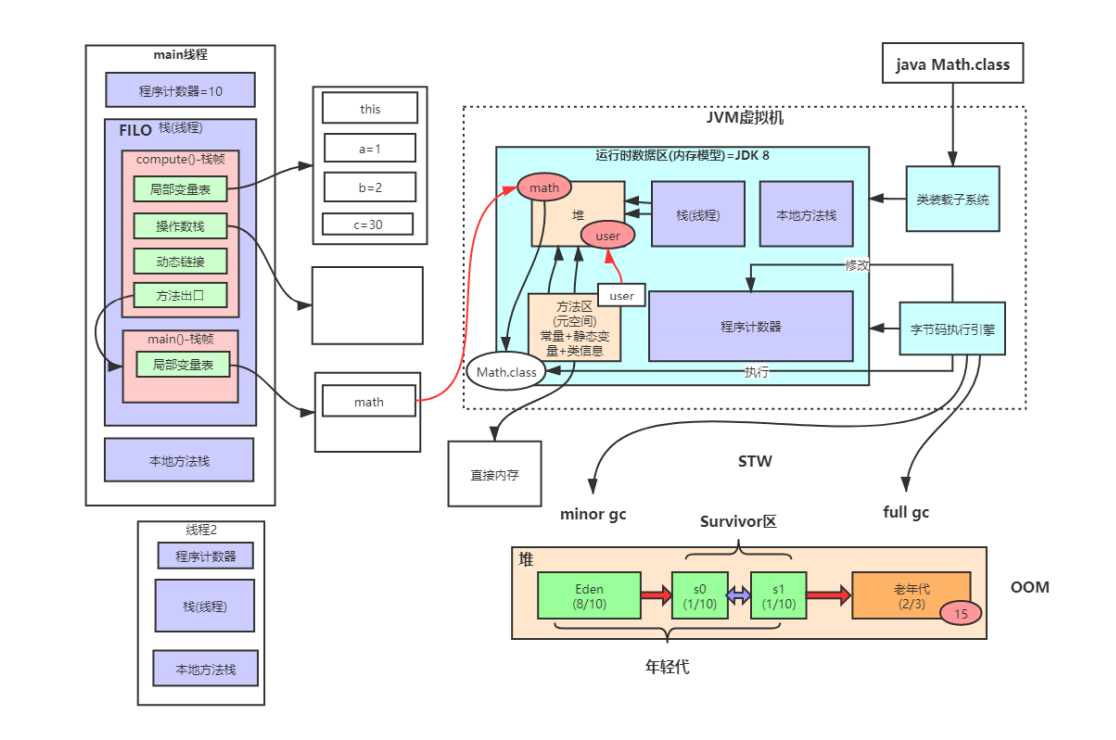


JVM内存结构



JVM的内存模型如上图，运行时数据区分为本地方法栈、线程栈、堆、方法区程和序计数器（ProgramCounter寄存器）。

1. 方法栈（Java栈）：
   1. Java栈的区域很小，只有1M，特点是存取速度很快，所以在stack中存放的都是快速执行的任务，基本数据类型的数据，和对象的引用（reference）。
   2. 驻留于常规RAM（随机访问存储器）区域。但可通过它的“栈指针”获取处理的直接支持。栈指针若向下移，会创建新的内存；若向上移，则会释放那些内存。这是一种特别快、特别有效的数据保存方式，仅次于寄存器。创建程序时，Java编译器必须准确地知道堆栈内保存的所有数据的“长度”以及“存在时间”。这是由于它必须生成相应的代码，以便向上和向下移动指针。这一限制无疑影响了程序的灵活性，所以尽管有些Java数据要保存在栈里——特别是对象句柄，但Java对象并不放到其中。
   3. JVM只会直接对JavaStack（Java栈）执行两种操作：①以帧为单位的压栈或出栈；②通过-Xss来设置， 若不够会抛出StackOverflowError异常。
   4. 每个线程包含一个栈区，栈中只保存基本数据类型的数据和自定义对象的引用(不是对象)，对象都存放在堆区中
   5. 每个栈中的数据(原始类型和对象引用)都是私有的，其他栈不能访问。
   6. 栈分为3个部分：基本数据类型的变量区、执行环境上下文、操作指令区(存放操作指令)。
   7. 栈是存放线程调用方法时存储局部变量表、操作、方法出口等与方法执行相关的信息，Java栈所占内存的大小由Xss来调节，方法调用层次太多会撑爆这个区域。
2. 程序计数器（ProgramCounter）寄存器
   1. PC寄存器（PCregister）：每个线程启动的时候，都会创建一个PC（ProgramCounter，程序计数器）寄存器。PC寄存器里保存有当前正在执行的JVM指令的地址。每一个线程都有它自己的PC寄存器，也是该线程启动时创建的。保存下一条将要执行的指令地址的寄存器是：PC寄存器。PC寄存器的内容总是指向下一条将被执行指令的地址，这里的地址可以是一个本地指针，也可以是在方法区中相对应于该方法起始指令的偏移量。
3. 本地方法栈
   1. Nativemethodstack(本地方法栈)：保存native方法进入区域的地址。
4. 堆
   1. 类的对象放在heap（堆）中，所有的类对象都是通过new方法创建，创建后，在stack（栈）会创建类对象的引用（内存地址）。
   2. 一种常规用途的内存池（也在RAM（随机存取存储器）区域），其中保存了Java对象。和栈不同：“内存堆”或“堆”最吸引人的地方在于编译器不必知道要从堆里分配多少存储空间，也不必知道存储的数据要在堆里停留多长的时间。因此，用堆保存数据时会得到更大的灵活性。要求创建一个对象时，只需用new命令编辑相应的代码即可。执行这些代码时，会在堆里自动进行数据的保存。当然，为达到这种灵活性，必然会付出一定的代价：在堆里分配存储空间时会花掉更长的时间。
   3. JVM将所有对象的实例（即用new创建的对象）（对应于对象的引用（引用就是内存地址））的内存都分配在堆上，堆所占内存的大小由-Xmx指令和-Xms指令来调节，sample如下所示：

public class HeapOOM {

static class OOMObject {

}

public static void main(String[] args) {

List list = new ArrayList();

while (true) {

list.add(new OOMObject());

}

}

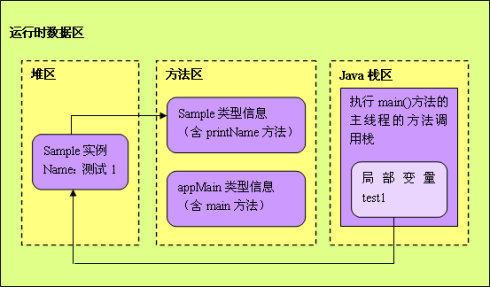
}

加上JVM参数-verbose:gc -Xms10M -Xmx10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:+HeapDumpOnOutOfMemoryError，就能很快报出OOM异常（内存溢出异常）：Exception in thread "main" java.lang.OutOfMemoryError: Java heap space并且能自动生成Dump。

1. 方法区（元空间）
   1. method（方法区）又叫静态区，存放所有的①类（class），②静态变量（static变量），③静态方法，④常量和⑤成员方法。
   2. 又叫静态区，跟堆一样，被所有的线程共享。
   3. 方法区中存放的都是在整个程序中永远唯一的元素。这也是方法区被所有的线程共享的原因。
   4. （顺便展开静态变量和常量的区别：静态变量本质是变量，是整个类所有对象共享的一个变量，其值一旦改变对这个类的所有对象都有影响；常量一旦赋值后不能修改其引用，其中基本数据类型的常量不能修改其值。）
   5. Java里面是没有静态变量这个概念的，不信你自己在某个成员方法里面定义一个static int i=0；Java里只有静态成员变量。它属于类的属性。至于他放哪里？楼上说的是静态区。我不知道到底有没有这个翻译。但是深入JVM里是翻译为方法区的。虚拟机的体系结构：①Java栈，②堆，③PC寄存器，④方法区，⑤本地方法栈，⑥运行常量池。而方法区保存的就是一个类的模板，堆是放类的实例（即对象）的。栈是一般来用来函数计算的。随便找本计算机底层的书都知道了。栈里的数据，函数执行完就不会存储了。这就是为什么局部变量每一次都是一样的。就算给他加一后，下次执行函数的时候还是原来的样子。
   6. 方法区的大小由-XX:PermSize和-XX:MaxPermSize来调节，类太多有可能撑爆永久代。静态变量或常量也有可能撑爆方法区。

package com.test;  
public class AppMain {  
 public static void main(String[] args) { //main成员方法本身放入方法区。  
 Sample test1 = new Sample(" 测试1 "); //test1是引用，所以放到栈区里，Sample是自定义对象应该放到堆里面  
 Sample test2 = new Sample(" 测试2 ");  
 test1.printName();  
 test2.printName();  
 }  
}  
// Sample.java  
class Sample { //运行时，JVM把appmain的信息都放入方法区。  
 private String name; //new Sample实例后，name引用放入栈区里，name对象放入堆里。  
 public Sample(String name) {  
 this.name = name;  
 }  
 public void printName() {// printName()成员方法本身放入方法区里。  
 System.*out*.println(name);  
 }  
}

发出指令：“java AppMain”



1. 系统收到了我们发出的指令，启动了一个Java虚拟机进程，这个进程首先从classpath中找到AppMain.class文件，读取这个文件中的二进制数据，然后把Appmain类的类信息存放到运行时数据区的方法区中。这一过程称为AppMain类的加载过程。
2. 接着，JVM定位到方法区中AppMain类的Main()方法的字节码，开始执行它的指令。这个main()方法的第一条语句就是：

Sample test1 = new Sample("测试1");

1. 语句很简单，就是让JVM创建一个Sample实例，并且使引用变量test1引用这个实例。
   1. Java虚拟机去方法区（方法区存放已经加载的类的相关信息，如类、静态变量和常量）找Sample类的类型信息。这会儿的方法区里还没有Sample类（即Sample类的类信息还没有进入方法区中）。然后加载Sample类，把Sample类的相关信息存放在了方法区中。
   2. Sample类的相关信息加载完成后。Java虚拟机做的第一件事情就是在堆中为一个新的Sample类的实例分配内存，这个Sample类的实例持有着指向方法区的Sample类的类型信息的引用（Java中引用就是内存地址）。这里所说的引用，实际上指的是Sample类的类型信息在方法区中的内存地址
   3. 在JVM中的一个进程中，每个线程都会拥有一个方法调用栈，用来跟踪线程运行中一系列的方法调用过程，栈中的每一个元素被称为栈帧，每当线程调用一个方法的时候就会向方法栈中压入一个新栈帧。这里的帧用来存储方法的参数、局部变量和运算过程中的临时数据。位于“=”前的test1是一个在main()方法中定义的变量，它是一个局部变量，因此，test1这个局部变量会被JVM添加到执行main()方法的主线程的Java方法调用栈中。而“=”将把这个test1变量指向堆区中的Sample实例，也就是说，test1这个局部变量持有指向Sample类的实例的引用（即内存地址）。
   4. 接下来，JVM将继续执行后续指令，在堆区里继续创建另一个Sample类的实例，然后依次执行它们的printName()方法。当JVM执行test1.printName()方法时，JVM根据局部变量test1持有的引用，定位到堆中的Sample类的实例，再根据Sample类的实例持有的引用，定位到方法区中Sample类的类型信息（包括①类，②静态变量，③静态方法，④常量和⑤成员方法），从而获取printName()成员方法的字节码，接着执行printName()成员方法包含的指令。