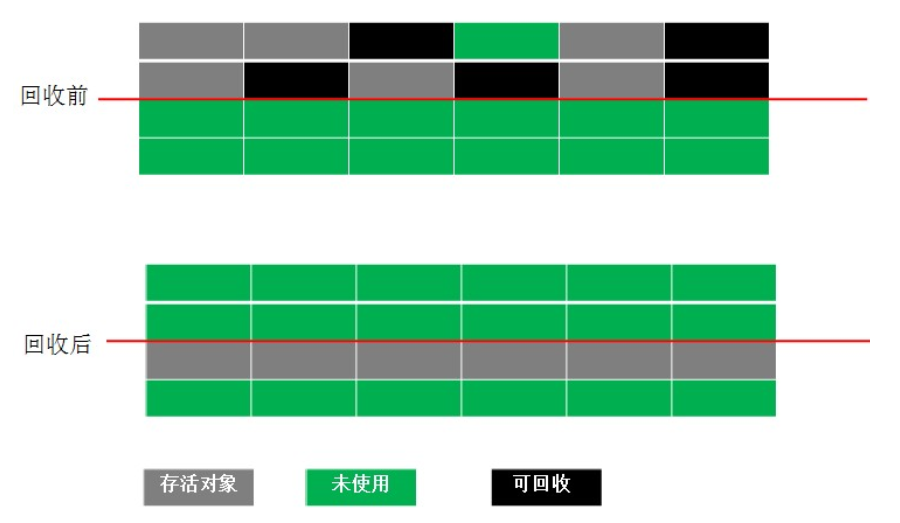
GC回收算法：标记－清除、标记－压缩、标记-复制、增量回收、分代回收

标记-清除算法(Mark-Sweep)：先标记出可回收的空间，然后回收空间，但是这样会导致内存碎片化，如果给大对象分配内存的时候就会触发full gc



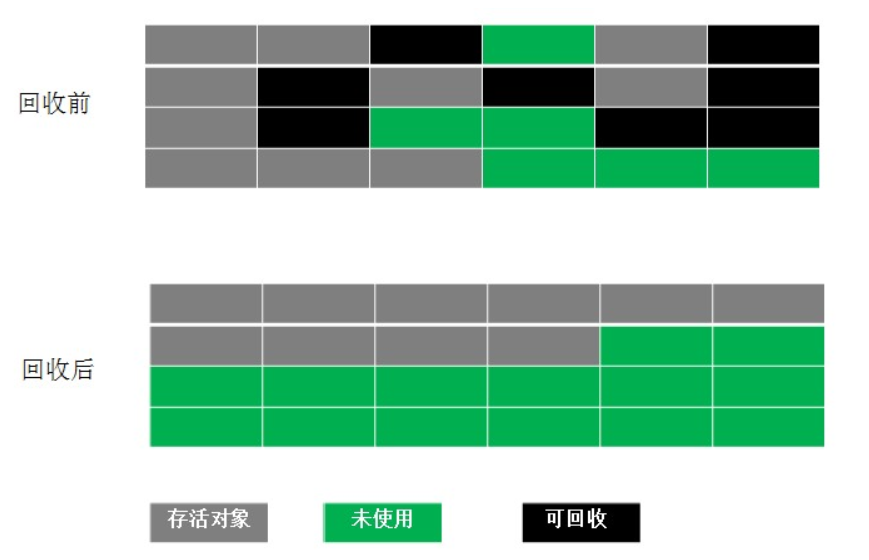
复制算法(Copying)：

将内存空间一分为二，只使用其中一块，回收时，将存活的复制到另一块未被使用的(连续)，再清空当前这块(Survivor)，复制算法的高效是建立在存活对象少，垃圾多的前提，常发生在新生代



标记-整理算法(Mark-Compact)：

在标记-清除基础上做了些优化，先标记，清除完所有的可回收空间后，对可以空间进行压缩，老年代使用的算法



GC垃圾回收器类型：

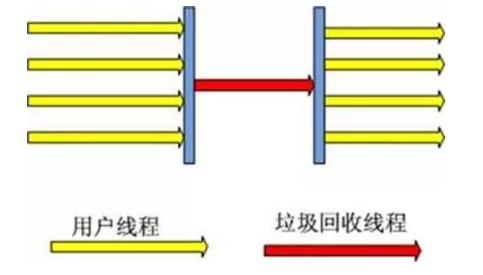
新生代收集器使用的收集器：Serial、PraNew、Parallel Scavenge

老年代收集器使用的收集器：Serial Old、Parallel Old、CMS

只有分为三类：串行回收、并发回收、并行回收，

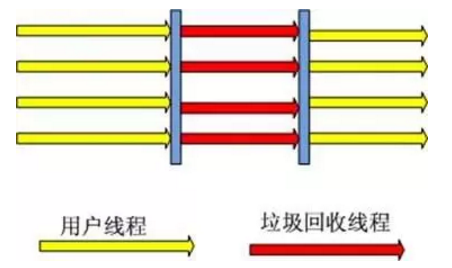
串行(Serial Garbage Collector)：

用单线程处理所有垃圾回收工作，因为无需多线程交互，所以效率比较高。但是，也无法使用多处理器的优势，所以此收集器适合单处理器机器。运行时会导致用户线程暂停



并行(Parallel Garbage Collector)：

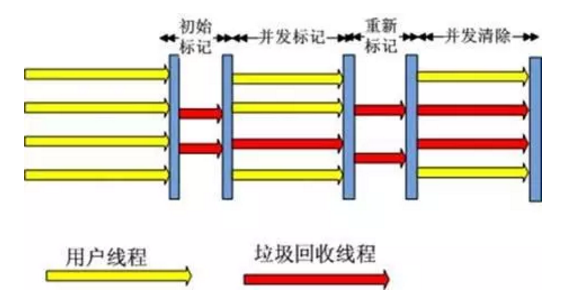
用多线程处理所有垃圾回收工作，利用多核处理器的优势。对于空间不大的区域（如young generation），采用并行收集器停顿时间很短，回收效率高，适合高频率执行。但是如果线程数量过多，导致线程之间频繁调度，也会影响性能。一般并行收集的线程是处理器的个数。



并发(CMS Garbage Collector)：

并发收集器GC时GC线程和应用线程大部分时间是并发执行，只是在初始标记（initial mark）和二次标记（remark）时需要stop-the-world，这可以大大缩短停顿时间（pause time），所以适用于响应时间优先的应用，减少用户等待时间。回收时系统程序不停止运行

回收过程：初始标记，并发标记，重新标记，并发清除

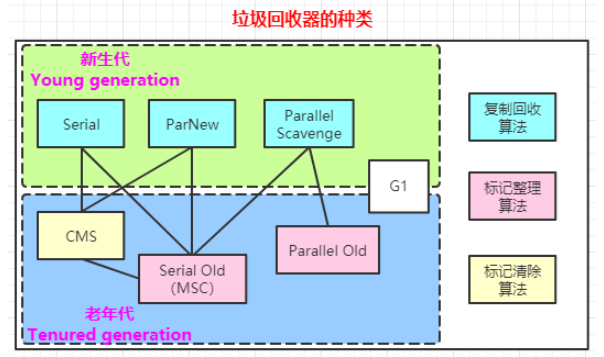


特点：

会影响系统程序的整体吞吐量(同时执行的)

清理不彻底，在清理时的同时，用户线程又在产生垃圾(浮动垃圾)

由于用户线程和垃圾回收线程是同时执行的，不能等空间不足时才执行GC，要提前执行(一般剩余20%开始GC，预留给浮动垃圾)



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，只使用一个线程去回收。新生代采用串行回收，采用复制回收算法，垃圾回收过程中会产生Stop the world，对于单个CPU环境，Serial效率最高(没有线程的切换)，

适用场景：平时的开发与调试程序使用，以及桌面应用交互程序

过程：当新生代内存不够用时，先暂停全部用户程序，然后开启一条GC线程使 用复制算法对垃圾进行回收，这一过程中可能会有一些对象提升到年老代

执行方式：单线程，串行

 开启参数：-XX:+UseSerialGC（client模式默认值）

Serial Old 收集器：

和Serial收集一样，只是回收场所是老年代，采用串行回收，标记压缩算法，垃圾回收过程中会产生Stop the world

ParNew收集器：

Serial回收器的多线程版本，任然会Stop the World，只是GC时是多个线程

可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数，对于容量较小的堆，Serial优于ParNew

过程：当新生代内存不够用时，先暂停全部用户程序，然后开启若干条GC线程 使用复制算法并行进行垃圾回收，这一过程中可能会有一些对象提升到年老代

执行方式：多线程，并行

适用场景：在中到大型的堆上，且系统处理器至少多于一个的情况

 开启参数：-XX:+UseParNewGC

Parallel Scavenge收集器：

和ParNew收集器类似，但是Parallel Scavenge收集关注的目标是系统吞吐量(-XX:ParallelGCThreads参数来限制垃圾收集的线程数)，停顿时间越短，用户体验感越好，主要适合在后台运算而不需要太多交互的任务。它可以让我们更精确的控制GC 停顿时间以及吞吐量；但是千万不要以为把大停顿时间调的越小越好，或者吞吐量越大越好，在使用 parallel scavenge搜集器时，主要有三个性能指标，大停顿时间、吞吐量以及新生代区 域的小值。

   parallel scavenge搜集器具有相应的调节策略，它将会优先满足大停顿时间的目 标，次之是吞吐量，后才是新生代区域的小值。

开启参数：数-XX:+UseParallelGC开启，server模 式下默认的新生代搜集器

Parallel Old收集器：

Parallel Scavenge收集器的老年代版本，采用标记-整理算法；在注重吞吐量及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器。开启Parallel Scavenge后，老年代默认采用Parallel Old

CMS(Concurrent Mark Swep)收集器：

CMS是一种获取最短回收停顿时间为目标的收集器，这使得它很适合用于和用户交互的业务，采用标记-清除算法，收集过程分为4部：初始标记(：需要暂停应用程序，快速标记存活对象)，并发标记(：恢复应用程序，并发跟踪GC Roots)，重新标记(：需要暂停应用程序，重新标记跟踪遗漏的对象)，并发清除(恢复应用程序，并发清除未标记的垃圾对象)；CMS用于老年代，CMS在内存碎片的整理过程中，是无法并发的(即不能分配内存)[通过 -XX:+UseCMSCompactAtFullCollection参数是否开启，默认开启]，但是这样会导致等待时间变长，-XX:CMSFullGCsBeforeCompaction参数用于设置执行多少次不压缩的FULL GC后跟着来一次带压缩的（默认值为0，表示每次进入Full GC时都进行碎片整理）。

缺点：GC线程和用户线程交替执行，系统吞吐量降低(弱于parallel scavenge)，而GC过程有新垃圾产生

并发模式失败：老年代在并发清除阶段清除不及时(即还存在内存碎片)，因此 造成的空闲内存不足，，在并发模式失败的情况下，serial old会作为备选搜集器，进行一次全局 GC（Full GC），因此serial old也算是CMS的“替补”。但是会造成较长的停顿时间，以调节XX:CMSInitiatingOccupancyFraction=<N>参数去控制当年老代的内存占用达到多少 的时候（N%），便开启并发搜集器开始回收年老代，避免并发模式失败

G1收集器：

用来替换CMS，G1有很多优点(充分利用多核CPU，缩短stop the world时间；分配大对象时不会因为空间不连续而触发GC；和CMS一样，可以指定在M毫秒时间内，消耗在垃圾收集上的时间不超过N毫秒)，但是CMS还是主流

根据堆的大小，系统吞吐量，系统响应时间来决策选择哪种

新生代和老年代的回收根据收集器的特点以及使用场景的不同可以搭配使用：



常用组合：

serial & serial old

client模式下的默认垃圾搜集器组合，也可 以使用参数-XX:+UseSerialGC强制开启。没有线程切换的开销，适用于小型应用，桌面应用以及开发，调试等

Parallel Scavenge & Parallel Old

开发不常见，但是很多对 吞吐量（throughout）要求较高或者对停顿时间（pause time）要求不高的应用程序的首 选，并且这个组合是server模式下的默认组合(用-XX:+UseParallelGC参数强制开启)， 该组合无论是新生代还是年老代都采用并行搜集，因此停顿时间较短，系统的整体吞 吐量较高。它适用于一些需要长期运行且对吞吐量有一定要求的后台程序。

ParNew & CMS（Serial Old作为替补）

开发不常见，但是对相应时间 （response time）要求较高的应用程序的首选。该组合需要使用参数XX:+UseConcMarkSweepGC开启。该组合在新生代采用并行搜集器，因此新生代的GC速度会非常快，停顿时间很短。 而年老代的GC采用并发搜集，大部分垃圾搜集的时间里，GC线程都是与应用程序并发执行 的，因此造成的停顿时间依然很短。它适用于一些需要长期运行且对相应时间有一定要求的 后台程序。典型的就是 我们的WEB应用程序。

垃圾搜集器选择参数

  UseSerialGC：开启此参数使用serial & serial old搜集器（client模式默认值）。

 UseParNewGC：开启此参数使用ParNew & serial old搜集器（不推荐）。

UseConcMarkSweepGC：开启此参数使用ParNew & CMS（serial old为替补）搜 集器。

 UseParallelGC：

开启此参数使用parallel scavenge & parallel old搜集器（server 模式默认值）。

UseParallelOldGC：开启此参数在年老代使用parallel old搜集器（该参数在JDK1.5 之后已无用）。

内存泄漏是堆中存在不再使用的对象但垃圾收集器无法从内存中删除它们的情况，因此它们会被不必要地一直存在。

当JVM中的一个线程OOM后，不会导致其他线程OOM，但线程OOM后，它占用的空间将会被释放，腾出了更多的内存空间

Jvm在创建对象时，需要确保线程安全，多个创建对象请求时，不会造成对内存地址的冲突，实现方式：

CAS+失败重试：通过CAS机制，分配失败后，再次请求分配，知道成功

TLAB：为每个线程预先在Eden去分配一块内存，在创建对象时，首先在TLAB上分配，当对象大于TLAB剩余的内存时，再采用CAS+失败重试

Minor GC/Full GC:

Minor GC发生在新生代，触发条件是当Eden区满时

Full GC发生在老年代，触发条件是，手动System.gc；老年代空间不足；执行CMS的同时，新垃圾产生(浮动垃圾)过多；

Minor GC后进入老年代的平均大小大于老年代的内存空间；大对象分配给新生代时，内存空间不足，可直接将大对象存放到老年代(此时判断空间是否足够)

Full GC = Minor GC + Major GC(老年代在执行Major GC时，会先执行一次Minor GC，所以说老年代直接执行的是Full GC)

Survivor的存在意义，就是减少被送到老年代的对象，进而 减少Full GC的发生，Survivor的预筛选保证，只有经历16次Minor GC还能在新生代中存活 的对象，才会被送到老年代。设置两个Survivor区最大的好处就是解决了碎片化(复制算法)

触发GC的条件：

当 Eden 区空间满时，就将触发一次 Minor GC

手动System.gc()触发Full GC，可通过 -XX:+ DisableExplicitGC 来禁止 RMI 调用 System.gc()

eden满了minor gc，升到老年代的对象(大对象/年龄达到老年代)大于老年代剩余空间(连续空间)，则执行full gc

在执行Minor GC之前，JVM会检查老年代的空间是否大于历次晋升到老年代对象的平均大小，如果小于，会检查是否设置了-XX:+HandlePromotionFailure（允许担保失败），如果允许，则只会进行Minior GC，如果不允许，执行完Minior CG后执行Major GC，也就是执行一次Full GC(尽量让对象在Minior GC时被回收，让对象在新生代多存活一段时间，以及不要创建大对象，大数组)

gc与非gc时间耗时超过了GCTimeRatio的限制引发OOM

执行CMS的同时，新垃圾产生(浮动垃圾)过多，导致暂时空间不足而触发Full GC

为了避免由于新生代对象晋升到旧生代导致旧生代空间不足的现象，在进行Minor GC时，如果统计所得到的Minor GC晋升到老年代的平均大小大于老年代的剩余空间，那么就直接触发Full GC

调优：通过NewRatio控制新生代老年代比例，通过MaxTenuringThreshold控 制进入老年前生存次数等.

永久代回收时机：

JDK1.8后，永久代占用的是系统内存，一般只要不超过系统内存，是不会触发OOM的，永久代回收条件：1该类的实例都被回收；2加载该类的ClassLoader以被回收；3，该类不能通过反射访问到其他方法，由此见永久代被回收的条件非常苛刻，执行Full GC的概率很小

JVM性能调优：

基本参数：

-Xmx1024m：JVM中最大可用内存

-Xms1024m：JVM中最小可用内存(一般-Xms和-Xmx设置相同，避免每次GC后，JVM重新分配内存)

-Xmn512m：设置年轻代内存大小，JVM内存大小 = 老年代 + 年轻代 + 永久代(JDK1.8以后，永久代占用系统内存)，官方推荐新生代的大小占整个堆的3/8

-Xss128k：设置每个线程的堆栈大小，这里应该占用的是系统内存，JVM中的内存被对象占用

-XX：NewSize=n：设置年代大小

-XX:NewRatio=4：设置年轻代（包括Eden和两个Survivor区）与年老代的比值；设置为4，则年轻代与年老代所占比值为1：4，年轻代占整个堆栈的1/5(设置年轻代大小的方式：-Xmn和-XXNewRatio)

-XX:SurvivorRatio=4：设置年轻代中Eden区与Survivor区的大小比值。设置为4，则两个Survivor区与一个Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6

-XX:MaxPermSize=16m：永久代大小

-XX:MaxTenuringThreshold=0：设置垃圾的最大年龄，默认15

-XX:+UseSerialGC:设置串行收集器

-XX:+UseParallelGC:设置并行收集器

-XX:+UseParalledlOldGC:设置并行年老代收集器

-XX:+UseConcMarkSweepGC:设置并发收集器

-XX:ParallelGCThreads=n:设置并行收集器收集时使用的CPU数。并行收集线程数。

-XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间

-XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)

-XX:+CMSIncrementalMode:设置为增量模式。适用于单CPU情况。

-XX:ParallelGCThreads=n:设置并发收集器年轻代收集方式为并行收集时，使用的CPU数。并行收集线程数。

垃圾回收器的选择：

**吞吐量**优先：

-Xmn，-Xms，-Xmn，-Xss等基本设置

-XX:+UseParallelGC：采用并行垃圾收集器(只是年轻代采用并行，老年代任采用串行)

-XX:ParallelGCThreads=20：并行收集器的线程数(一般和处理器数目相等)

-XX:+UseParallelOldGC：老年代采用并行收集器

-XX:MaxGCPauseMillis=100：设置每次年轻代执行Minior GC的最长时间，超过此时间，JVM会自动调整年轻代大小，以满足此值

-XX:+UseAdaptiveSizePolicy：并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等(建议使用并行收集器时开启)

并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。

**响应时间**优先：

方式1

-Xmn，-Xms，-Xmn，-Xss等基本设置

-XX:+UseConcMarkSweepGC：设置老年代为并发收集器(使用这个后，-XX：NewRatio=4会失效，所以用-Xmn设置年轻代大小)

-XX:+UseParNewGC:设置年轻代为并行收集。可与CMS收集同时使用。JDK5.0以上，JVM会根据系统配置自行设置，所以无需再设置此值。

方式2

-Xmn，-Xms，-Xmn，-Xss等基本设置

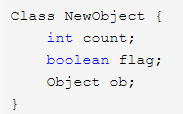
-XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection：打开对年老代的压缩。可能会影响性能，但是可以消除碎片

尽量少用finalize函数。finalize函数是Java提供给程序员一个释放对象或资源的机会。但是，它会加大GC的工作量，因此尽量少采用finalize方式回收资源。

使用软弱引用帮助GC回收

一个Object对象大小是8byte(Object ob = new Object();的大小为4byte+8byte)



空对象大小(8byte)+int大小(4byte)+Boolean大小(1byte)+空Object引用的大小(4byte)=17byte，因为Java在对对象内存分配时都是以8的整数倍来分，因此大于17byte的最接近8的整数倍的是24，因此此对象的大小为24byte

内存泄漏检测：

老年代被占满：老年代被占满说明对象任然在被引用，可能一些对象的引用未被释放，一般是集合对象引用未clean，长期未被使用的对象任占用内存

堆栈溢出：StackOverflowError

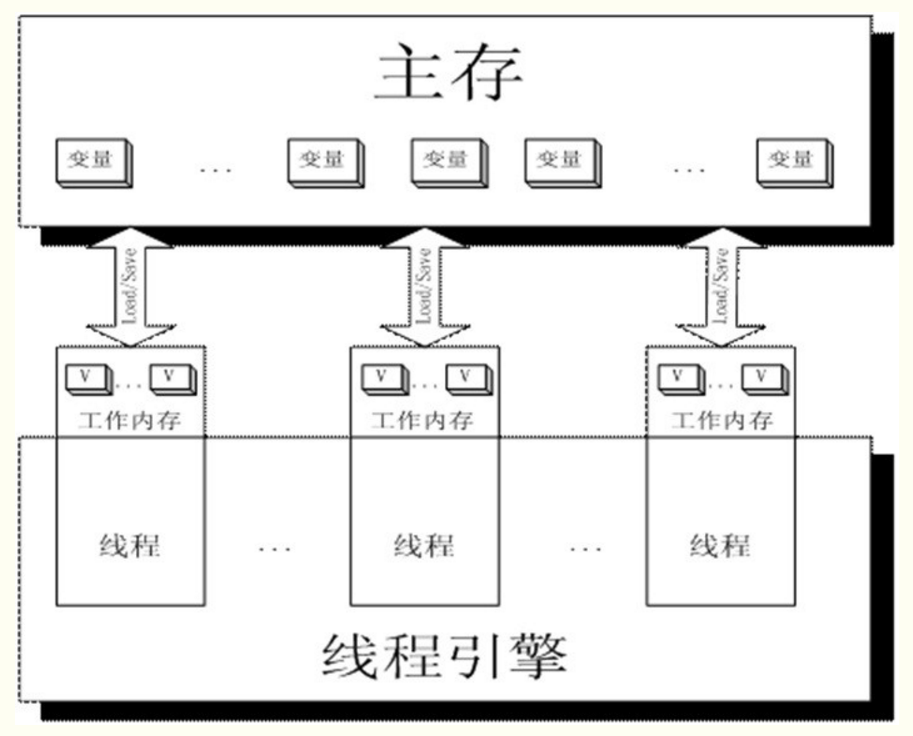
递归或循环，导致大于堆栈的深度

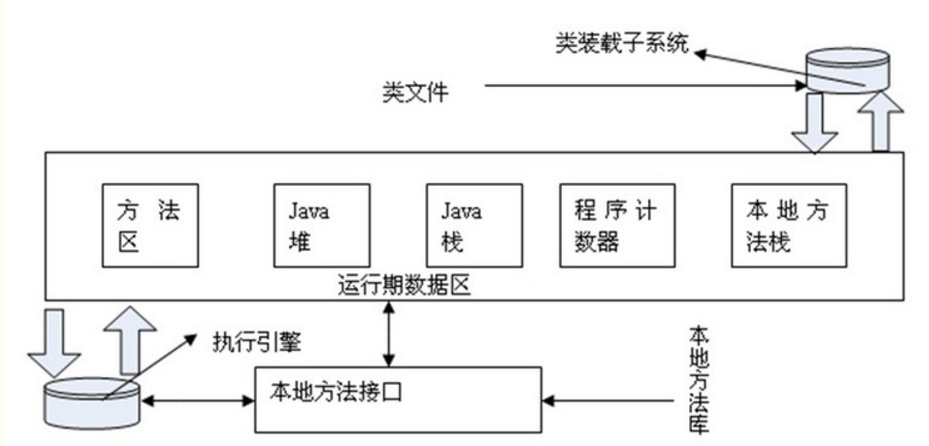
线程栈满：Stack size too small

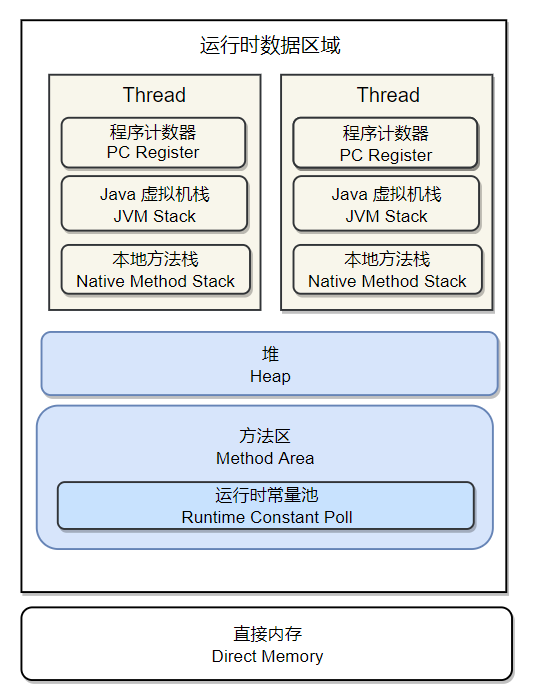
线程数目过多，使用-Xss增加线程数

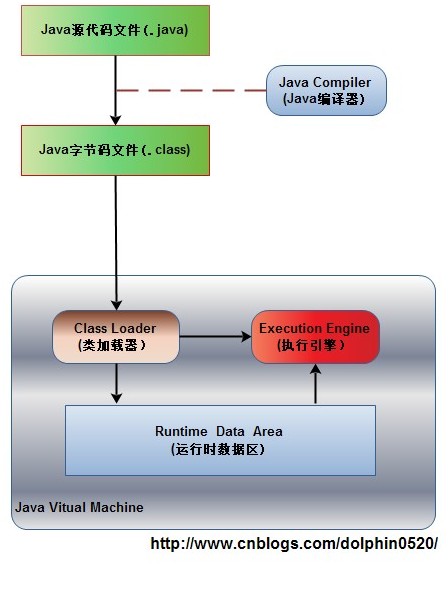
系统内存：OutOfMemoryError: unable to create new native thread

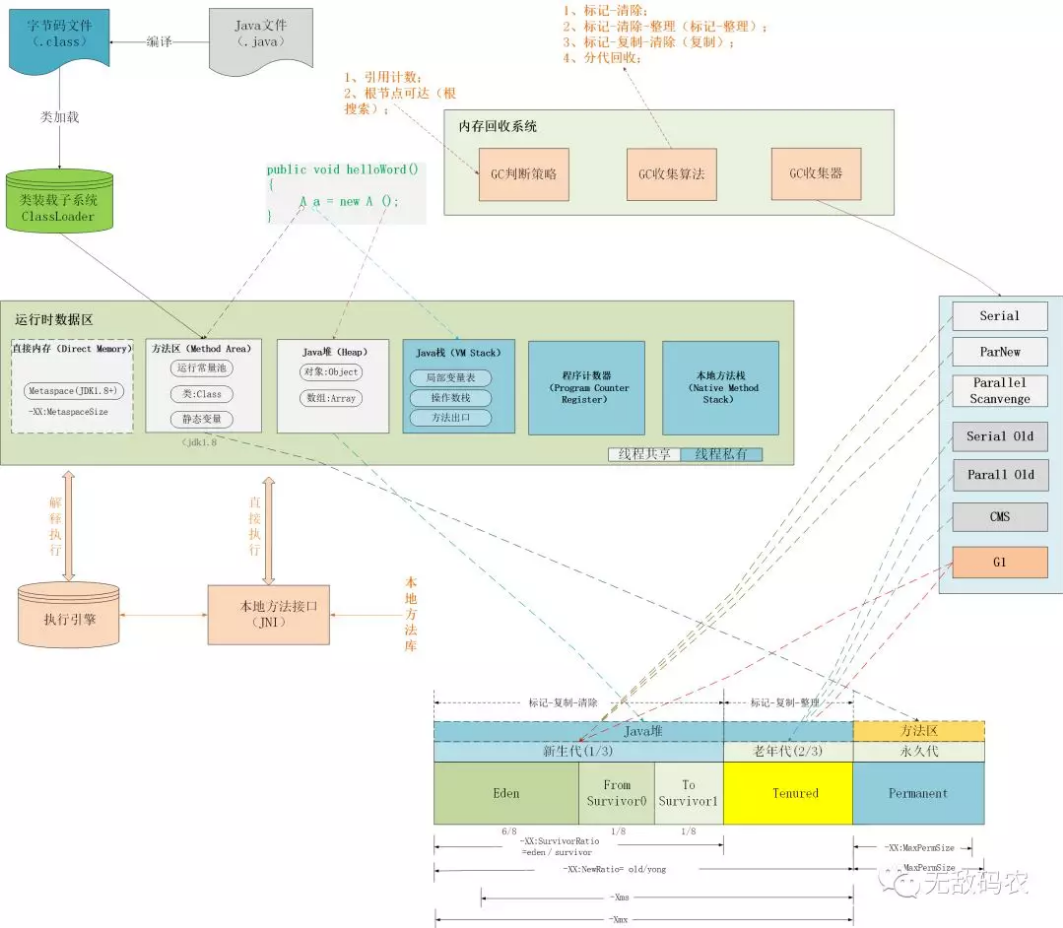
由于操作系统没有足够的资源来产生这个线程造成的。系统创建线程时，除了要在Java堆中分配内存外，操作系统本身也需要分配资源来创建线程。



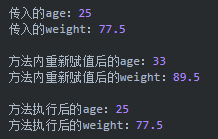






P指向堆中Person对象，在执行PersonCrossTest方法时，person也是指向的堆中Person对象，所以对person的修改实际就是对堆中对象的修改

Int等基本数据类型是在栈中，每个对应一个栈，私有

基本数据类型保存的就是自己本身，而引用类型保存的是引用值，值传递对应基本数据类型，引用传递对应引用传递

操作系统对一个进程内能创建的线程个数是有限制的，不能无限生成