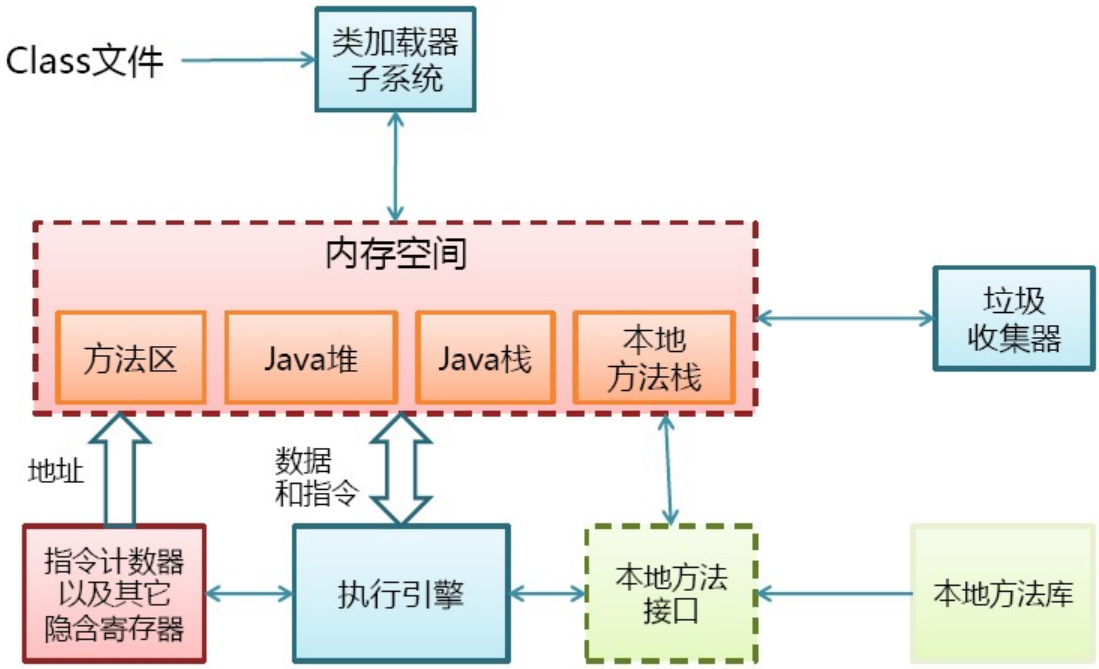
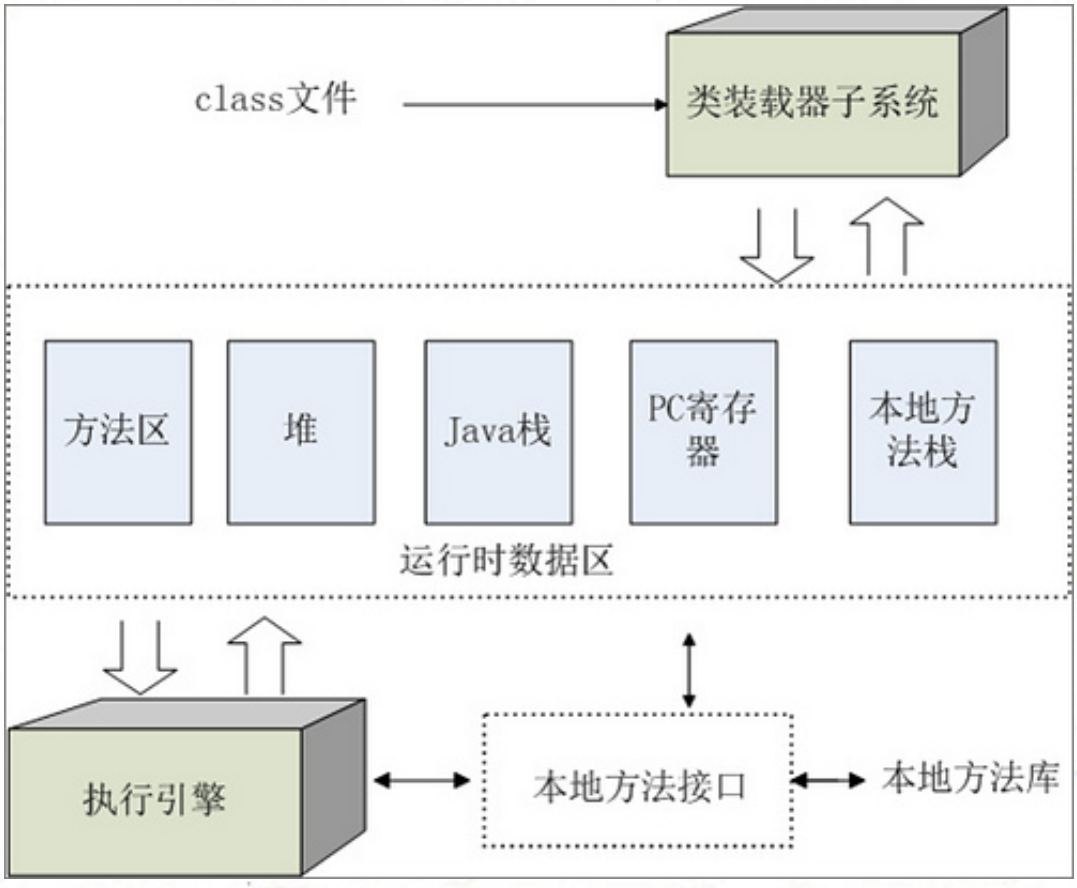
https://my.oschina.net/wangsifangyuan/blog/711329

最近在看《深入理解Java虚拟机》，书中给了几个例子，比较好的说明了几种OOM（OutOfMemory）产生的过程，大部分的程序员在写程序时不会太关注Java运行时数据区域的结构：



感觉有必要通过几个实在的例子来加深对这几个区域的了解。

Java程序运行时，数据会分区存放，JavaStack（Java栈）、 heap（堆）、 method（方法区）。



1、Java栈

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）。

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。

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。

1.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

2.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。

3.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)。

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

PC寄存器（PC register）：每个线程启动的时候，都会创建一个PC（Program Counter，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

Nativemethodstack(本地方法栈)：保存native方法进入区域的地址。

4、堆

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）。

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。

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{}
    /**
     * @param args
     */
    public static void main(String[] args) {
        List list = new ArrayList();// List类和ArrayList类都是集合
```

类，

```
                                                    // 但是ArrayList可以理解为顺序表，
                                                    // 属于线性表。
        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。

5、方法区

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(final修饰,不可变)和⑤成员方法。

1.又叫静态区，跟堆一样，被所有的线程共享。

2.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。

（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。）

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。

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。

6、运行常量池

这儿的“静态”是指“位于固定位置”。程序运行期间，静态存储的数据将随

时等候调用。可用static关键字指出一个对象的特定元素是静态的。但Java对象本身永远都不会置入静态存储空间。

这个区域属于方法区。该区域存放类和接口的常量，除此之外，它还存放成员变量和成员方法的所有引用。当一个成员变量或者成员方法被引用的时候，JVM就通过运行常量池中的这些引用来查找成员变量和成员方法在内存中的的实际地址。

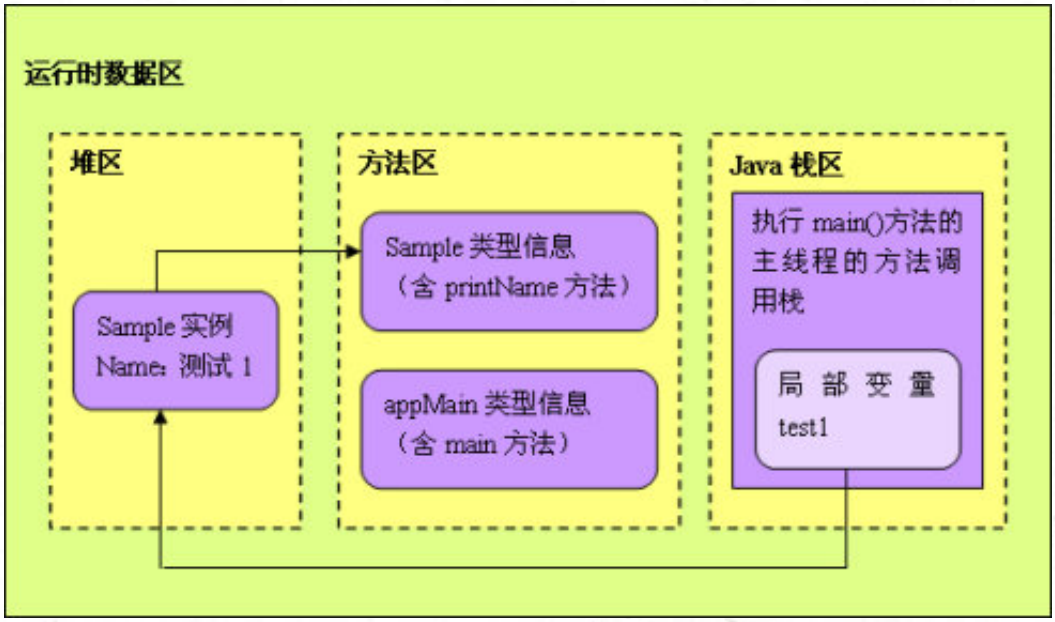
7、举例分析

例子如下：

为了更清楚地搞明白程序运行时，数据区里的情况，我们来准备2个小道具（2个非常简单的小程序）。

```
// AppMain.java
public class AppMain {           //运行时，JVM把AppMain的信息都放入方法区

    public static void main(String[] args) { //main成员方法本身放入方法区。

        //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 test1 = new  Sample( " 测试1 " );
        Sample test2 = new  Sample( " 测试2 " );
        test1.printName();
        test2.printName();
    }
}

// Sample.java
public class Sample {   //运行时，JVM把appmain的信息都放入方法区。
    private  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);
    }
}
```

OK，让我们开始行动吧，出发指令就是：“java AppMain”，包包里带好我们的行动向导图。



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。

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");

语句很简单啦，就是让JVM创建一个Sample实例，并且呢，使引用变量test1引用这个实例。貌似小case一桩哦，就让我们来跟踪一下JVM，看看它究竟是怎么来执行这个任务的：

1、Java虚拟机一看，不就是建立一个Sample类的实例吗，简单，于是就直奔方法区（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）而去，先找到Sample类的类型信息再说。结果呢，嘿嘿，没找到@@，这会儿的方法区里还没有Sample类呢（即Sample类的类信息还没有进入方法区中）。可JVM也不是一根筋的笨蛋，于是，它发扬“自己动手，丰衣足食”的作风，立马加载了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，其实，就是有点类似于C语言里的指针啦~~，而这个地址呢，就存放了在

Sample类的实例的数据区中。

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。OK，原理讲完了，就让我们来继续我们的跟踪行动！位于“=”前的test1是一个在main()方法中定义的变量，可见，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）。

OK，到这里为止呢，JVM就完成了这个简单语句的执行任务。参考我们的行动向导图，我们终于初步摸清了JVM的一点点底细了，COOL！

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。