JVM

# 概述

JVM官方文档

https://docs.oracle.com/javase/specs/jvms/se8/html/index.html

JVM(Java Virtual Machine) 中文名字叫做Java虚拟机

作用是用来保证Java语言的跨平台执行

JVM可以看作的一台抽象的计算机，如真实的计算机那样具有自己的指令集及运行时的内存区域

JVM与Java并没有必然的联系，它只是与特定格式的二进制文件有关联(class)，可以将源码编译成class文件的除了Java还有Kotlin，Scalac等编程语言

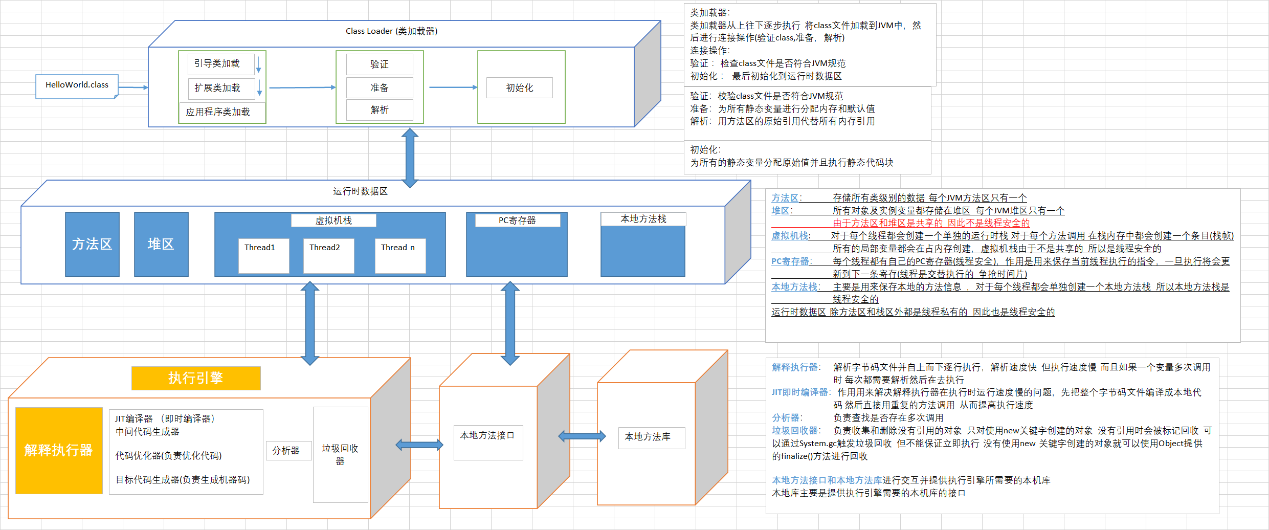
JVM是字节码编译器 它将字节码文件编译成各个不同平台对应的机器码，确保字节码文件能够在各系统正确运行。

# 为什么要学习JVM

JVM就像做数学，物理，化学题时用到各种公式，我们要知其然也要知其所以然，那么在没有公式的时候我们才可以同样保证做题的质量。

在编程中 我们为了能够编写出效率更高的应用程序 必然需要了解应用程序的运行原理及运行时的内存构造等，而这些都包含在JVM中。

# JVM体系结构



## 类加载器

负责将class文件加载进JVM，然后进行连接操作

连接操作主要是用来校验class文件是否符合JVM规范，符合则进行初始化 不符合则抛出编译异常

## 运行时数据区

### 分为方法区和栈区

由于方法区和栈区是共享的 因此线程不安全的

方法区存放类级别的数据 （全局变量）

栈区存放对象和实例的数据（局部变量）

### 虚拟机栈

对每个线程都会创建一个单独的运行时线程，针对每个方法的调用 在栈内存中都会创建一个条目(栈帧) 所有的局部变量都在栈内存中创建，虚拟机栈是线程安全的，它不是共享的

### PC寄存器

每个线程都有自己的PC寄存器(线程安全) 作用的用来保存当前线程的执行指令，一旦执行将更新到下一条寄存

本地方法栈：用来保存本地方法信息，对于每个线程都会单独创建一个本地方法栈，因此本地方法栈也是线程安全的

运行时数据区除了方法区和栈区外，其它都是线程私有的，也是线程安全的

## 执行引擎

### 解析执行器

解析字节码文件并自上而下逐行执行，解析速度快但执行速度慢，因为当一个变量多次被调用时 每次都会进行解析执行。

### JIT即时编译器

其目的是用来解决解析执行器执行效率慢的问题，它是先把字节码文件编译成本地代码 然后直接重复的方法调用 从而提高执行速度

### 分析器

负责查找是否存在多次调用

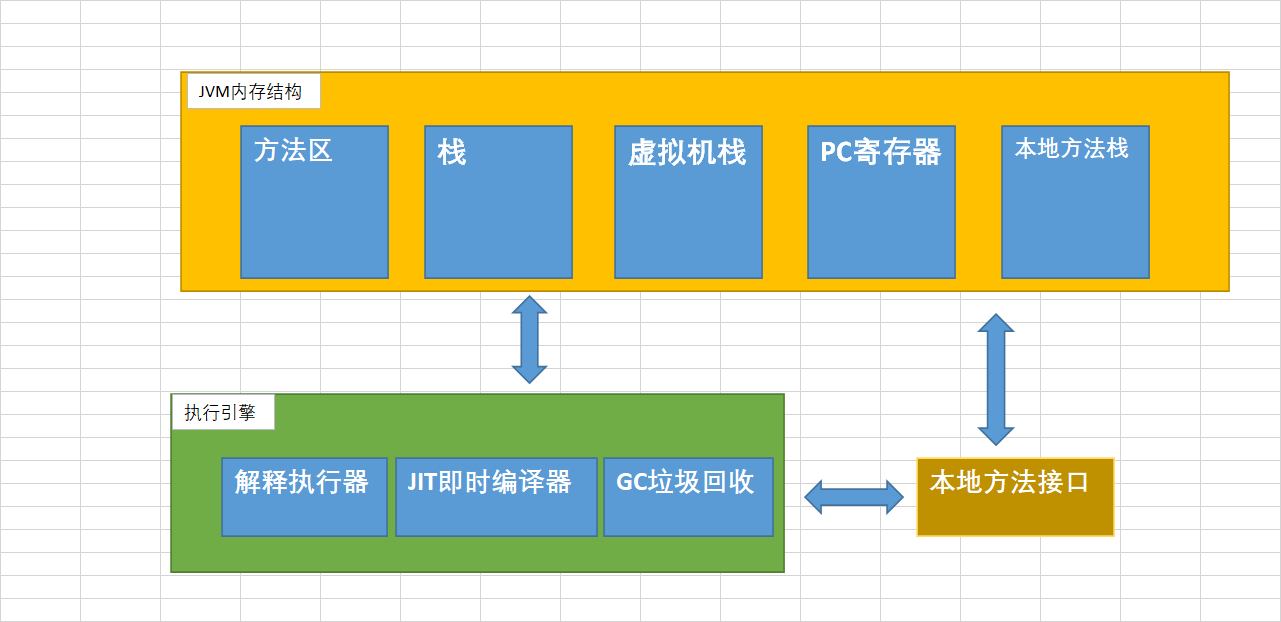
### 垃圾回收器

负责收集和清理没有引用的对象，我们通过new关键字创建的对象可以使用System.gc()方法主动触发垃圾回收机制，如果是非new出来的对象 则可以直接使用Object类提供的finalize()方法进行垃圾回收

### 本地方法接口和本地库

本地方法接口和本地库进行交互并提供执行引擎所需要的本机库，本地库主要是提供执行引擎需要的本机库的接口

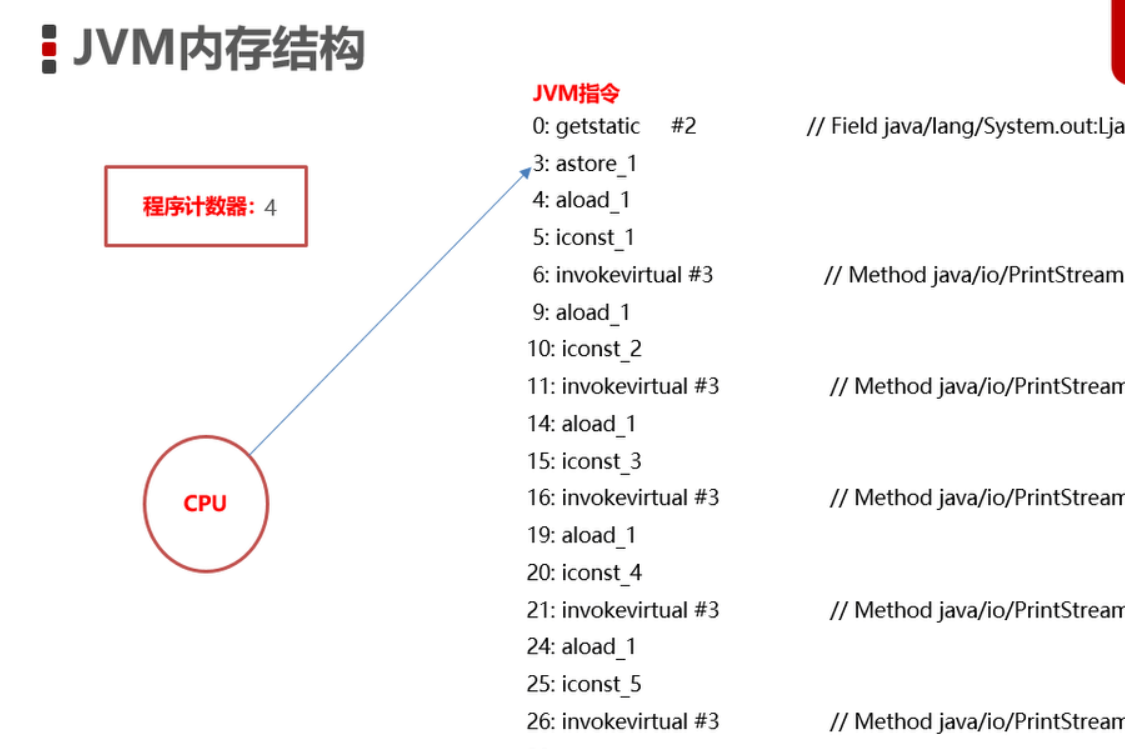
# JVM内存结构



## 程序计数器

程序计数器也叫做PC寄存器

作用是保存当前执行指令的地址，一旦指令被执行 程序计数器就会更新到下一条指令



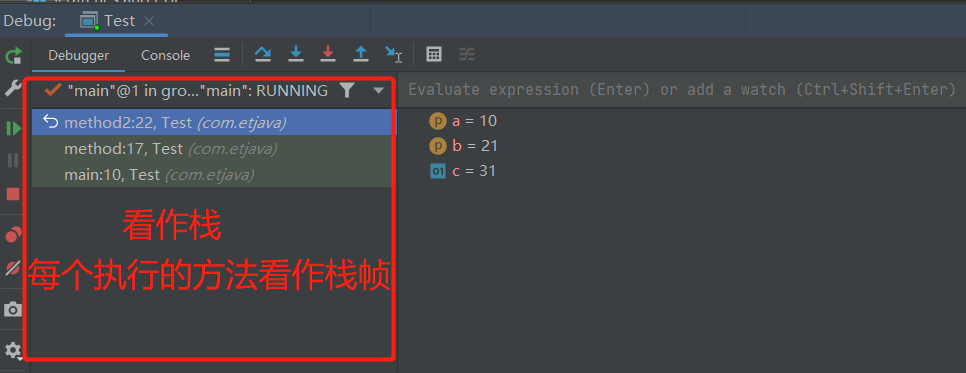
## 虚拟机栈

每个线程运行时所需要的内存空间 称之为虚拟机栈

每个栈由多个栈帧组成，对应着每次方法调用所占用的内存

每个栈只能有一个活动栈帧，对应着当前正在执行的那个方法

执行结束后 栈中的每个栈帧遵循先进后出的原则逐个退出栈(释放内存)



## 栈帧

栈帧是由局部变量表，操作数栈，动态连接，方法返回地址组成

### 局部变量表

存放局部变量的列表

一个局部变量可以保存类型为:

boolean,byte,char,short,float ,reference,returnAddress的数据

两个局部变量可以保存一个类型为double和long类型的数据

局部变量使用索引进行定位访问的，第一个局部变量的索引值为0

### 操作数栈

操作数栈也叫做操作栈，它是一个先进后出的栈

当一个方法刚刚开始执行的时候 操作数栈中是空的，随着方法和字节码指令的执行 会从局部变量表或对象实例的字段中复制常量或变量写入到操作数栈，紧跟着计算的进行将栈中元素出栈到局部变量表或返回给方法调用者，也就是出栈、入栈操作 一个完整的方法执行期间往往会包含多个这样的出栈 入栈的过程。

**简单理解 操作栈就是线程的实际操作台。**

### 动态连接

动态链接就是指向运行时常量池的引用

在class文件中 描述一个方法调用了其它方法 或者访问其它成员变量是通过符号引用来表示的 动态连接的作用就是将这些符号引用所表示的方法转换成实际方法的直接引用。

### 方法返回地址

方法的调用和返回包括正常返回(有返回值) 和异常返回(没有返回值) 不同的返回类型有不同的指令

无论采用那种方式退出，在方法推出后都需要返回到方法被调用的位置，程序才能继续执行，方法返回时可能需要在当前栈帧中保存一些信息，用来帮它恢复它的上层方法执行状态

# 栈内存溢出

栈内存溢出常见有两种，一种是栈帧过多，一种是栈帧过大

栈内存大小可以通过指令进行设置的

-Xss size 来指定栈内存大小 默认单位是bytes 可以使用k或KB或MB又或是GB来设置栈内存的大小

linux64 默认1024KB

Solaris 默认 1024KB

windows 默认是值依赖于JVM虚拟机的内存大小

可以这样写

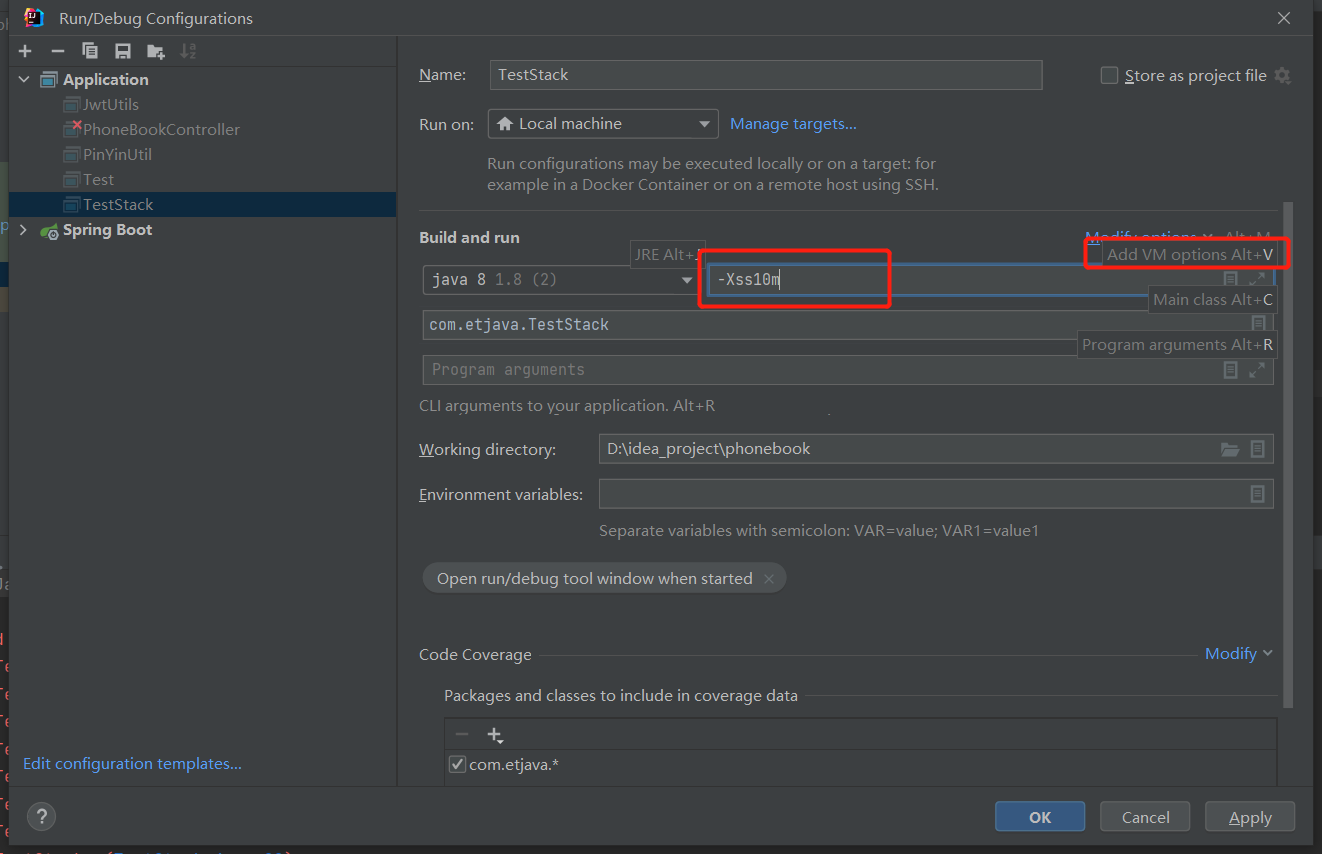
-Xss 1m

-Xss 1024k

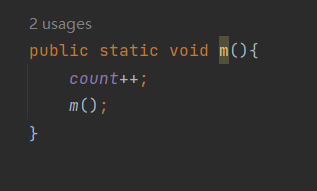
-Xss 1048576

## 栈帧过多

IDE中可以通过调整VM Options参数控制栈内存大小



递归但没有程序出口 所以导致无线递归 因此产生了过多的栈帧



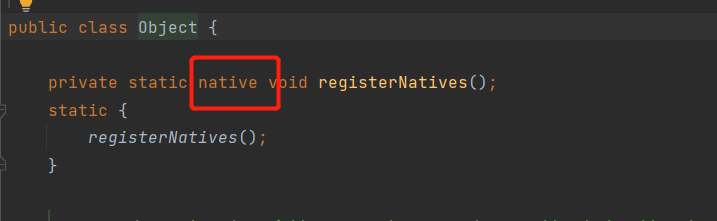
## 栈帧过大

栈帧过大通常是出现了过多的嵌套引用导致的，例如A找到B 然后根据条件又找到了A 那么到A中执行的时候根据条件又取到了B 然后一致这样循环下来就会导致栈帧过大 也就是我们说的死循环

# 本地方法栈

被native修饰的方法就是本地方法

例如Object类中就有很多本地方法



本地方法栈的功能和特点类似虚拟机栈，线程也是私有的

不同的是本地方法栈服务的对象是JVM执行的native方法，而虚拟机栈服务的是JVM执行的Java方法

如何服务本地方法(native)，本地方法使用什么语言实现，如何组织像栈帧这种为了服务方法的数据结构 JVM并未给出墙纸规定，因此不同的虚拟机可以进行自由实现

# 堆

JVM中的堆 是JVM管理的最大的一块内存空间

堆是用来存放对象的内存区域

特点

1堆是被所有线程共享的一块内存区域，在虚拟机启动时创建

2堆的区域是用来存放对象实例的 因此也是垃圾回收器管理的主要区域

3 堆在逻辑上又划分为新生代和老年代 其中新生代又划分为Eden区，ServivorFrom区和ServivorTo区 三个分区

(Java中的对象大部分是朝生夕灭的 还有一部分对象是能够长期驻留在内存中的 为了对这两种对象进行有效的回收那么垃圾回收器九江堆划分为新生代和老年代并且执行不同的回收策略 不同的垃圾收集器对这两个逻辑区的回收机制也是不相同的)

4 堆的大小是可扩展的，使用-Xms 和 -Xmx来调整堆的大小

# 堆内存溢出

堆内存中存放的是对象实例，如果对象的实例一直没有释放 并且在不断产生新的实例 那么就会导致堆内存溢出

堆的错误提示

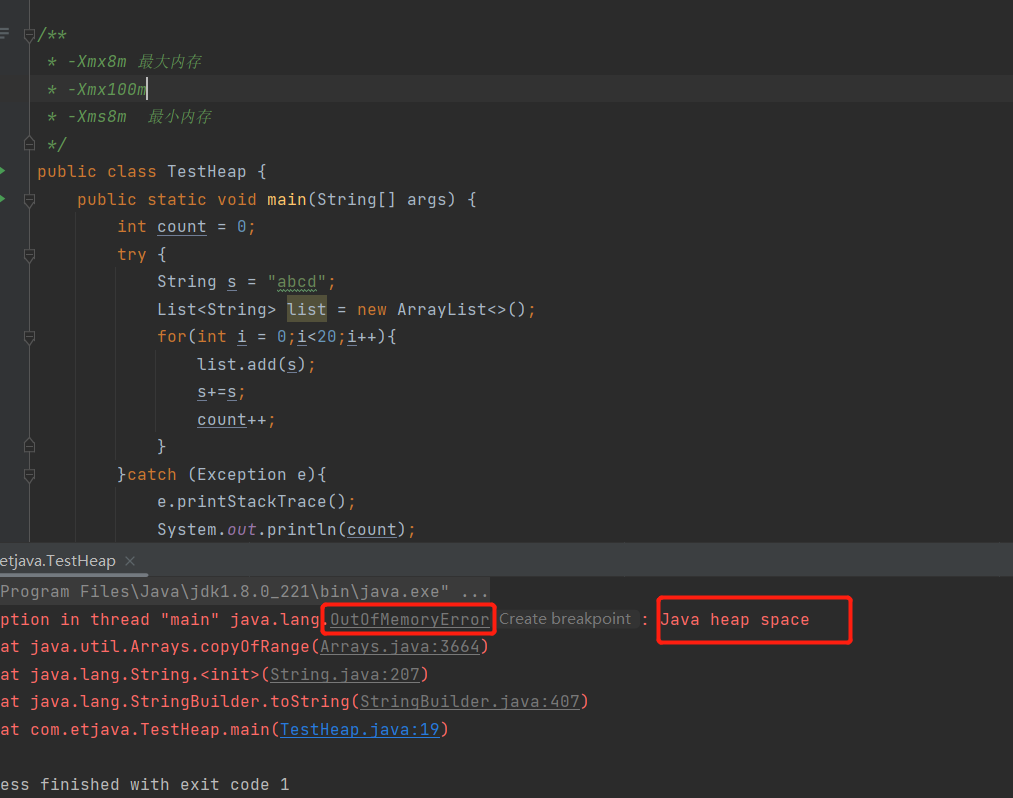
java.lang.OutOfMemoryError: Java heap space

错误原因有两种 一种是内存确实不够用 一种是死循环导致的堆溢出

针对第一种情况可以通过调整堆内存大小来控制(死循环除外 例如 while(true){})

-Xmx100m 表示堆最大内存

-Xms10m 表示堆最小内存



# 堆内存的诊断

## 常用工具

jps，jmap,jconsole 都是Java自带的查看堆内存查看工具 可以直接使用指令在dos窗口执行即可

通过下面这段代码分析堆内存使用情况

第一次执行是程序启动 然后休眠30s ,

第二次执行是休眠30s之后创建一个10m大小的数组然后休眠10s

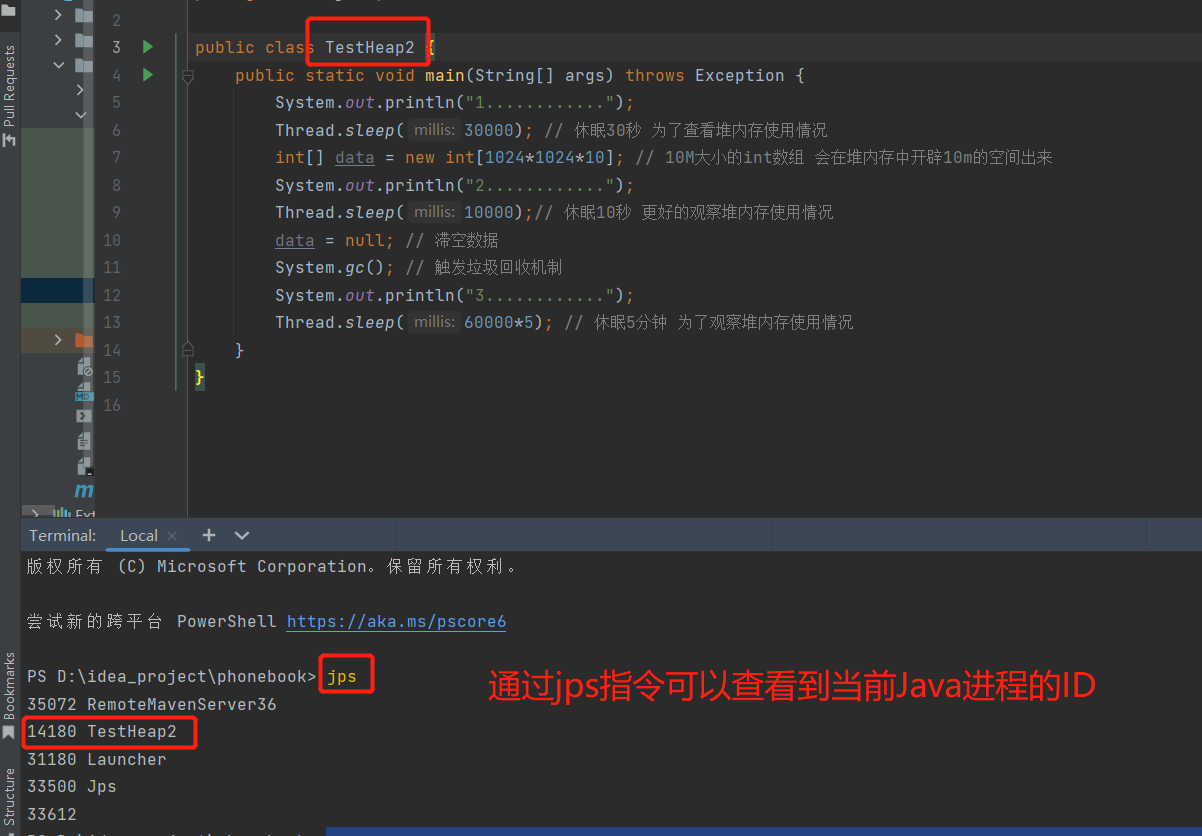
第三次执行休眠10s之后将数组滞空并触发垃圾回收机制



### jps

jps 查看当前系统中有哪些Java进程

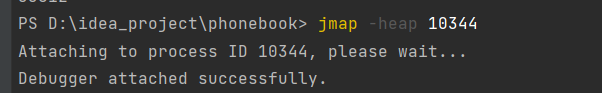
命令 jps



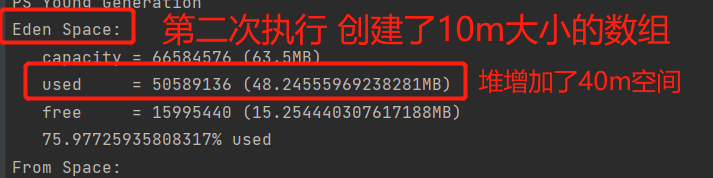
### jmap

jmap 查看堆内存占用情况(某一进程某一时刻的占用情况)

命令 jmap -heap 进程ID







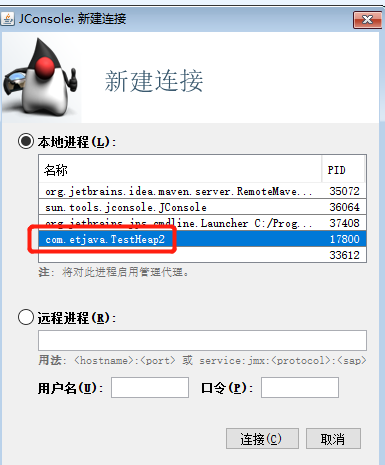
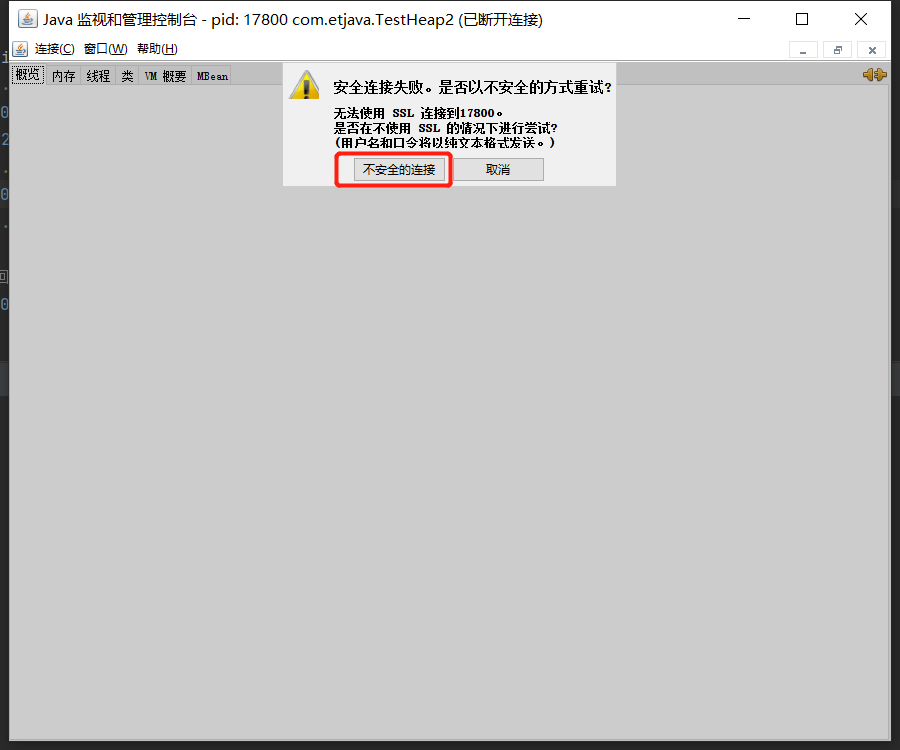


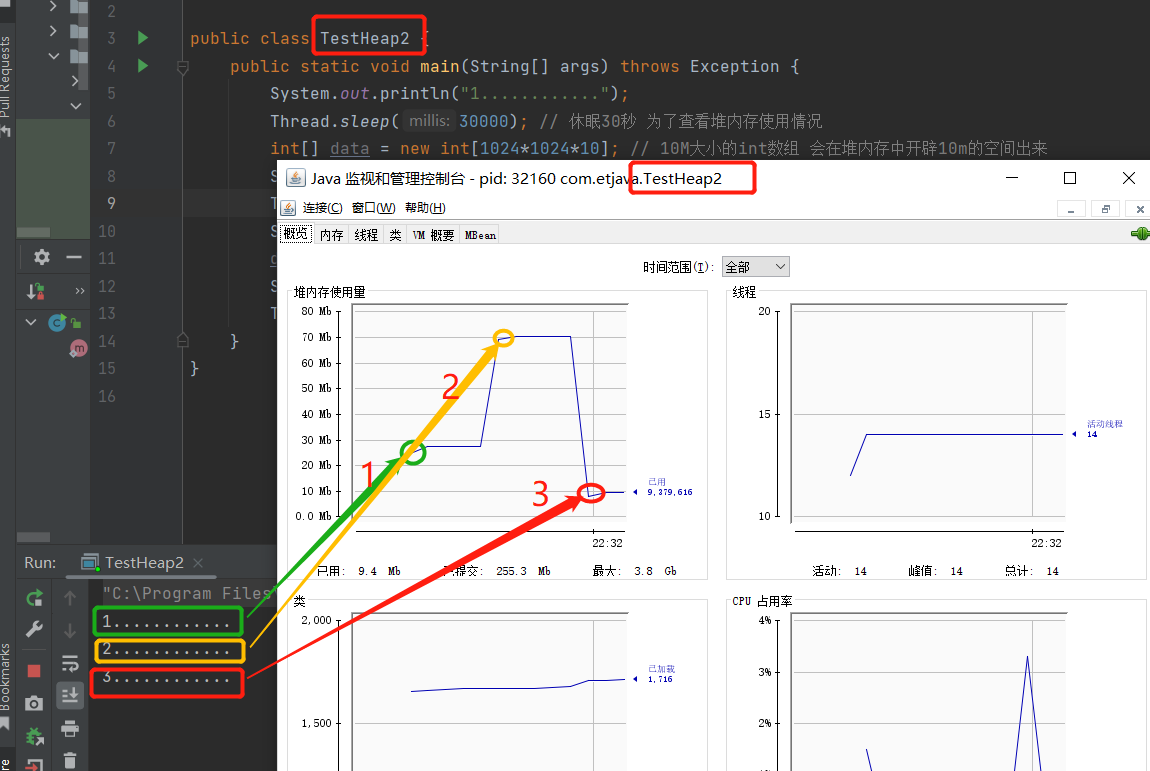
### jconsole

jconsole 图形化的，内置Java性能分析器 多功能的监测工具 可以连续监测堆内存使用情况

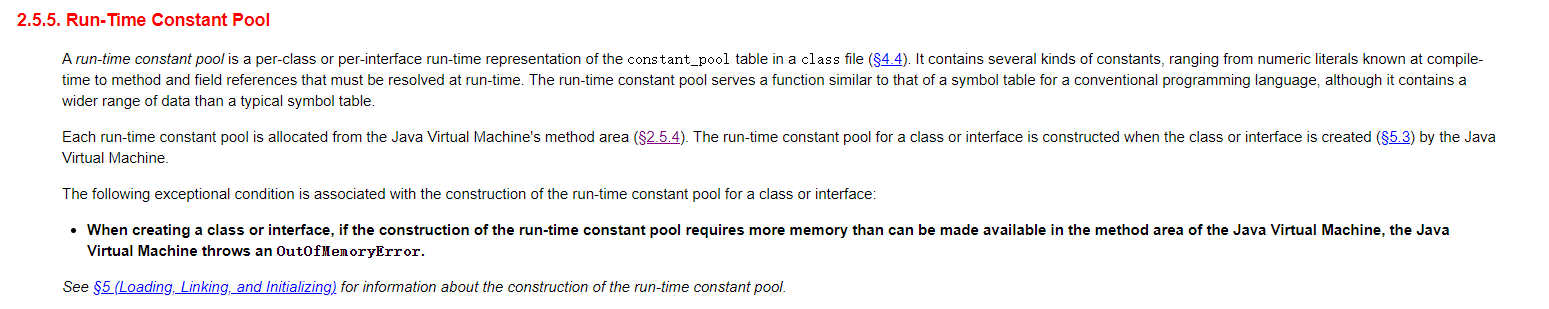
命令 jconsole

项目启动 然后在dos中输入jconsole即可



# 方法区

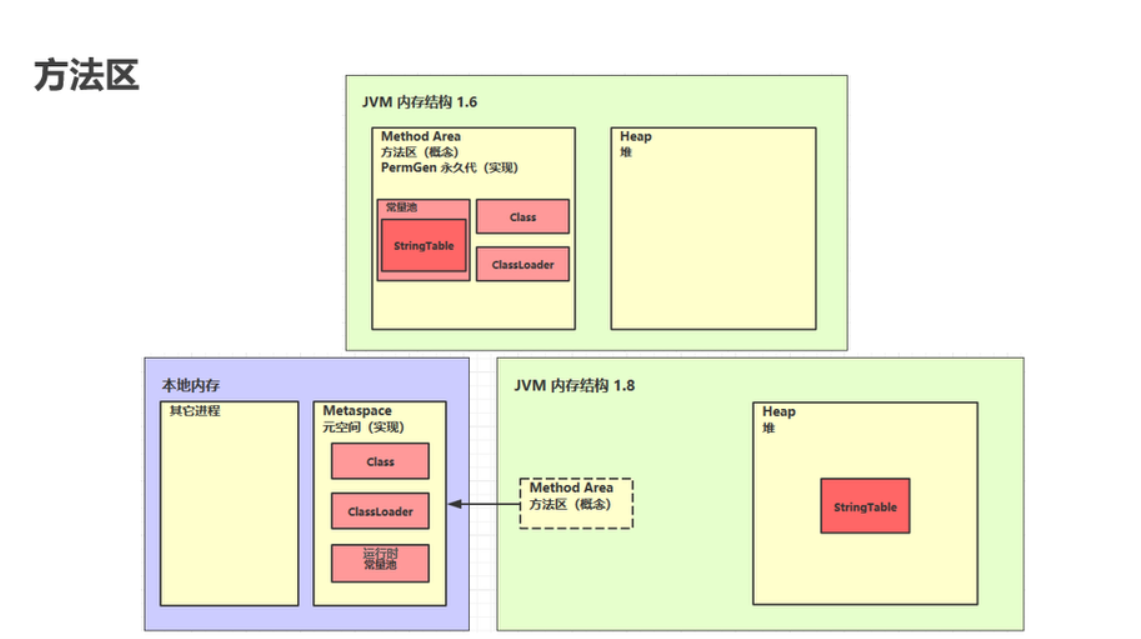


作用是用来存储每个类的结构

例如运行时常量池，字段和方法数据及方法和构造方法 同时还包含了用于类和实例初始化及接口初始化的特殊方法等

方法区的实现与JVM版本有关，

6.0时 方法区采用永久代方式实现，在方法区中常量池包含了字符串常量池，而到了8.0 改用元空间实现方法区，将常量池放到了本地内存中，但把字符串常量池放到了堆空间 这也是为什么上边我们对一个字符串不停追加时会导致堆溢出原因所在 如下图所示



# 方法区内存溢出

## 8.0方法区溢出

调整方法区大小

-XX:MaxMetaspaceSize=8m -XX:MaxMetaspaceSize=10m -XX:-UseCompressedOops

java.lang.OutOfMemoryError: Metaspace

|  |
| --- |
| package com.etjava;  import com.sun.org.apache.xpath.internal.compiler.OpCodes;  import jdk.internal.org.objectweb.asm.ClassWriter;  import jdk.internal.org.objectweb.asm.Opcodes;  /\*  8.0 调整大小 为了演示方法区溢出  -XX:MaxMetaspaceSize=8m -XX:MaxMetaspaceSize=10m -XX:-UseCompressedOops  方法区是用来存放类的结构 这里为了演示方法区溢出 模拟一个类加载器  \*/  public class TestSpace extends ClassLoader{  public static void main(String[] args) {  int count = 0;  try {  TestSpace t = new TestSpace();  for(int i=0;i<20000;i++,count++){  // 生成类的二进制字节码  ClassWriter c = new ClassWriter(i);  // 版本号，方法修饰符，类名，包名，父类，接口  c.visit(Opcodes.V1\_8,Opcodes.ACC\_PUBLIC,"Class"+i,null,"java/lang/Object",null);  byte[] code = c.toByteArray();  // 执行类的加载  t.defineClass("Class"+i,code,0,code.length);  }  }catch (Exception e){  e.printStackTrace();  System.out.println(count);  }  }  } |

## 6.0方法区溢出

通过-XX:MaxPermSize=8m 调整方法区大小 然后再通过上述8.0demo演示即可 前提是需要有6.0JDK（因为每个版本的JDK都会自带对于的JVM）

6.0方法区溢出报错信息 permGen space