**JVM探究：**

**·请你谈谈你对JVM的理解? java8虚拟机和之前的变化更新?**

**<https://blog.csdn.net/qq_38262266/article/details/107208357>**

**·什么是OOM，什么是栈溢出StackOverFlowError?怎么分析?**

**·JVM的常用调优参数有哪些?**

**·内存快照如何抓取，怎么分析Dump文件?知道吗?**

**·谈谈JVM中，类加载器你的认识?**

**目录**

**1.JVM的位置**

**2.JVM的体系结构**

**3.类加载器**

**4.双亲委派机制**

**5.沙箱安全机制**

**6.Native**

**7.PC寄存器**

**8.方法区**

**9.栈**

**10.三种JVM**

**11.堆**

**12.新生区、老年区**

**13.永久区**

**14.堆内存调优**

**15.GC**

**·常用算法**

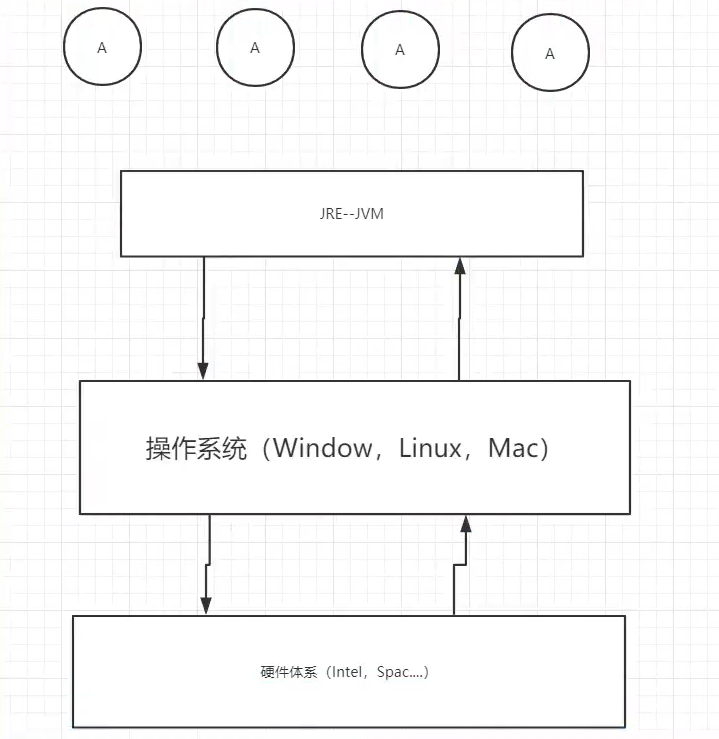
**16.JMM**

**17.总结**

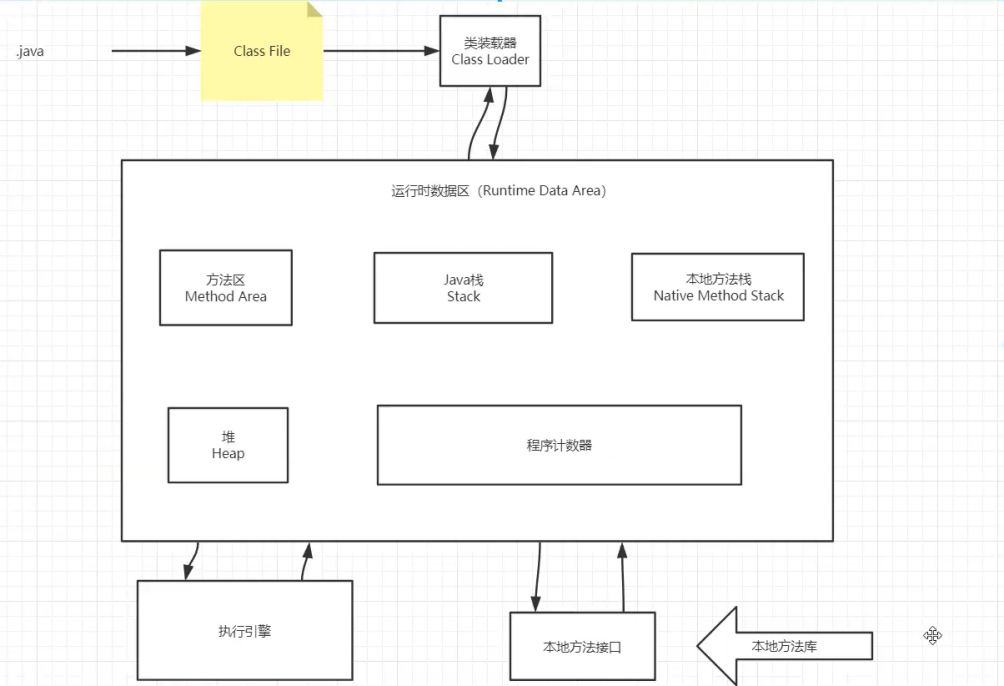
**1、jvm的位置：jre包含了jvm**

**·计算机系统分为软件系统和硬件系统**

**·软件系统又分为系统软件和应用软件（jre和jvm属于应用软件）**



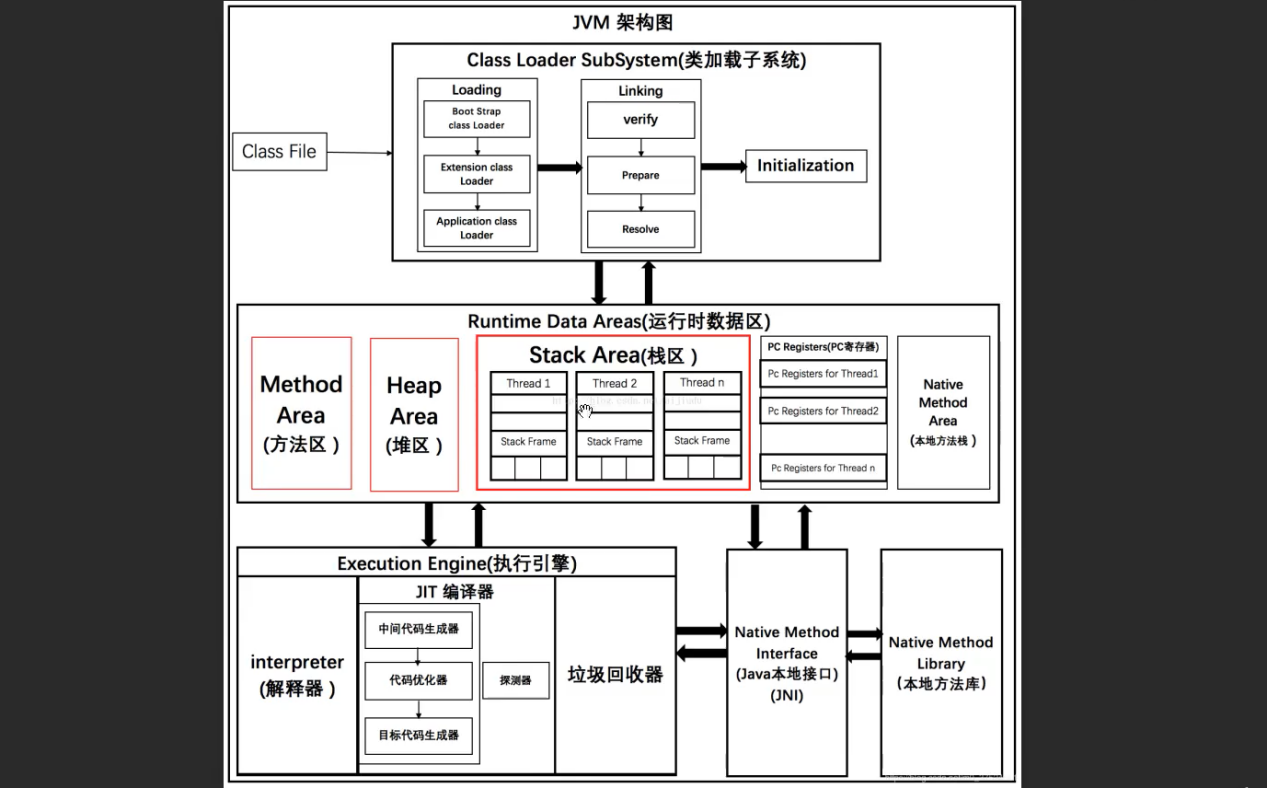
**2、jvm体系结构：**



·java栈，本地方法区，程序计数器不会有垃圾回收（java栈不会有垃圾，是因为不需要的数据就会被栈弹出）

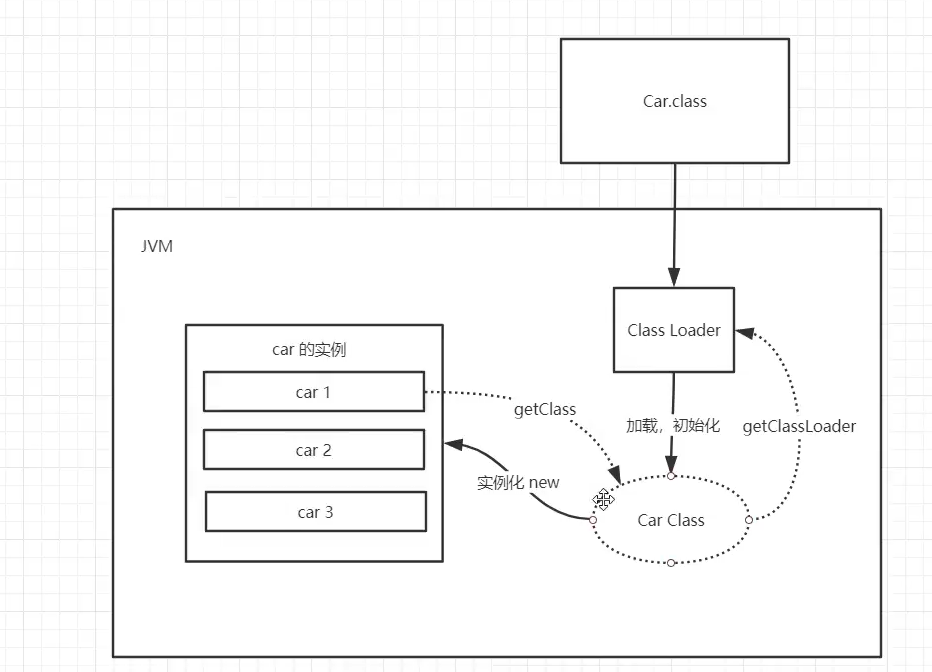
·所谓的jvm调优：就是在调整方法区和堆（大部分）这两个区域。（方法区是特殊的堆，大部分调优还是在调堆）。

更为详细的架构图：



**3、类加载器**

作用：加载class文件~



**·八**种基本数据类型存储在栈中，栈内创建和销毁非常的快

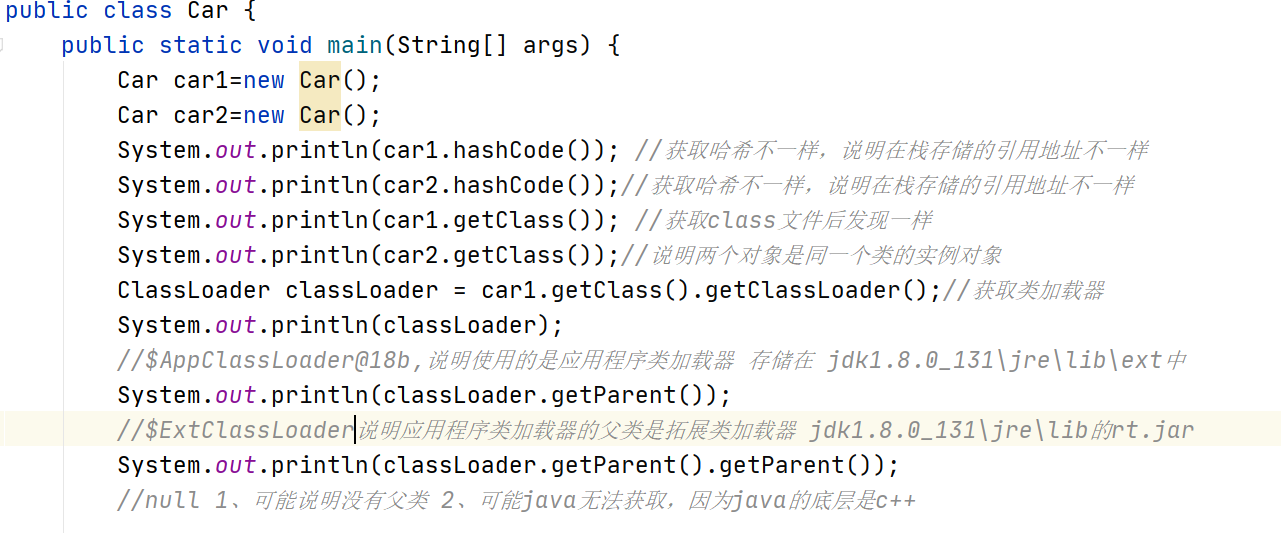
·八个基本数据类型都有对应的包装类，包装类就是对象了。比如Integer j = new Integer（10）。j属于对象的引用，引用放在栈中，而实际的数据10 则放在堆中。 （堆区适合存放大的数据对象，但是操作速度远远不及栈中）。

·（提示：对象的销毁---对象的引用放在栈中，所以使用完引用就被从栈中销毁了，但是实际的对象仍然存放在堆中，只有在没有任何的引用使用它的时候才被垃圾bai收器销毁掉）

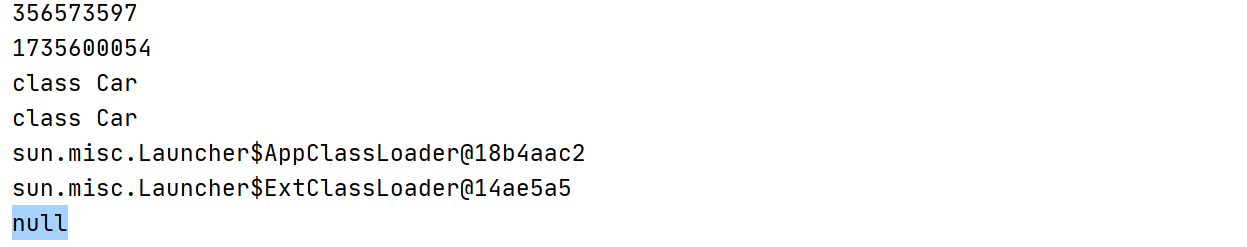
**类加载器分类：**

1. 虚拟机自带的加载器
2. 启动类（根）加载器（Bootstrap
3. 扩展类加载器 （ExtClassLoader
4. 应用程序（系统类）加载器（AppClassLoader

测试一下查看类加载器：扩展类加载器是应用程序加载器的父类



对应输出结果：

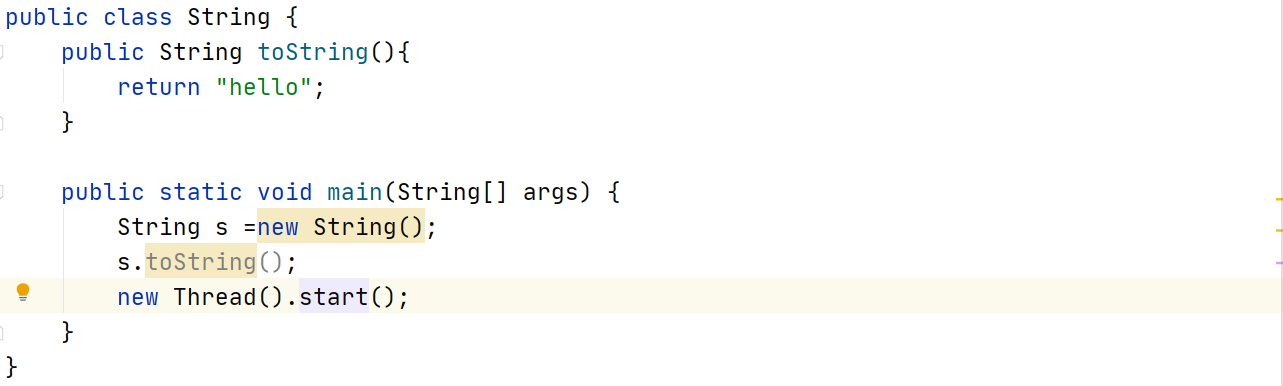


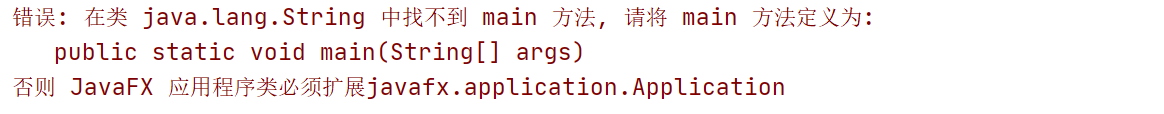
#出现null的原因，是因为java调用不到，因为上层是c和c++写的

1. **双亲委派机制**
   1. 类加载器收到类加载请求
   2. 将这个请求向上委托给父类加载器区完成，一直向上委托，直到根加载器（或者叫启动类加载器，bootstrap）
   3. 启动类加载器检查是否能够加载该类，能就加载结束，使用当前加载器。否则，抛出异常，通知子类进行加载。
   4. 重复步骤3

#如果都找不到会报class not found~

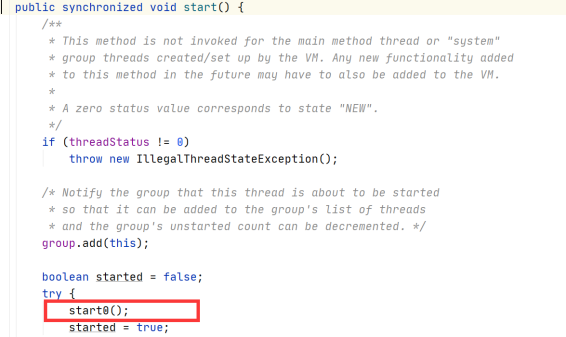
案例：



输出结果：因为向上委派的时候，发现有这个类，但是这个类需要有字符串长度，所以报错

#java是在c++的基础上去掉了繁琐的功能例如指针，内存管理（交给jvm），进入线程查看一下：

·进入start方法

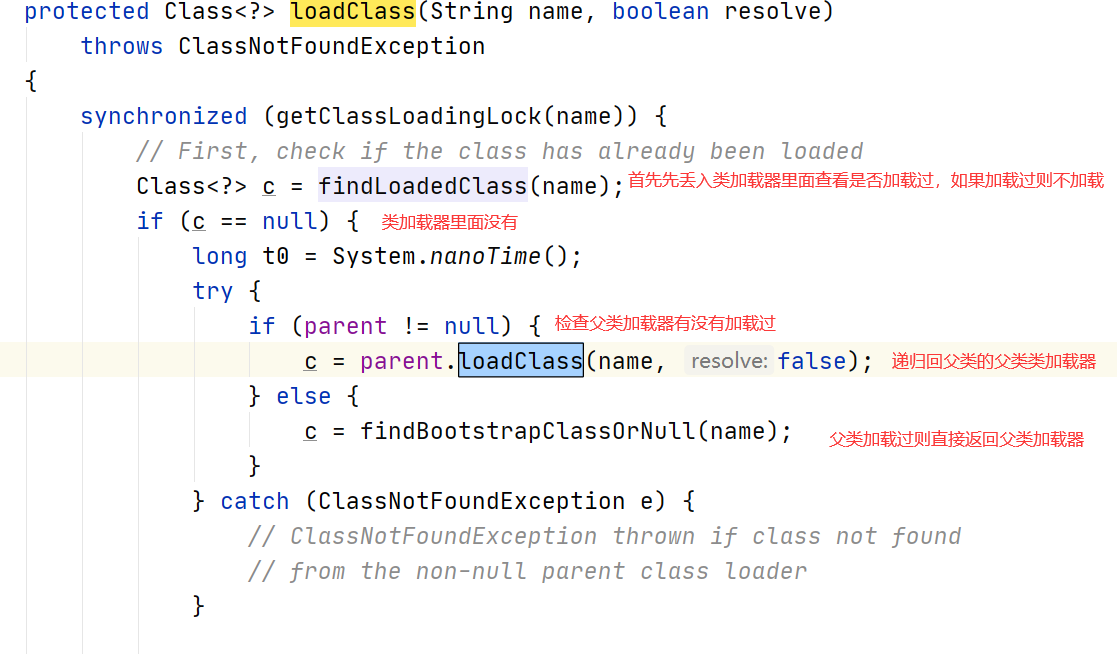


发现线程调用了start0，但跟进后发现start0并没有方法，说明start0不是java方法，百度后知道，start0是jni即Java Native Interface就是JNI，java本地方法接口。



**百度双亲委派机制：**

**·**双亲委派机制源码



public Class<?> loadClass(String name) throws ClassNotFoundException {  
 return loadClass(name, false);  
}

protected Class<?> loadClass(String name, boolean resolve)  
 throws ClassNotFoundException  
{  
 synchronized (getClassLoadingLock(name)) {  
 // First, check if the class has already been loaded  
 Class<?> c = findLoadedClass(name);  
 if (c == null) {  
 long t0 = System.nanoTime();  
 try {  
 if (parent != null) {  
 c = parent.loadClass(name, false);  
 } else {  
 c = findBootstrapClassOrNull(name);  
 }  
 } catch (ClassNotFoundException e) {  
 // ClassNotFoundException thrown if class not found  
 // from the non-null parent class loader  
 }  
 if (c == null) {  
 // If still not found, then invoke findClass in order  
 // to find the class.  
 long t1 = System.nanoTime();  
 c = findClass(name);  
 // this is the defining class loader; record the stats  
 sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);  
 sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);  
 sun.misc.PerfCounter.getFindClasses().increment();  
 }  
 }  
 if (resolve) {  
 resolveClass(c);  
 }  
 return c; }}

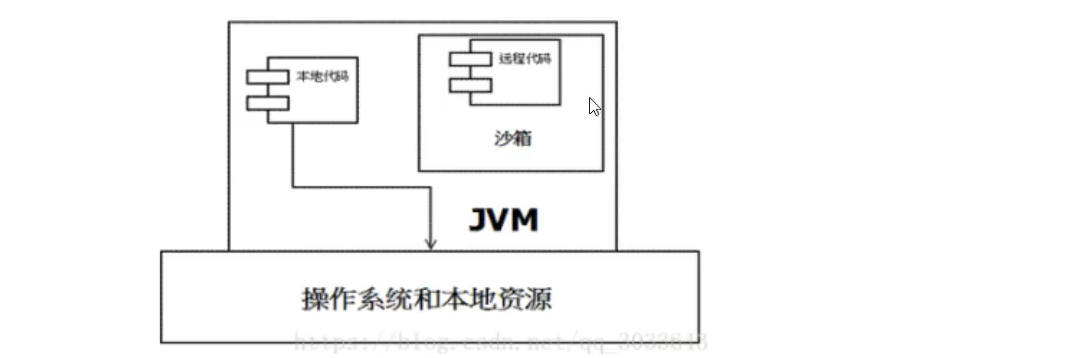
1、防止重复加载同一个.class。通过委托去向上面问一问，加载过了，就不用再加载一遍。保证数据安全。  
2、保证核心.class不能被篡改。通过委托方式，不会去篡改核心.class，即使篡改也不会去加载（比如你也写了个string类），即使加载也不会是同一个.class对象了。不同的加载器加载同一个.class也不是同一个Class对象。这样保证了Class执行安全

**沙箱安全机制：**

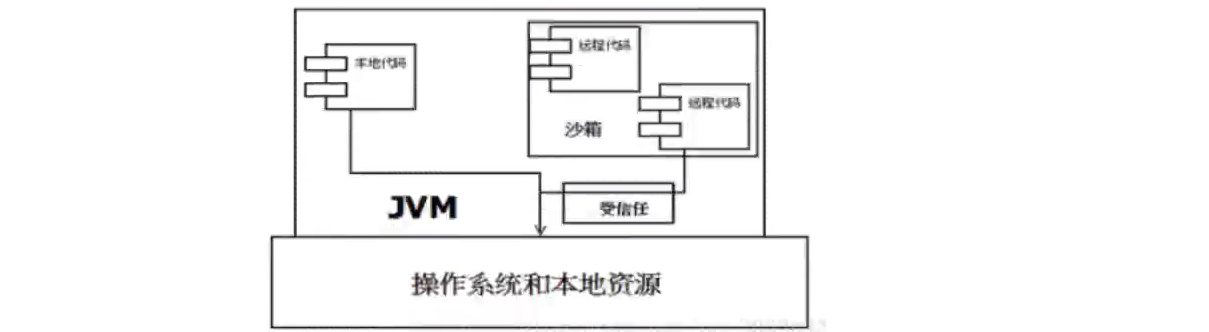
**· J**ava安全模型的核心就是Java沙箱(sandbox)，**什么是沙箱**?沙箱是一个限制程序运行的环境。沙箱机制就是将Java代码限定在虚拟机(JVM)特定的运行范围中，并且严格限制代码对本地系统资源访问，通过这样的措施来保证对代码的有效隔离，防止对本地系统造成破坏。沙箱主要限制系统资源访问，那系统资源包括什么?CPU、内存、文件系统、网络。不同级别的沙箱对这些资源访问的限制也可以不一样。

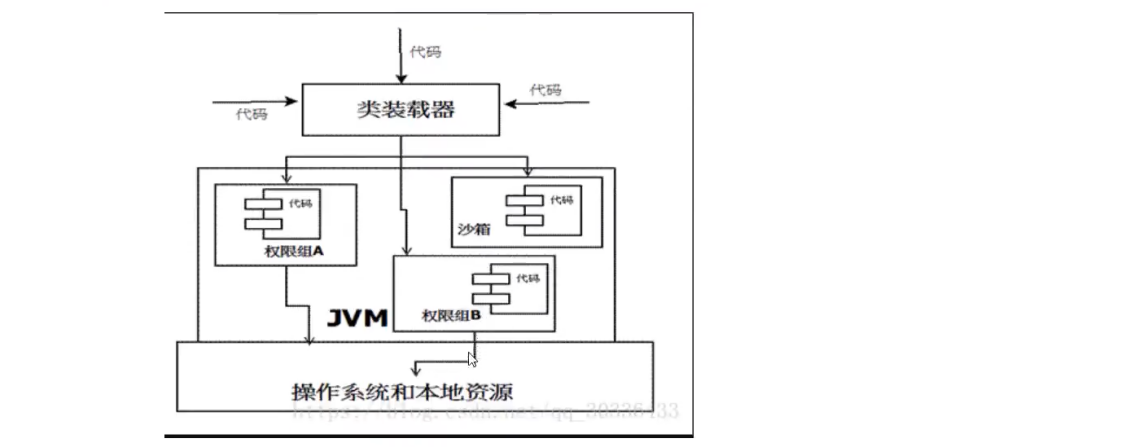
·所有的Java程序运行都可以指定沙箱，可以定制安全策略。

·在]ava中将执行程序分成**本地代码**和**远程代码**两种，本地代码默认视为可信任的，而远程代码则被看作是不受信的。对于授信的本地代码，可以访问一切本地资源。而对于非授信的远程代码在早期的Java实现中，安全依赖于沙箱(Sandbox)机制。如下图所示JDK1.0安全模型。

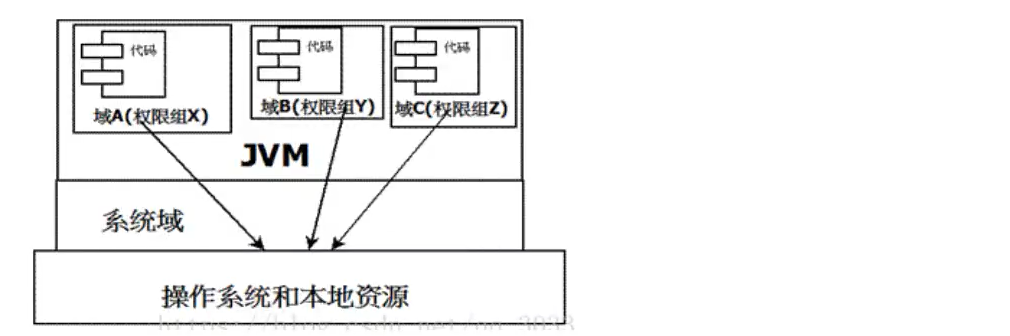


但如此严格的安全机制也给程序的功能扩展带来障碍，比如当用户希望远程代码访问本地系统的文件时候，就无法实现。因此在后续的Java1.1版本中，针对安全机制做了改进．增加了安全策略，允许用户指定代码对本地资源的访问权限。如下图所示JDK1.1安全模型。



在Java1.2版本中，再次改进了安全机制，增加了代码签名。不论本地代码或是远程代码，都会按照用户的安全策略设定，由类加载器加载到虚拟机中权限不同的运行空间，来实现差异化的代码执行权限控制。如下图所示JDK1.2安全模型

当前最新的安全机制实现，则引入了域(Domain)的概念。虚拟机会把所有代码加载到不同的系统域和应用域，系统域部分专门负责与关键资源进行交互，而各个应用域部分则通过系统域的部分代理来对各种需要的资源进行访问。虚拟机中不同的受保护域(Protected Domain)，对应不一样的权限(Permission)。存在于不同域中的类文件就具有了当前域的全部权限，如下图所示最新的安全模型(jdk 1.6)。比如a方法调用b方法，b调用a方法，这种情况会导致栈溢出，系统不会让其调用。



组成沙箱的基本组件:

1、**字节码校验器**(bytecode verifier)︰确保java类文件遵循lava语言规范。这样可以帮助lava程序实现内存保护。但并不是所有的类文件都会经过字节码校验，比如核心类。java javax

2、**类装载器(class loader)** :其中类装载器在3个方面对Java沙箱起作用

·它防止恶意代码去干涉善意的代码;

·它守护了被信任的类库边界;

·它将代码归入保护域，确定了代码可以进行哪些操作。

虚拟机为不同的类加载器载入的类提供不同的命名空间，命名空间由一系列唯一的名称组成，每一个被装载的类将有一个名字，这个命名空间是由java虚拟机为每一个类装载器维护的，它们互相之间甚至不可见。

类装载器采用的机制是双亲委派模式。

1.从最内层JVM自带类加载器开始加载，外层恶意同名类得不到加载从而无法使用;

2.由于严格通过包来区分了访问域，外层恶意的类通过内置代码也无法获得权限访问到内层类，破坏代码就自然无法生效。

·存贩控制器〈access controller):存取控制器可以控制核心API对操作系统的存取权限，而这个控制的策略设定，可以由用户指定。

·安全管理器(secuirity manager):是核心API和操作系统之间的主要接口。实现权限控制，比存取控制器优先级高。

·安全软件包(security package) : java.security下的类和扩展包下的类，允许用户为自己的应用增加新的安全特性，包括:

·安全提供者

·消息摘要

·数字签名keytools

·加密

·鉴别

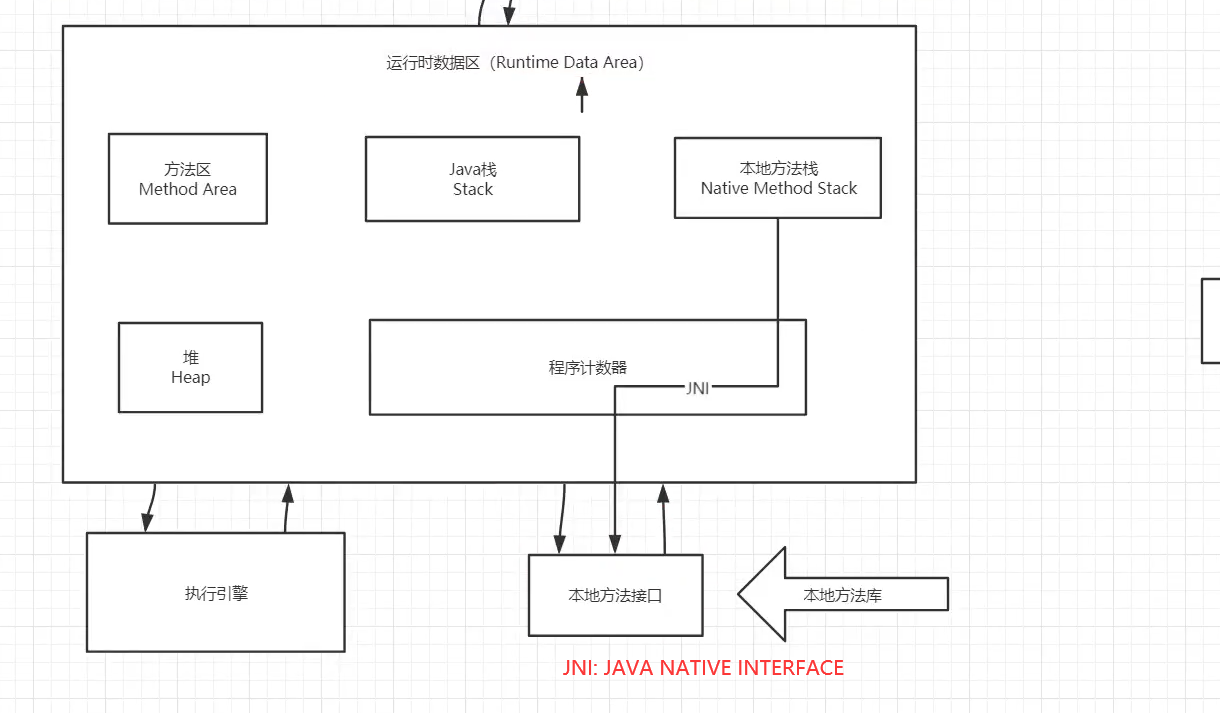
**6、Native：**

native :凡是带了native 关键字的，说明java的作用范围达不到了，会去调用底层c语言的库，会进入本地方法栈去JNI（java native interface里面去找）

·调用本地方法本地接口：JNI

JNI作用:扩展Java的使用，融合不同的编程语言为Java所用!最初: C、C++。Java诞生的时候C、C++横行，想要立足，必须要有调用C、C++的程序~

//它在内存区域中专门开辟了一块标记区域:Native Method Stack，登记 nativa方法，在最终执行的时候,加载本地方法库中的方法通过了NI



1. **PC寄存器**

程序计数器：Program Counter Register

每个线程都有一个程序计数器，是线程私有的，就是一个指针，指向方法区中的方法字节码（用来存储指向像一条指令的地址，也即将要执行的指令代码)，在执行引擎读取下一条指令，是一个非常小的内存空间，几乎可以忽略不计。

1. **方法区：存储static、final、Class、常量池~（1.8后方法区移到了元空间里）**

方法区：Method Area

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，此区域属于共享区间;

静态变量（static）、常量（final）、类信息(构造方法、接口定义，class)、运行时的常量池存在方法区中，但是实例变量存在堆内存中，和方法区无关。

static、final、class、常量池

1. **栈：栈内存，主管程序的运行，生命周期和线程同步;（非共用）**

**（**8大基本数据类型+对象引用+实例方法。 栈是线程私有的，意思是说每个线程都有自己的栈。**）**

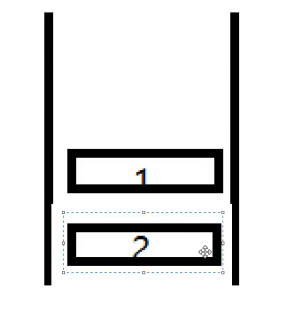
程序=数据结构＋算法︰持续学习~

程序=框架＋业务逻辑:吃饭~

常见的几种数据结构：

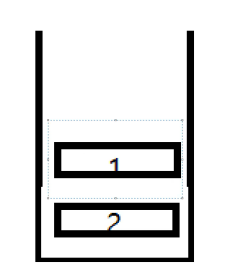
1、队列（FIFO,first input first output）：先进先出

·队列的示意图：2先进去，2先出来，1后进来，1后出来



2、栈:先进后出、后进先出︰桶

·栈的示意图：2先进去，1后进去，但是1先出来，2后出来。



·这也就解释了为什么main方法先执行最后结束，因为main方法先压栈进去，然后main方法里的其他方法在压栈进来，这时候只能先等main以外的方法结束了，main方法才能出栈。

·栈溢出：假设a调用b，b调用a，然后主方法调用a或者b，则两个方法会不停的压进栈，最终从栈里面溢出，栈溢出后会报stackOverflowError

#线程结束，栈内存也就是释放，对于栈来说，不存在垃圾回收问题。

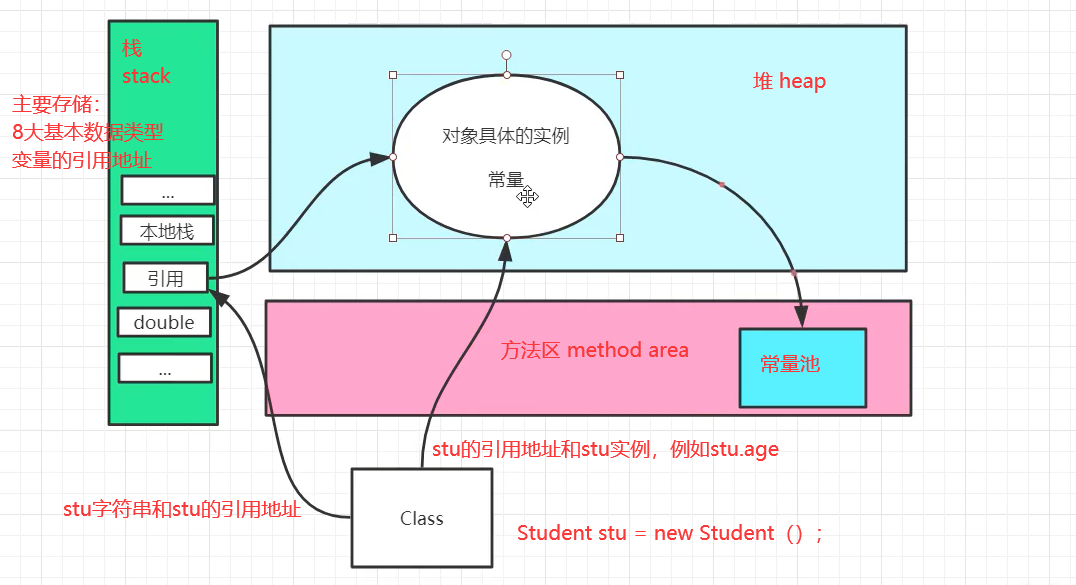
#一旦线程结束，栈就Over!

·栈：8大基本数据类型+对象引用+实例方法。

3、栈帧：每一个方法的调用都会产生一个栈帧



栈、堆、方法区：交互关系



图里面的class 应是对象。

#静态变量（static）、常量（final）、类信息(构造方法、接口定义，class)、运行时的常量池存在方法区中，但是实例变量存在堆内存中，和方法区无关。

#栈还负责压入各种方法

理解：

·栈里面的东西具体是怎么存的？

栈中存储什么？

·每个线程都有自己的栈，栈中的数据都是以栈帧的格式存在。

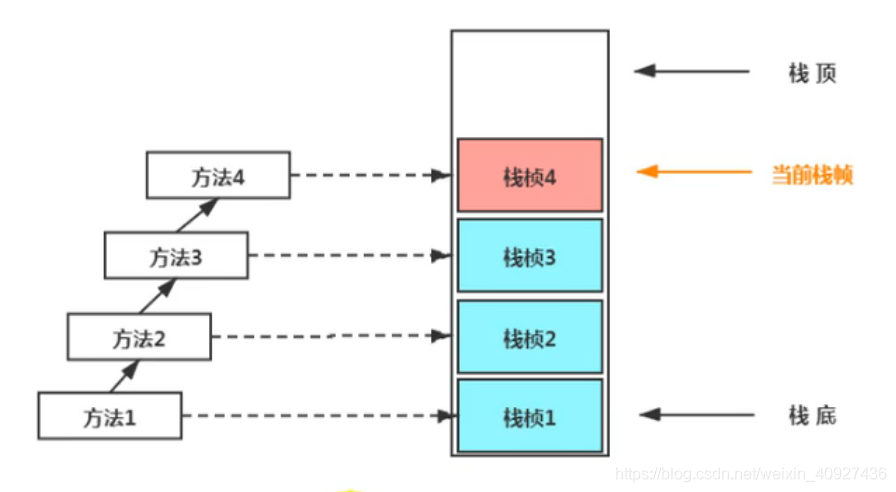
·在这个线程上正在执行的每一个方法都各自对应一个栈帧

·栈帧是一个内存区块，是一个数据集维系着方法执行过程中的各种数据信息

在一条活动线程中，一个时间点上，只会有一个活动的栈帧，即只有当前正在执行的方法的栈帧（栈顶栈帧）是有效的，这个栈帧被称为当前栈帧（Current Frame），与当前栈帧对应的方法就是当前方法（Current method），定义这个方法的类就是当前类（Current Class）

执行引擎运行的所有字节码指令只会针对当前栈帧进行操作

如果在该方法中调用了其他方法，对应的新的栈帧就会被创建出来，放在栈的顶端，称为新的当前帧。



栈运行的原理：

·不同的线程中所包含的栈帧是不允许互相之间引用的，即不可能在一个栈帧之中引用另外一个线程的栈帧。

·如果当前方法调用了其他方法，方法返回之际，当前栈帧会传回此方法的执行结果给前一个栈帧，接着，虚拟机会丢弃当前栈帧，使得前一个栈帧重新成为当前栈帧。

·Java方法由两种返回函数的方式，一种是正常的函数返回，使用return指令；另一种是抛出异常。不管使用哪种方式，都会导致栈帧被弹出

·当没有使用抛出异常时，methodA的异常会返回给main方法，看main方法有没有对应的异常处理，如果没有则报错并且导致System.out.println(“methodA执行完毕”);没有执行，

·画出一个对象实例化的过程在内存中？

[java对象的实例化过程](https://www.cnblogs.com/fangfangfanga/p/12070363.html)

Student stu =new Student（）;

·简单类对象的实例化过程

1、在方法区加载类；

2、在栈内存申请空间，声明变量stu；

3、在堆内存中开辟空间，**分配对象地址**；

4、在对象空间中，对对象的属性进行默认初始化，类成员变量显示初始化；

5、构造方法进栈，进行初始化；

6、初始化完成后，将堆内存中的地址赋给引用变量，构造方法出栈；

·子类对象的实例化过程

1、在方法区先加载父类，再加载子类；

2、在栈中申请空间，声明变量stu；

3、在堆内存中开辟空间，分配对象地址；

4、在对象空间中，对对象的属性（包括父类的属性）进行默认初始化；

5、子类构造方法进栈；

6、显示初始化父类的属性；

7、父类构造方法进栈，执行完毕出栈；

8、显示初始化子类的属性；

9、初始化完毕后，将堆内存中的地址值赋给引用变量stu，子类构造方法出栈；

**10、三种jvm**

·Sun公司的HotSpot

·BEA JRockit

·IBM J9VM

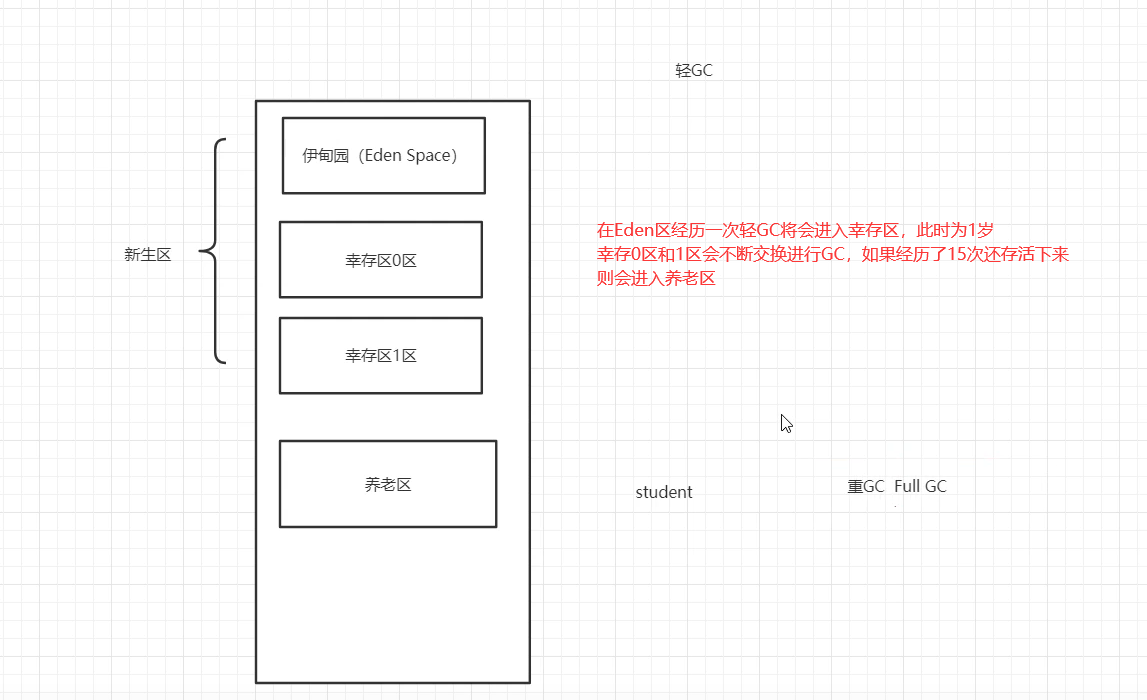
我们学习的都是：基于Sun公司的HotSpot

**11、堆：字符串常量池（新生代，老年代，永久代都在堆里）**

Heap，一个jvm只有一个堆内存，堆内存大小是可以调节的。

·类加载器读取了类文件后，一般会把什么东西放到堆中？new 出来的对象，对象的数据

jdk8以前的示意图：new出来的对象会进入伊甸园区eden



GC垃圾回收，主要是在伊甸园区（新生代）和养老区（老年代）~

假设内存满了就会爆OOM错误，即堆内存不够！java.lang.OutOfMemoryError: Java heap space！

#java1.8以后取消了永久代，即永久存储区，变成了元空间

#元空间里面负责存储：运行时常量池和永久代

**新生代**：

·类：诞生和成长的地方，甚至死亡（在首次轻gc中没有被引用的）

·伊甸园区（Eden）：所有的对象都是在Eden

·幸存区：分为from区和to区（哪边是空的哪边是to区），from区

**老年代：**

·当新生代经历15次轻GC后还存在引用的或新生代满了，则被转移到老年代

**元空间：这个区域常驻内存的，用来存放jdk自身携带的class对象，interface元数据，存储的是java运行时的一些环境或类信息~这个区域不存在垃圾回收！关闭虚拟机释放内存**

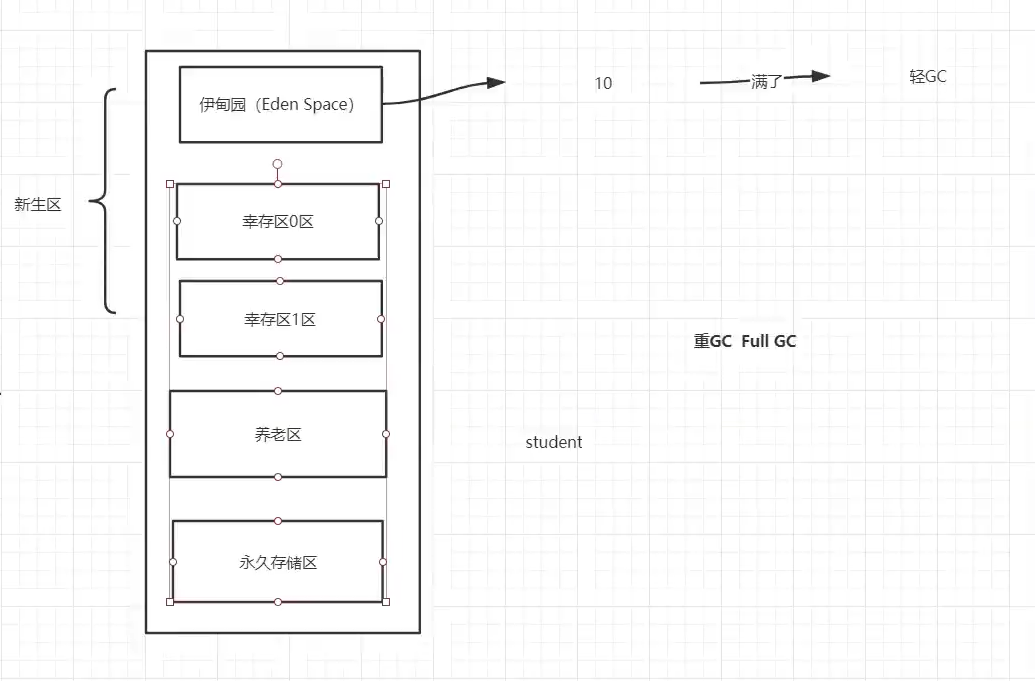
·jdk1.6之前：永久代，运行时常量池和字符串常量池在方法区

·jdk1.7 ：永久代，但是慢慢退化，去永久代，字符串常量池在堆中

·jdk1.8之后：无永久代，运行时常量池在元空间，字符串常量池在堆中

#一个启动类加载了大量的第三方jar包，tomcat部署了太多应用，大量动态生成的反射类，不断被加载，直到内存满就会出现OOM

GC过程图解：



1. 首先出现Student stu =new Student（）；的时候，伊甸区Eden会出现new 一个新的对象出来，然后假设此时的新生代只允许存储15个实例对象，如果某段时间内新生区15个对象满了，就会触发轻GC，轻GC下会清除一些没有被引用的对象出去，没有被清除的则进入幸存区，幸存区和伊甸区Eden分开计算存储空间。
2. 重复1的步骤直到我们的伊甸区和幸存区都满了，则会触发重GC ,重CG会同时对幸存区和伊甸区的所有对象进行清理，如果还有引用的对象则会保留下来，进入老年代。
3. 重复2的动作，如果此时，老年代也满了，则会触发OOM，java.lang.OutOfMemoryError：heap，即堆溢出。

jdk1.8后的堆结构图：

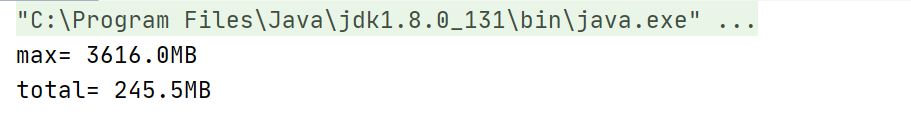


#元空间，逻辑上存在，物理上并不存在，内存不在堆里

**尝试一下常看JVM内存和调整JVM内存大小：**

1、查看JVM默认内存大小

public class Demo1 {  
 public static void main(String[] args) {  
 long max = Runtime.getRuntime().maxMemory();//返回虚拟机请求的最大内存，单位是字节  
 long total=Runtime.getRuntime().totalMemory();//返回jvm初始化总内存  
  
 System.out.println("max= "+(max/(double)1024/1024)+"MB");  
 System.out.println("total= "+(total/(double)1024/1024)+"MB");  
  
 }  
}



#默认最大请求内存是当前内存的1/4

1. 修改请求内存大小

·可以修改的参数：

-Xmx   Java Heap最大值，默认值为物理内存的1/4，最佳设值应该视物理内存大小及计算机内其他内存开销而定；

-Xms   Java Heap初始值，Server端JVM最好将-Xms和-Xmx设为相同值，开发测试机JVM可以保留默认值；

-Xmn   Java Heap Young区大小，不熟悉最好保留默认值；

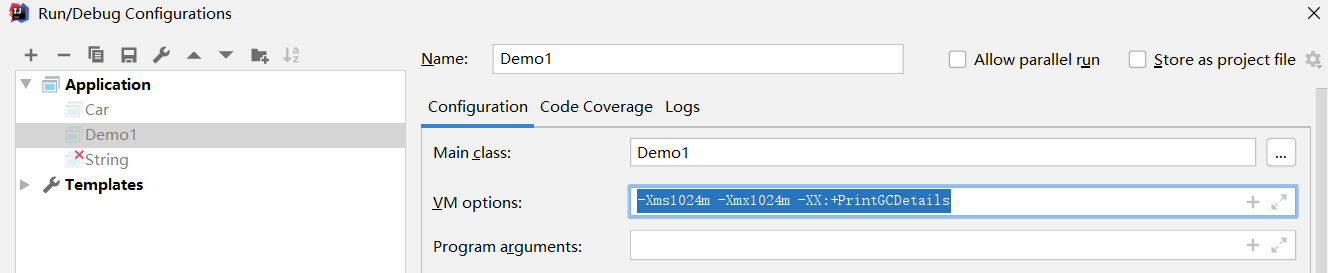
-Xss   每个线程的Stack大小，不熟悉最好保留默认值；

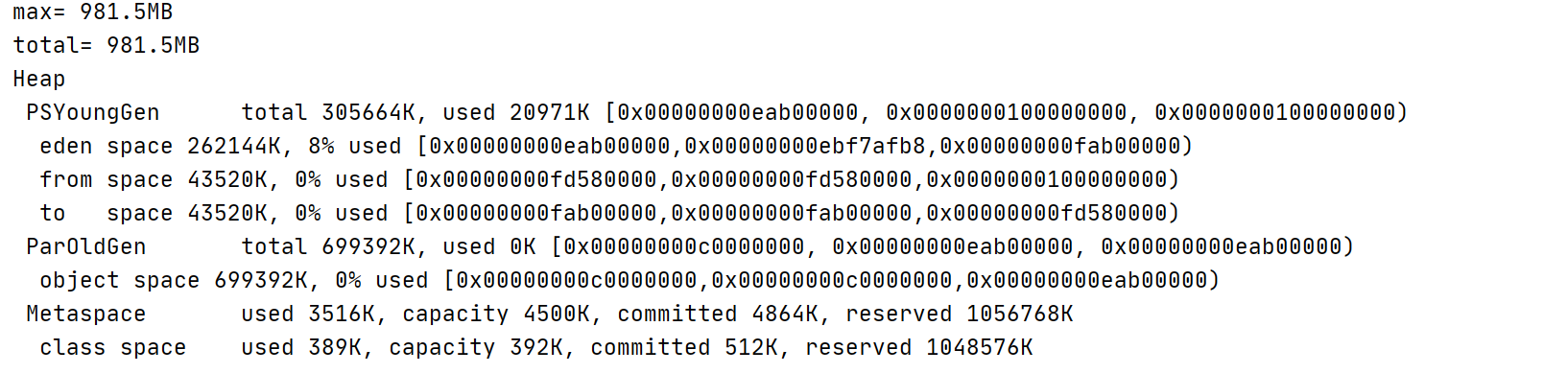
·详细看博客：

<https://blog.csdn.net/nimqbiyq/article/details/82256202>

在JVM配置文件里填入：

-Xms1024m -Xmx1024m -XX:+PrintGCDetails





打印后发现：YoungGen （新生代）+ParOldGen（老年代）=堆内存

说明元空间只是逻辑存在，物理上并不存在于堆中，在本地内存。

**当出现OOM内存溢出的错误时候，怎么处理？**

1、尝试扩大内存看结果（在idea的配置文件里面修改VM options），如果改变完后还出现问题则说明是我的代码存在着错误。

2、使用内存快照分析工具MAT或Jprofiler分析，看一下哪个地方出现了问题

3、debug，一行一行分析代码（项目上线后并不能使用debug）

MAT,Jprofiler作用：

·分析Dump内存文件，快速定位内存泄漏

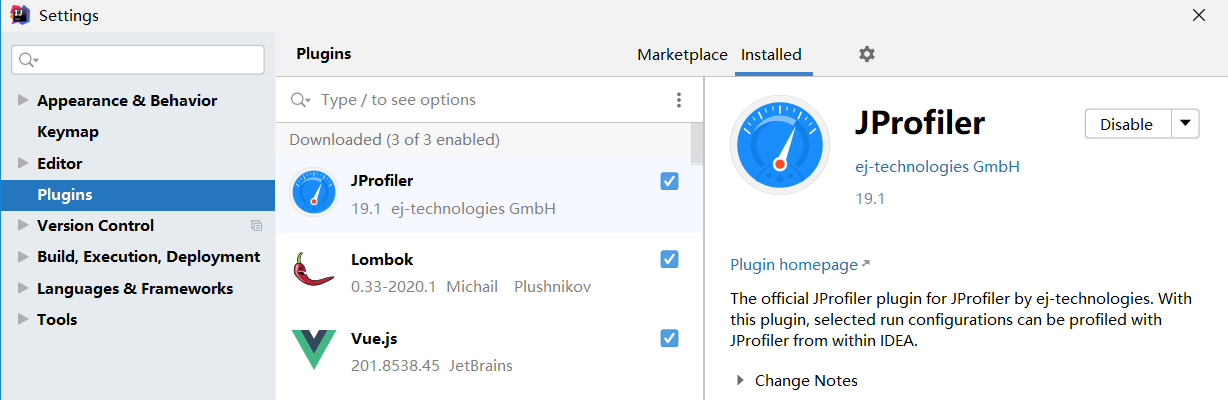
·获得堆中的数据

·获得大的对象

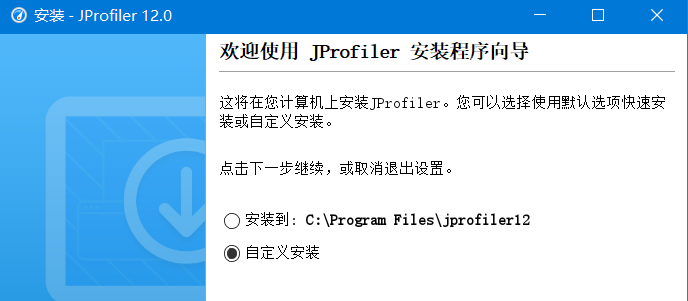
·。。。

Jprofiler的使用：

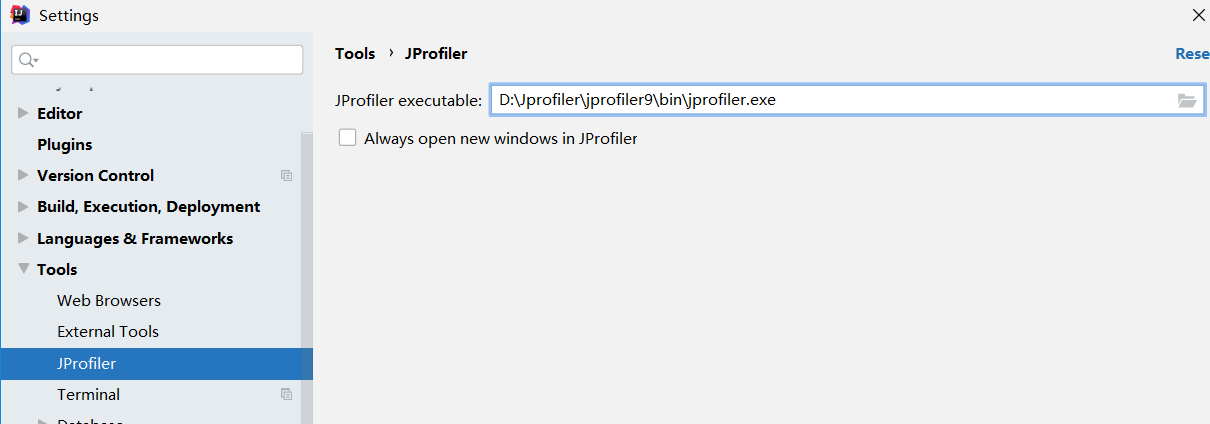
1. 插件里面下载一个Jprofiler



1. 电脑安装一个Jprofiler，安装路径不能有空格，不能有中文



1. 在settings下的tools找到jprofiles然后写入目录



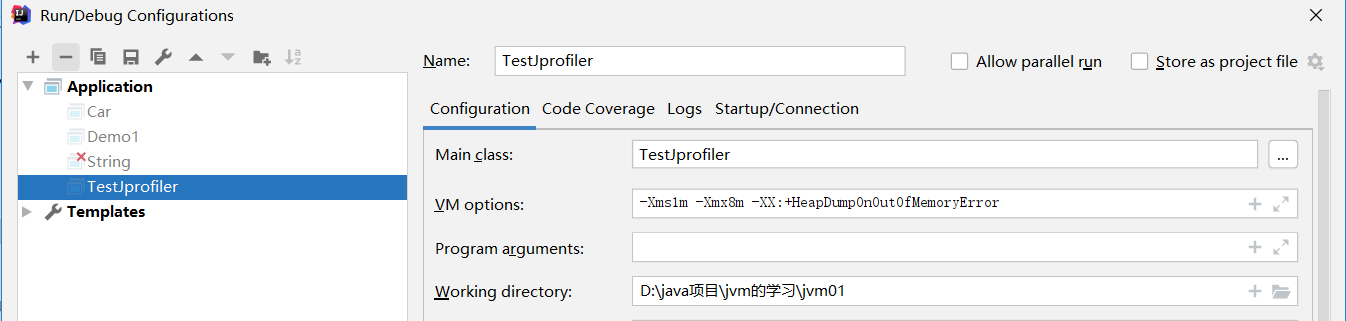
1. 在jvm的配置文件里面使用-XX:+HeapDumpOnOutOfMemoryError打印OOM的错误信息

-Xms 设置默认分配内存大小

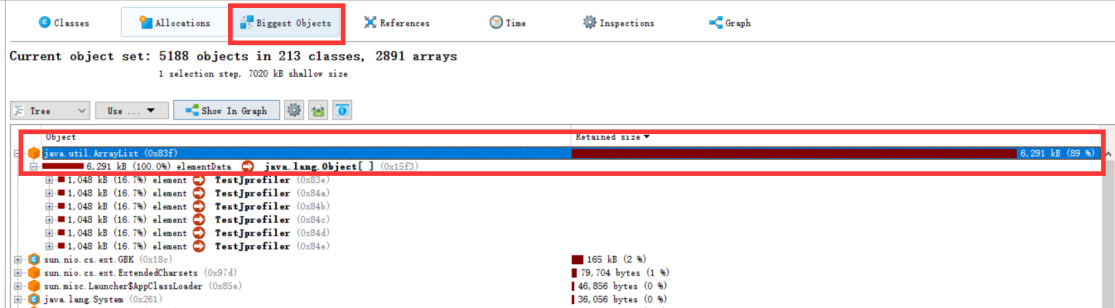
-Xmx 设置最大分配内存

-XX:+PrintGCDetails 打印GC清理的信息，以及堆，新生代，老年代，元空间的大小

-XX:HeapDumpOnOutOfMemoryError 生成Dump文件，用于给内存快照分析工具分析

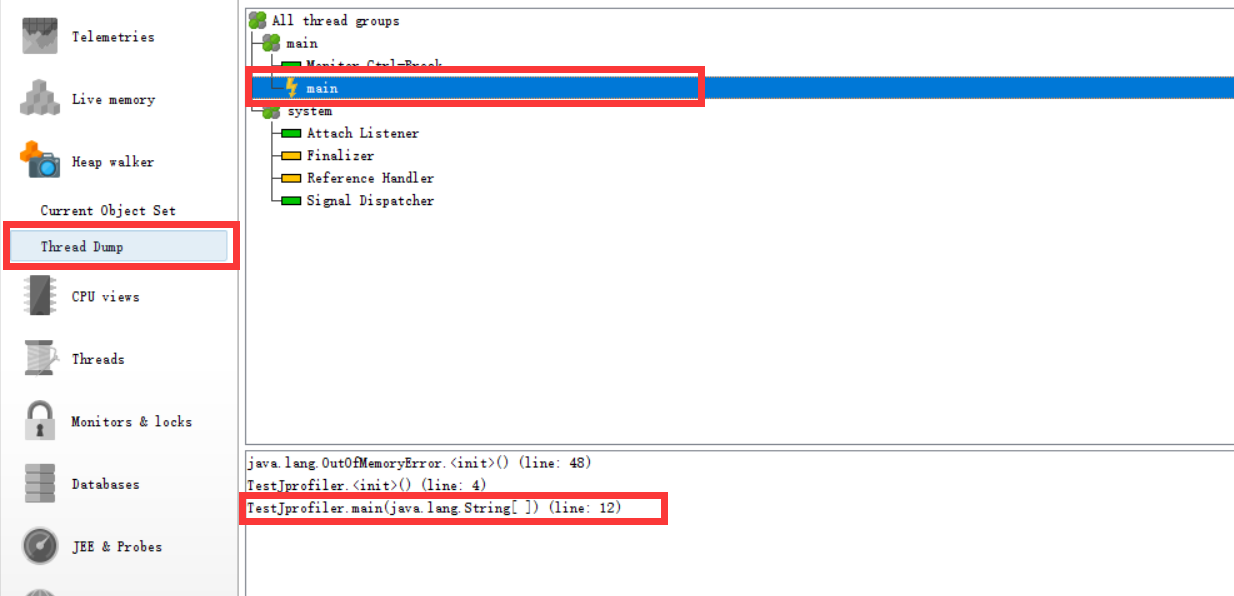


1. 分析profiler文件，选择big Object发现Arraylist占了百分之89，说明存在问题



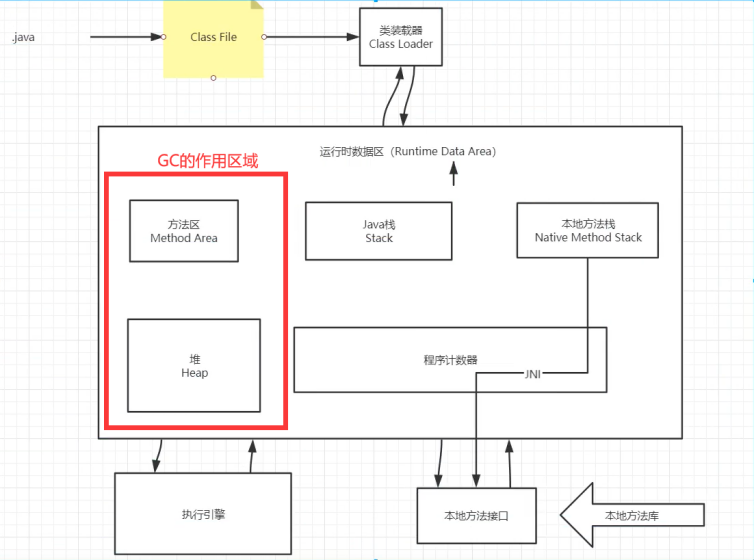
1. 选择Thread Dump会发现是main里面的线程出现了问题

·初始化在第4行，出现问题在12行



**GC垃圾回收：不能够手动操作，手动只能够提醒回收，都是自动回收。**

·GC的作用区域：方法区和堆



JVM在进行GC的时候，并不是对新生代和老年代统一回收，大部分时候只针对新生代。

·新生代

·Eden区

·幸存from区和幸存to区（哪边为空哪边是to区，两边会不停的交换位置，当from区有一个，to区也有一个垃圾的时候，from会将自己的复制到to区，此时为0个，to区两个，from成为新的to区）

·老年代

GC的两种类：轻GC(或称之为普通GC，作用于新生代)和重GC（或称之为全局GC，作用于新生代和老年代）

GC题目：

·轻GC和重GC分别在什么时候发生？

轻GC：Eden区空间不足。

·Eden区满了或者当前要存储的对象大小>Eden所剩空间大小，则会触发轻GC

重GC：老年代空间不足。

·当有大对象在Eden区保存不下的时候，会直接放在老年代中，如果老年代也保存不下则会触发重GC。

·当幸存区对象达到特定年龄后会转移到老年代里面，如果此时老年代空间不足，则会触发重GC。

·JVM的内存模型和分区~详细到分区放什么？

·堆里面的分区有哪些？Eden，from，to，老年区，说说他们的特点！

·jvm怎么判断哪些对象应该回收呢?

判断对象应该不应该回收有两个算法，一个是引用计数法，另一个是可达性分析算法。

引用计数算法：在对象中添加一个引用计数器，每当一个地方引用它时，计数器就加一；当引用失效时，计数器值就减一；任何时刻计数器为零的对象就是不可能再被使用的。但是它有一个问题，就是不能解决循环依赖，当Object 1和Object 2互相引用的时候，他们各自的计数都为1，不会被回收。

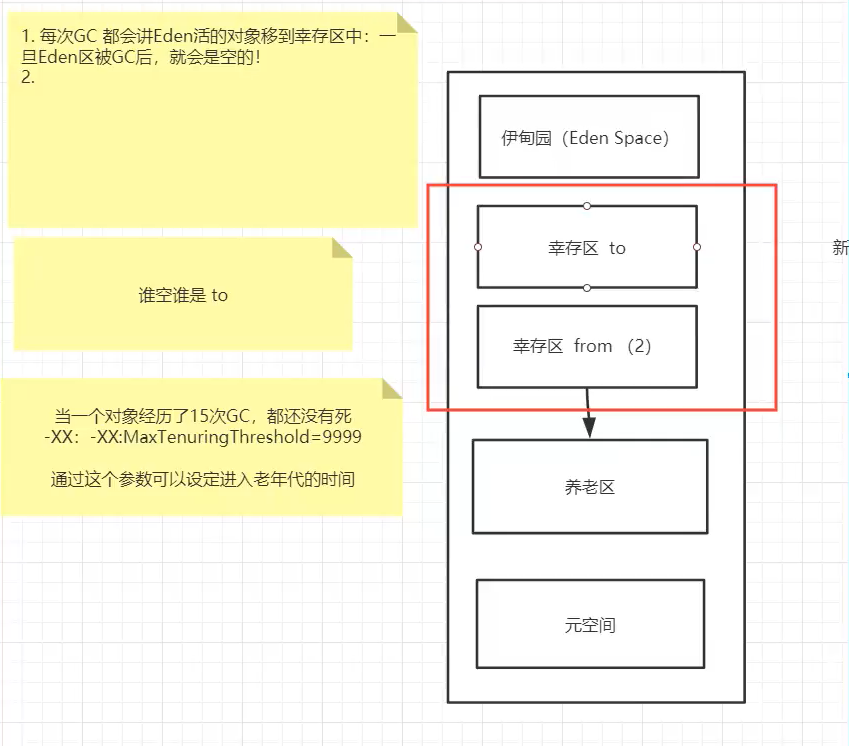
（jvm采用的方法）可达性分析算法：就是通过一系列的“GC Roots”，也就是根对象作为起始节点集合，从根节点开始，根据引用关系向下搜索，搜索过程所走过的路径称为引用链，如果某个对象到GC Roots间没有任何引用链相连。用图论的话来说就是从GC Roots到这个对象不可达时，则证明此对象是不可能再被使用的。所以此对象就是可以被回收的对象。

·GC的算法有哪些？标记清除法，标记整理法(标记压缩法)，复制算法怎么用？

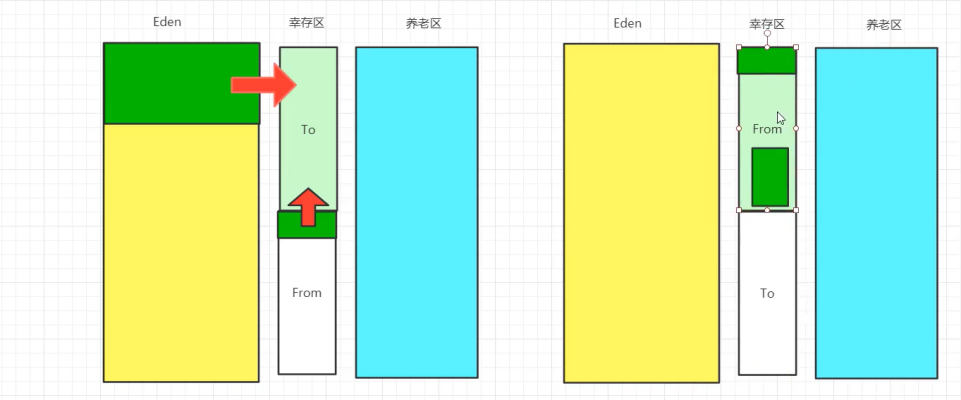
**GC之复制算法（新生代）**：主要运用在新生代的from区和to区里面，负责把from区的对象移到to区，然后让没有对象的from区成为新的to区：

进入老年代的方法：

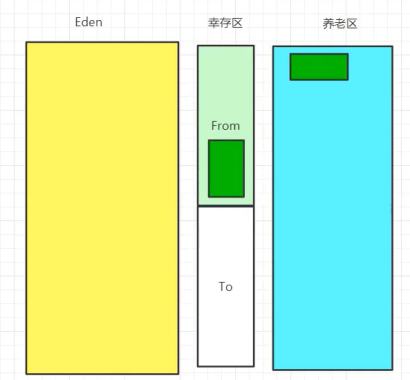
1. 经历15次轻gc后从幸存区到老年代
2. 当对象存储超过了幸存区的百分之50，则直接将部分大年龄的对象送入老年代



复制算法过程图：一开始gc后，会把对象移向to区，然后空的from区成为新的to区，



当对象经历过15次轻gc或者from区满了，对象就会进入养老区



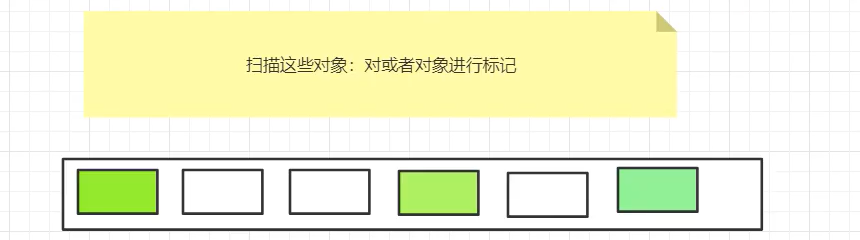
·好处:没有内存的碎片~

·坏处:浪费了内存空间~:多了一半空间永远是空to。

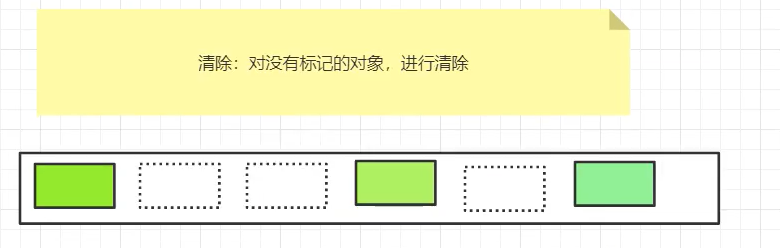
复制算法最佳使用场景:对象存活度较低的时候;新生区~

**标记清除算法：老年代使用**

·扫描对象：回收时对活着的对象进行标记（即扫描从root robots到对象是否有存在引用链，有就标记



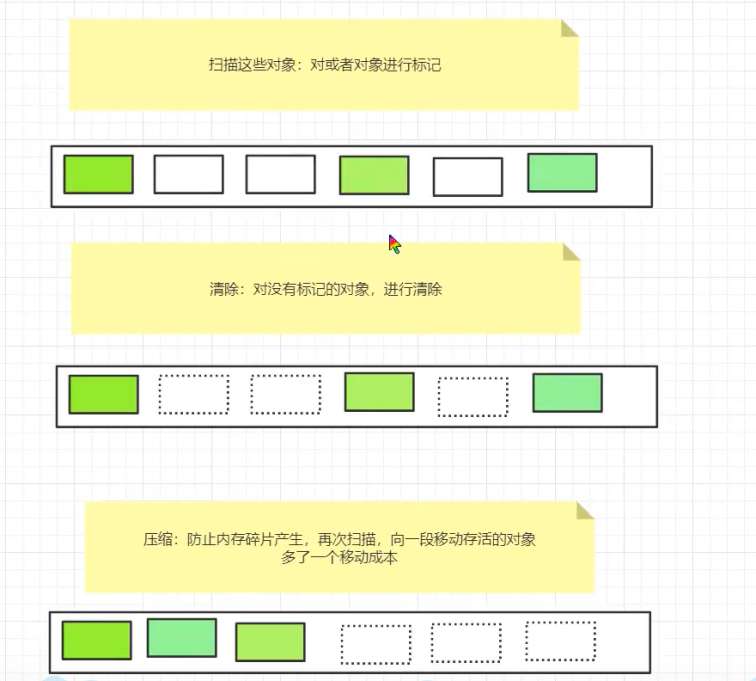
·清除：对没有标记的对象进行清除



优点:不需要额外的空间!

缺点:两次扫描，严重浪费时间，会产生内存碎片。

**标记压缩算法（全称：标记清除压缩算法）**：防止内存碎片的产生，再次扫描，向一端移动存活的对象。



优点：相较于标记清除算法没有了内存碎片。

缺点：多了移动成本。

#考虑一下优化：可以把前面的标记清除多使用几次，使得内存碎片多起来，在使用一次标记压缩。

**总结：**

内存效率：复制算法>标记清除算法>标记压缩算法（时间复杂度）

内存整齐度：复制算法=标记压缩算法>标记清除算法

内存利用率：标记压缩算法=标记清除算法>复制算法

思考一下：难道没有最优的算法嘛？

答案：没有，没有最好的算法，只有最合适的算法---->GC:也被称之为分代收集算法。

年轻代：

·存活率：低

·使用复制算法最为佳

老年代：

·区域大，存活率高。

·标记清除算法（内存碎片不是太多的情况下）+标记压缩算法（二者混合使用） 实现

**JMM:**

1. 什么是JMM？

JMM：java内存模型，Java Memory Model的缩写

1. JMM是做什么的？

作用:缓存一致性协议，用于定义数据读写的规则(遵守，找到这个规则)。

JMM定义了线程和主内存之间的抽象关系:线程之间的共享变量存储在主内存(Main Memory)中，每个线程都有一个私有的本地内存(Local Memory)。

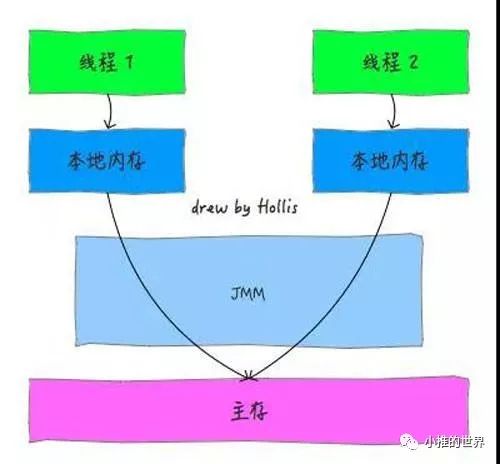
详细如下：

Java 内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存。

线程的工作内存中保存了该线程中用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。

不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行。

而 JMM 就作用于工作内存和主存之间数据同步过程。它规定了如何做数据同步以及什么时候做数据同步。



总结下来，JMM 是一种规范，是解决由于多线程通过共享内存进行通信时，存在的本地内存数据不一致、编译器会对代码指令重排序、处理器会对代码乱序执行等带来的问题。

·它该如何学习?

JMM︰抽象的概念，理论

JMM对这八种指令的使用，制定了如下规则:

·不允许read和load、store和write操作之一单独出现。即使用了read必须load，使用了store必须write。

·不允许线程丢弃他最近的assign（赋值）操作，即工作变量的数据改变了之后，必须告知主存

·不允许一个线程将没有assign（赋值）的数据从工作内存同步回主内存

·一个新的变量必须在主内存中诞生，不允许工作内存直接使用一个未被初始化的变量。就是对变量实施use、store操作之前，必须经过assign和load操作

·一个变量同一时间只有一个线程能对其进行lock。多次lock后，必须执行相同次数的unlock才能解锁

·如果对一个变量进行lock操作，会清空所有工作内存中此变量的值，在执行引擎使用这个变量前，必须重新load或assign操作初始化变量的值

·如果一个变量没有被lock，就不能对其进行unlock操作。也不能unlock一个被其他线程锁住的变量

·对一个变量进行unlock操作之前，必须把此变量同步回主内存

JMM对这八种操作规则和对volatile的一些特殊规则就能确定哪里操作是线程安全，哪些操作是线程不安全的了。但是这些规则实在复杂，很难在实践中直接分析。所以一般我们也不会通过上述规则进行分析。更多的时候，使用java的happen-before规则来进行分析。

**volatile关键字的作用**

        ·保证了新值能立即存储到主内存，每次使用前立即从主内存中刷新。   
        ·禁止指令重排序优化。   
        注：volatile关键字不能保证在多线程环境下对共享数据的操作的正确性。可以使用在自己状态改变之后需要立即通知所有线程的情况下。

final域

        final类型的域是不能修改的，除了这一点外，在Java内存模型中，final域还有着特殊的语义，final域能确保初始化过程的安全性，从而可以不受限制地访问不可变对象，并在共享这些对象时无须同步。具体而言，就是被final修饰的字段在构造器中一旦被初始化完成，并且构造器没有把“this”的引用传递出去（this引用逃逸是一件很危险的事情，其他线程有可能通过这个引用访问到“初始化了一半”的对象），那么在其他线程中就能看到final字段的值，而且其外、外部可见状态永远也不会改变。它所带来的安全性是最简单最纯粹的。