01 结合字节码指令理解Java虚拟机栈和栈帧

> 官网：https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.6
>
> 栈帧：每个栈帧对应一个被调用的方法，可以理解为一个方法的运行空间。

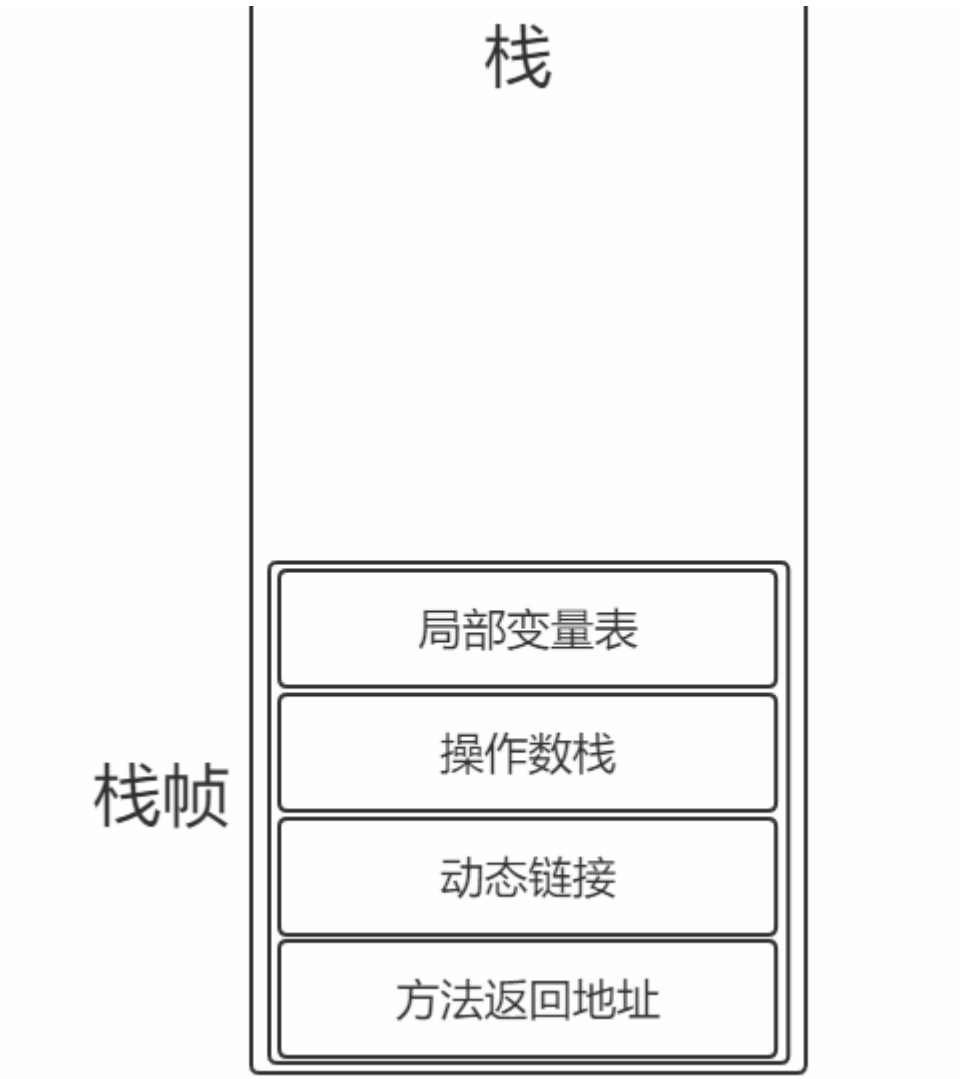
每个栈帧中包括局部变量表(Local Variables)、操作数栈(Operand Stack)、指向运行时常量池的引用(A reference to the run-time constant pool)、方法返回地址(Return Address)和附加信息。

局部变量表:方法中定义的局部变量以及方法的参数存放在这张表中
局部变量表中的变量不可直接使用，如需要使用的话，必须通过相关指令将其加载至操作数栈中作为操作数使用。

操作数栈:以压栈和出栈的方式存储操作数的

动态链接:每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接(Dynamic Linking)。

方法返回地址:当一个方法开始执行后,只有两种方式可以退出，一种是遇到方法返回的字节码指令；一种是遇见异常，并且这个异常没有在方法体内得到处理。

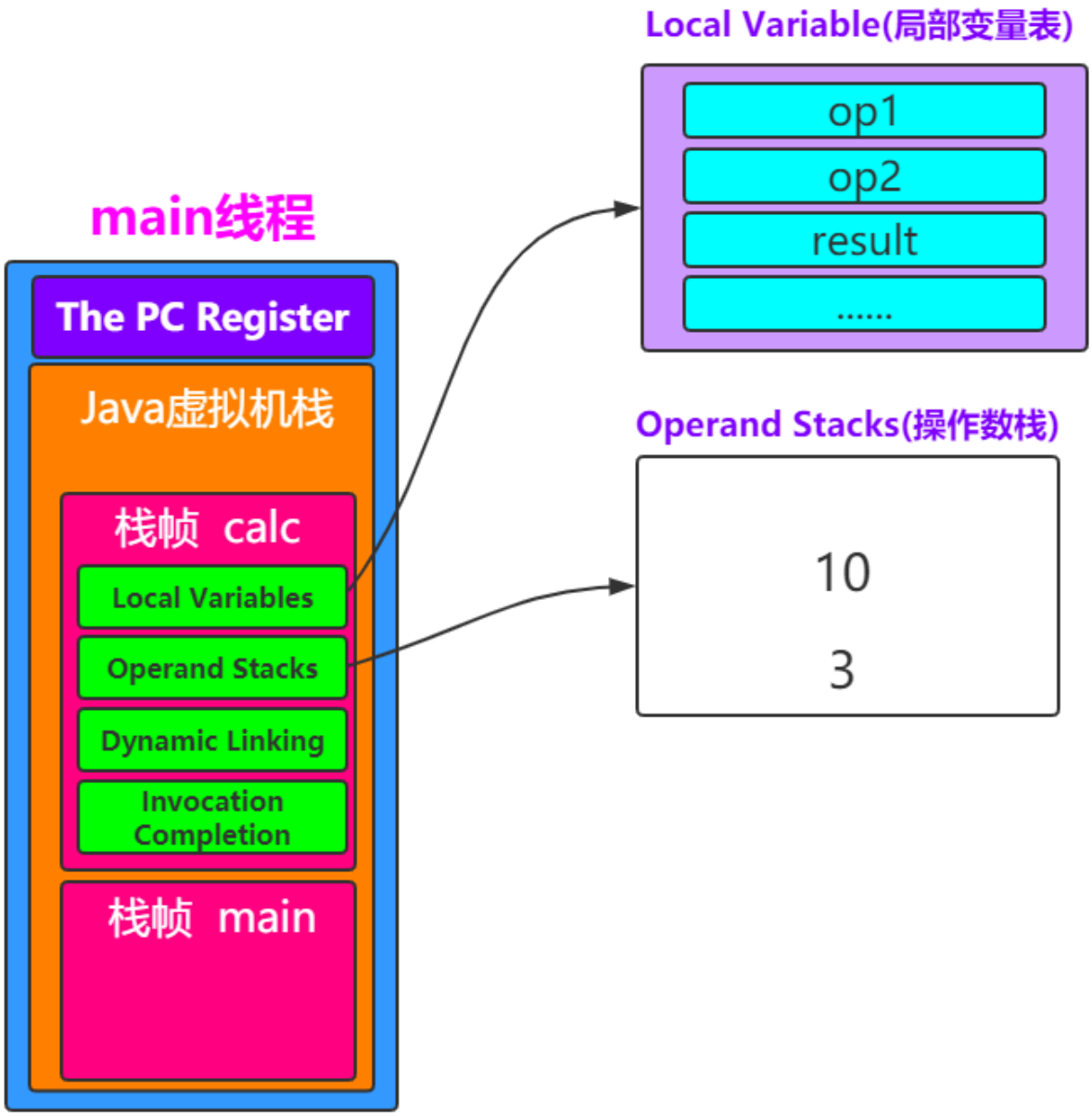



```
class Person{
    private String name="Jack";
    private int age;
    private final double salary=100;
    private static String address;
    private final static String hobby="Programming";
    public void say(){
        System.out.println("person say...");
    }
    public static int calc(int op1,int op2){
        op1=3;
        int result=op1+op2;
        return result;
    }
    public static void order(){

    }
    public static void main(String[] args){
        calc(1,2);
        order();
    }
}
```

> 此时你需要一个能够看懂反编译指令的宝典
>
> 比如官网的：https://docs.oracle.com/javase/specs/jvms/se8/html/index.html
>
> 当然网上也有很多，大家可以自己查找

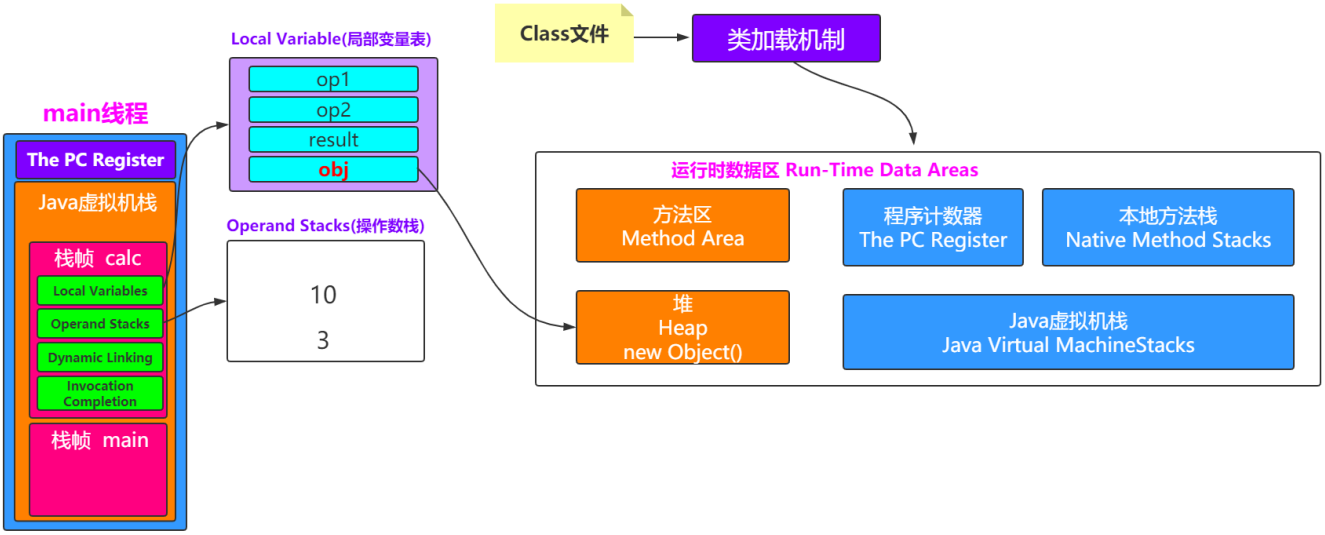
```
Compiled from "Person.java"
class Person {
  ...
  public static int calc(int, int);
    Code:
       0: iconst_3      //将int类型常量3压入[操作数栈]
       1: istore_0      //将int类型值存入[局部变量0]
       2: iload_0       //从[局部变量0]中装载int类型值入栈
       3: iload_1       //从[局部变量1]中装载int类型值入栈
       4: iadd          //将栈顶元素弹出栈，执行int类型的加法，结果入栈
       【For example, the iadd instruction (§iadd) adds two int values together. It
requires that the int values to be added be the top two values of the operand stack, pushed
there by previous instructions. Both of the int values are popped from the operand stack.
They are added, and their sum is pushed back onto the operand stack. Subcomputations may be
nested on the operand stack, resulting in values that can be used by the encompassing
computation.】
       5: istore_2      //将栈顶int类型值保存到[局部变量2]中
       6: iload_2       //从[局部变量2]中装载int类型值入栈
       7: ireturn       //从方法中返回int类型的数据
  ...
}
```



02 折腾一下

2.1栈指向堆

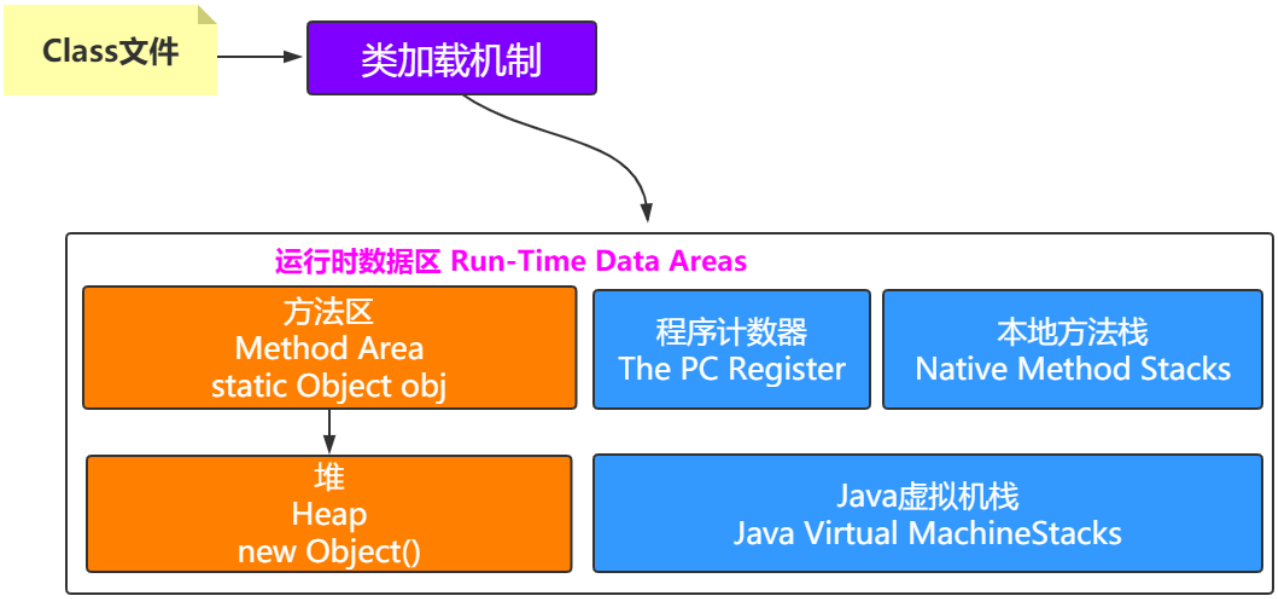
如果在栈帧中有一个变量，类型为引用类型，比如Object obj=new Object()，这时候就是典型的栈中元素指向堆中的对象。



2.2 方法区指向堆

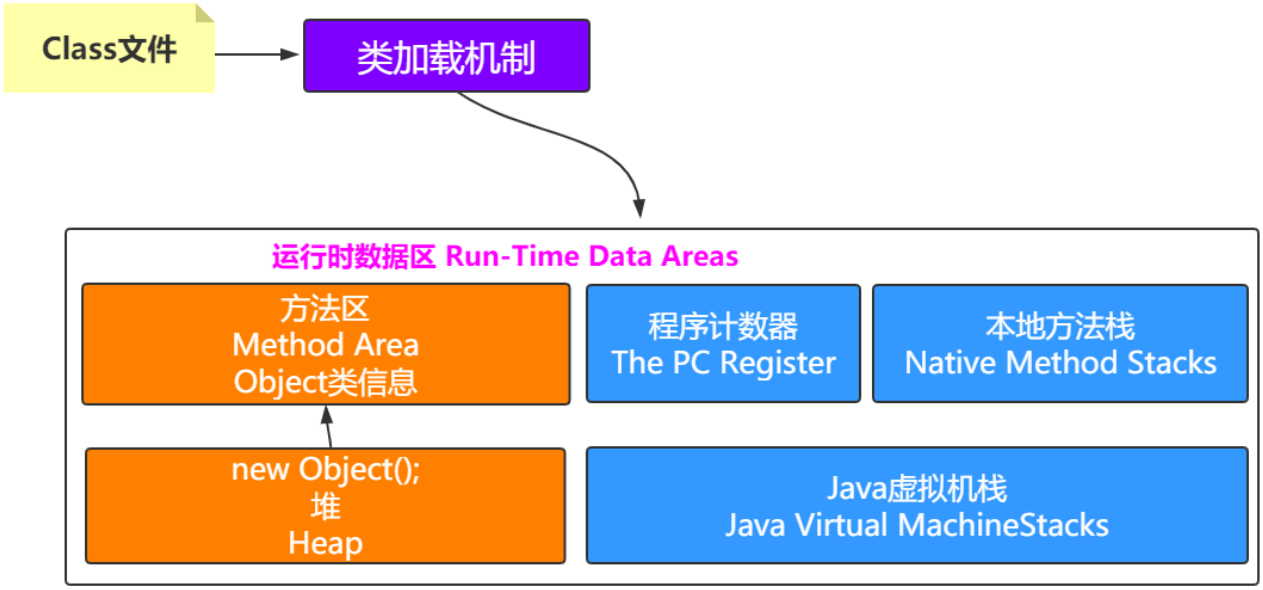
方法区中会存放静态变量，常量等数据。如果是下面这种情况，就是典型的方法区中元素指向堆中的对象。

```
private static Object obj=new Object();
```



2.3 堆指向方法区

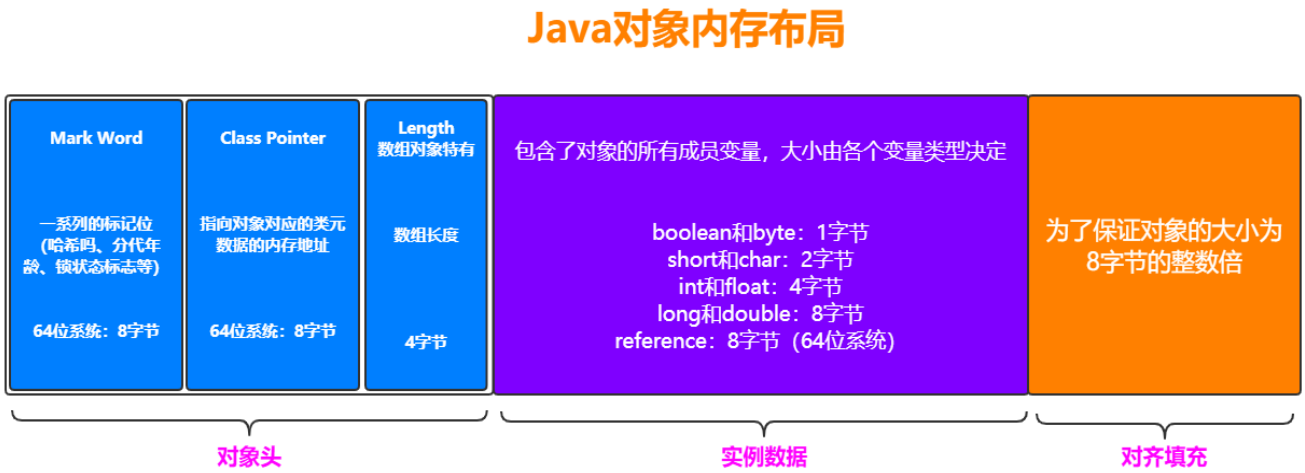
方法区中会包含类的信息，堆中会有对象，那怎么知道对象是哪个类创建的呢？



思考：一个对象怎么知道它是由哪个类创建出来的？怎么记录？这就需要了解一个Java对象的具体信息咯。

2.4 Java对象内存布局

> 一个Java对象在内存中包括3个部分：对象头、实例数据和对齐填充

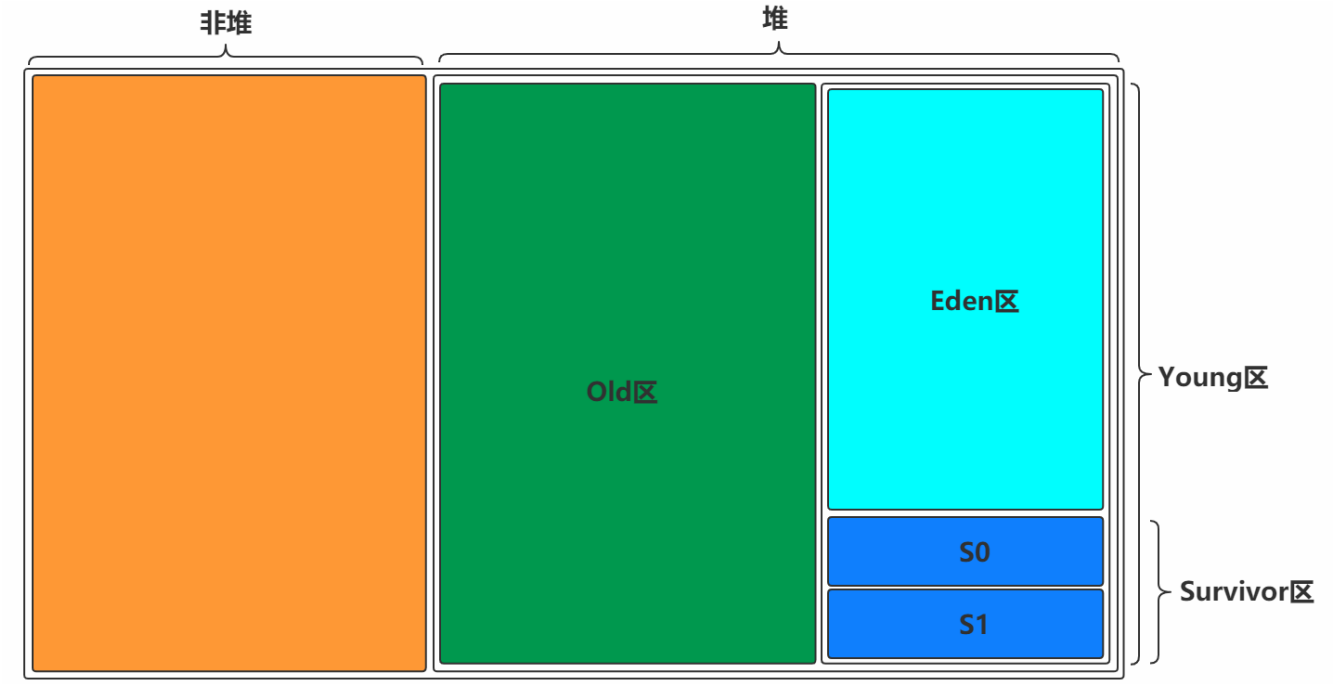


03 内存模型

3.1 图解

```
一块是非堆区，一块是堆区。
堆区分为两大块，一个是Old区，一个是Young区。
Young区分为两大块，一个是Survivor区（S0+S1），一块是Eden区。 Eden:S0:S1=8:1:1
S0和S1一样大，也可以叫From和To。
```

画个图理解一下



> 根据之前对于Heap的介绍可以知道，一般对象和数组的创建会在堆中分配内存空间，关键是堆中有这么多区域，那一个对象的创建到底在哪个区域呢?

3.2 对象创建所在区域

一般情况下，新创建的对象都会被分配到Eden区，一些特殊的大的对象会直接分配到Old区。

> 比如有对象A，B，C等创建在Eden区，但是Eden区的内存空间肯定有限，比如有100M，假如已经使用了100M或者达到一个设定的临界值，这时候就需要对Eden内存空间进行清理，即垃圾收集(Garbage Collect)，这样的GC我们称之为Minor GC，Minor GC指得是Young区的GC。
>
> 经过GC之后，有些对象就会被清理掉，有些对象可能还存活着，对于存活着的对象需要将其复制到Survivor区，然后再清空Eden区中的这些对象。

3.3 Survivor区详解

由图解可以看出，Survivor区分为两块S0和S1，也可以叫做From和To。

在同一个时间点上，S0和S1只能有一个区有数据，另外一个是空的。

> 接着上面的GC来说，比如一开始只有Eden区和From中有对象，To中是空的。
>
> 此时进行一次GC操作，From区中对象的年龄就会+1，我们知道Eden区中所有存活的对象会被复制到To区，From区中还能存活的对象会有两个去处。
>
> 若对象年龄达到之前设置好的年龄阈值，此时对象会被移动到Old区，没有达到阈值的对象会被复制到To区。
>
> 此时Eden区和From区已经被清空(被GC的对象肯定没了，没有被GC的对象都有了各自的去处)。
>
> 这时候From和To交换角色，之前的From变成了To，之前的To变成了From。
>
> 也就是说无论如何都要保证名为To的Survivor区域是空的。
>
> Minor GC会一直重复这样的过程，知道To区被填满，然后会将所有对象复制到老年代中。

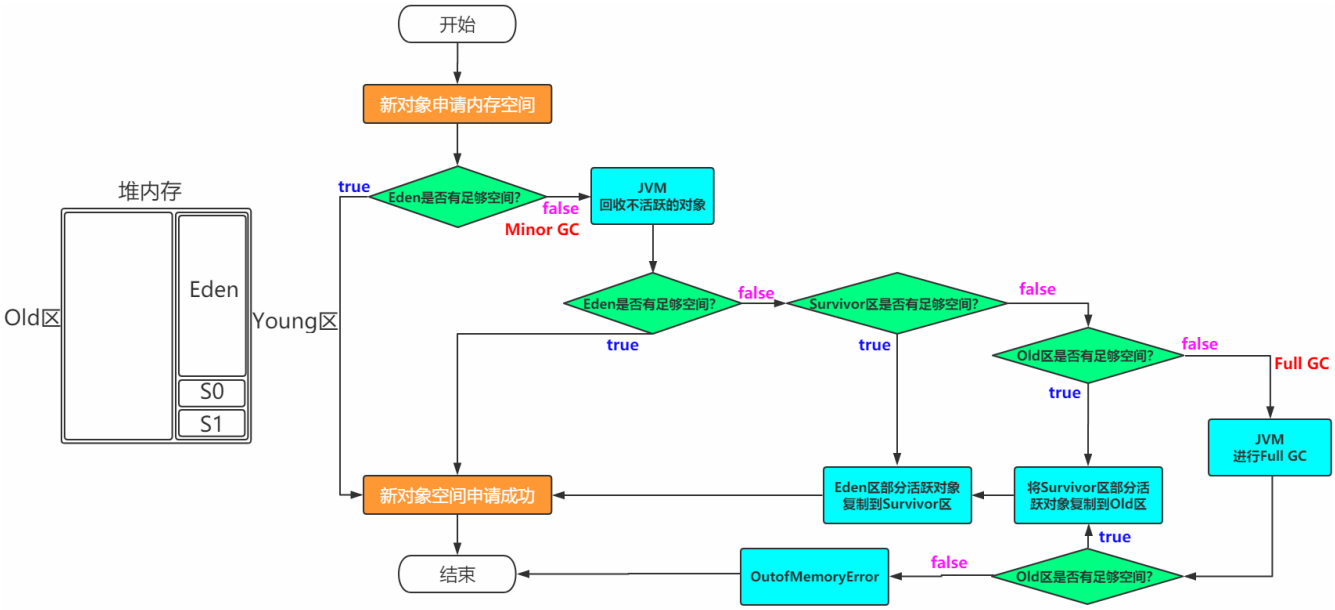
3.4 Old区详解

从上面的分析可以看出，一般Old区都是年龄比较大的对象，或者相对超过了某个阈值的对象。

在Old区也会有GC的操作，Old区的GC我们称作为Major GC，每次GC之后还能存活的对象年龄也会+1，如果年龄超过了某个阈值，就会被回收。

3.5 对象的一辈子理解

> 我是一个普通的Java对象，我出生在Eden区，在Eden区我还看到和我长的很像的小兄弟，我们在Eden区中玩了挺长时间。有一天Eden区中的人实在是太多了，我就被迫去了Survivor区的“From”区，自从去了Survivor区，我就开始漂了，有时候在Survivor的“From”区，有时候在Survivor的“To”区，居无定所。直到我18岁的时候，爸爸说我成人了，该去社会上闯闯了。
> 于是我就去了年老代那边，年老代里，人很多，并且年龄都挺大的，我在这里也认识了很多人。在年老代里，我生活了20年(每次GC加一岁)，然后被回收。



3.6 常见问题

- 如何理解Minor/Major/Full GC

```
Minor GC:新生代
Major GC:老年代
Full GC:新生代+老年代
```

- 为什么需要Survivor区?只有Eden不行吗?

如果没有Survivor,Eden区每进行一次Minor GC,存活的对象就会被送到老年代。
这样一来，老年代很快被填满,触发Major GC(因为Major GC一般伴随着Minor GC,也可以看做触发了Full GC)。
老年代的内存空间远大于新生代,进行一次Full GC消耗的时间比Minor GC长得多。
执行时间长有什么坏处?频发的Full GC消耗的时间很长,会影响大型程序的执行和响应速度。

可能你会说，那就对老年代的空间进行增加或者较少咯。
假如增加老年代空间，更多存活对象才能填满老年代。虽然降低Full GC频率，但是随着老年代空间加大,一旦发生Full GC,执行所需要的时间更长。
假如减少老年代空间，虽然Full GC所需时间减少，但是老年代很快被存活对象填满,Full GC频率增加。

所以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。

- 为什么需要两个Survivor区?

最大的好处就是解决了碎片化。也就是说为什么一个Survivor区不行?第一部分中,我们知道了必须设置Survivor区。假设现在只有一个Survivor区,我们来模拟一下流程:
刚刚新建的对象在Eden中,一旦Eden满了,触发一次Minor GC,Eden中的存活对象就会被移动到Survivor区。这样继续循环下去,下一次Eden满了的时候,问题来了,此时进行Minor GC,Eden和Survivor各有一些存活对象,如果此时把Eden区的存活对象硬放到Survivor区,很明显这两部分对象所占有的内存是不连续的,也就导致了内存碎片化。
永远有一个Survivor space是空的,另一个非空的Survivor space无碎片。

- 新生代中Eden:S1:S2为什么是8:1:1?

新生代中的可用内存：复制算法用来担保的内存为9：1
可用内存中Eden：S1区为8：1
即新生代中Eden:S1:S2 = 8：1：1

04 体验与验证

4.1 使用jvisualvm查看

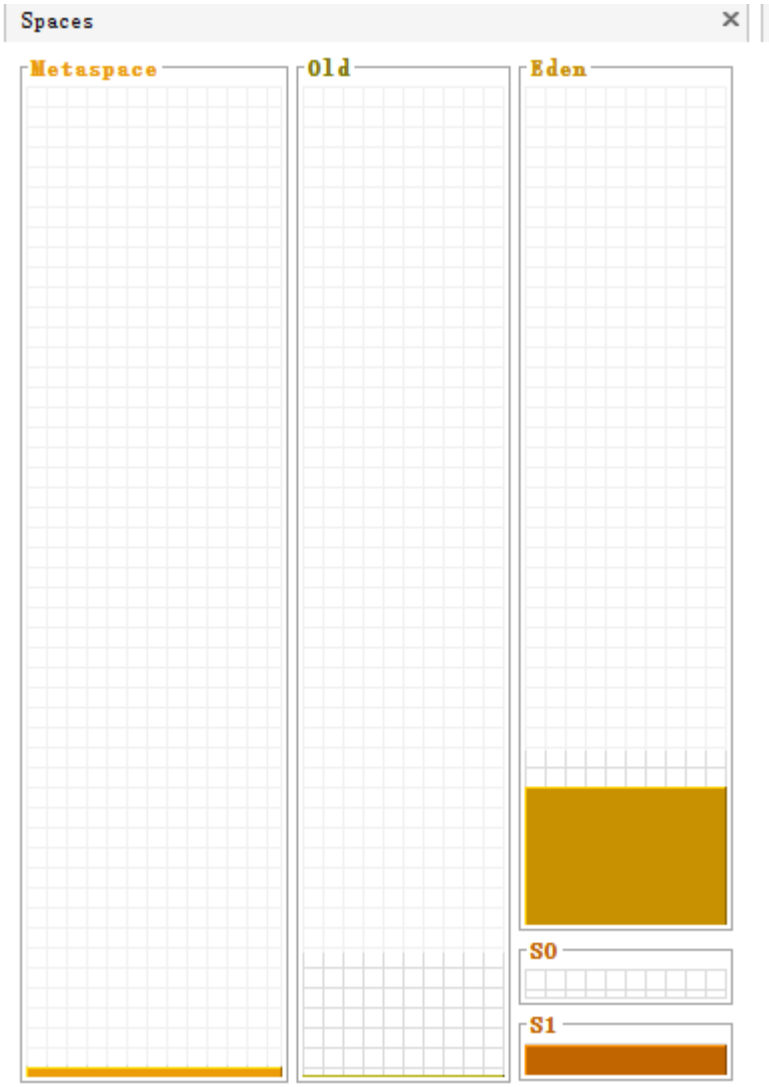
`visualgc插件下载链接`：

https://visualvm.github.io/pluginscenters.html --->选择对应版本链接--->**Tools**--->Visual GC

若上述链接找不到合适的，大家也可以自己在网上下载对应的版本

当然，大家如果是jdk1.8的版本，也可以直接用我放到网盘中的：

网盘/课程源码/com-sun-tools-visualvm-modules-visualgc.nbm



4.1 堆内存溢出

4.1.1 代码

```
@RestController
public class HeapController {
    List<Person> list=new ArrayList<Person>();
    @GetMapping("/heap")
    public String heap() throws Exception{
        while(true){
            list.add(new Person());
            Thread.sleep(1);
        }
    }
}
```

> 记得设置参数比如-Xmx20M -Xms20M

4.1.2 运行结果

访问->http://localhost:8080/heap

```
Exception in thread "http-nio-8080-exec-2" java.lang.OutOfMemoryError: GC overhead limit exceeded
```

4.2 方法区内存溢出

> 比如向方法区中添加Class的信息

4.2.1 asm依赖和Class代码

```
<dependency>
    <groupId>asm</groupId>
    <artifactId>asm</artifactId>
    <version>3.3.1</version>
</dependency>
```

```
public class MyMetaspace extends ClassLoader {
    public static List<Class<?>> createClasses() {
        List<Class<?>> classes = new ArrayList<Class<?>>();
        for (int i = 0; i < 10000000; ++i) {
            ClassWriter cw = new ClassWriter(0);
            cw.visit(Opcodes.V1_1, Opcodes.ACC_PUBLIC, "Class" + i, null,
                    "java/lang/Object", null);
            MethodVisitor mw = cw.visitMethod(Opcodes.ACC_PUBLIC, "<init>",
                    "()V", null, null);
            mw.visitVarInsn(Opcodes.ALOAD, 0);
            mw.visitMethodInsn(Opcodes.INVOKESPECIAL, "java/lang/Object",
                    "<init>", "()V");
            mw.visitInsn(Opcodes.RETURN);
            mw.visitMaxs(1, 1);
            mw.visitEnd();
            Metaspace test = new Metaspace();
            byte[] code = cw.toByteArray();
            Class<?> exampleClass = test.defineClass("Class" + i, code, 0, code.length);
            classes.add(exampleClass);
        }
        return classes;
    }
}
```

4.2.2 代码

```
@RestController
public class NonHeapController {
    List<Class<?>> list=new ArrayList<Class<?>>();

    @GetMapping("/nonheap")
    public String nonheap() throws Exception{
        while(true){
            list.addAll(MyMetaspace.createClasses());
            Thread.sleep(5);
        }
    }
}
```

> 设置Metaspace的大小，比如-XX:MetaspaceSize=50M -XX:MaxMetaspaceSize=50M

4.2.3 运行结果

访问->http://localhost:8080/nonheap

```
java.lang.OutOfMemoryError: Metaspace
    at java.lang.ClassLoader.defineClass1(Native Method) ~[na:1.8.0_191]
    at java.lang.ClassLoader.defineClass(ClassLoader.java:763) ~[na:1.8.0_191]
```

4.3 虚拟机栈

4.3.1 代码演示StackOverFlow

```
public class StackDemo {
    public static long count=0;
    public static void method(long i){
        System.out.println(count++);
        method(i);
    }
    public static void main(String[] args) {
        method(1);
    }
}
```

4.3.2 运行结果

```
7252
7253
7254
7255
Exception in thread "main" java.lang.StackOverflowError
    at sun.nio.cs.UTF_8$Encoder.encodeLoop(UTF_8.java:691)
    at java.nio.charset.CharsetEncoder.encode(CharsetEncoder.java:579)
```

4.3.3 理解和说明

Stack Space用来做方法的递归调用时压入Stack Frame(栈帧)。所以当递归调用太深的时候，就有可能耗尽Stack Space，爆出StackOverflow的错误。

-Xss128k：设置每个线程的堆栈大小。JDK 5以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。根据应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。

线程栈的大小是个双刃剑，如果设置过小，可能会出现栈溢出，特别是在该线程内有递归、大的循环时出现溢出的可能性更大，如果该值设置过大，就有影响到创建栈的数量，如果是多线程的应用，就会出现内存溢出的错误。