2.4 JVM内存模型

2.4.1 与运行时数据区

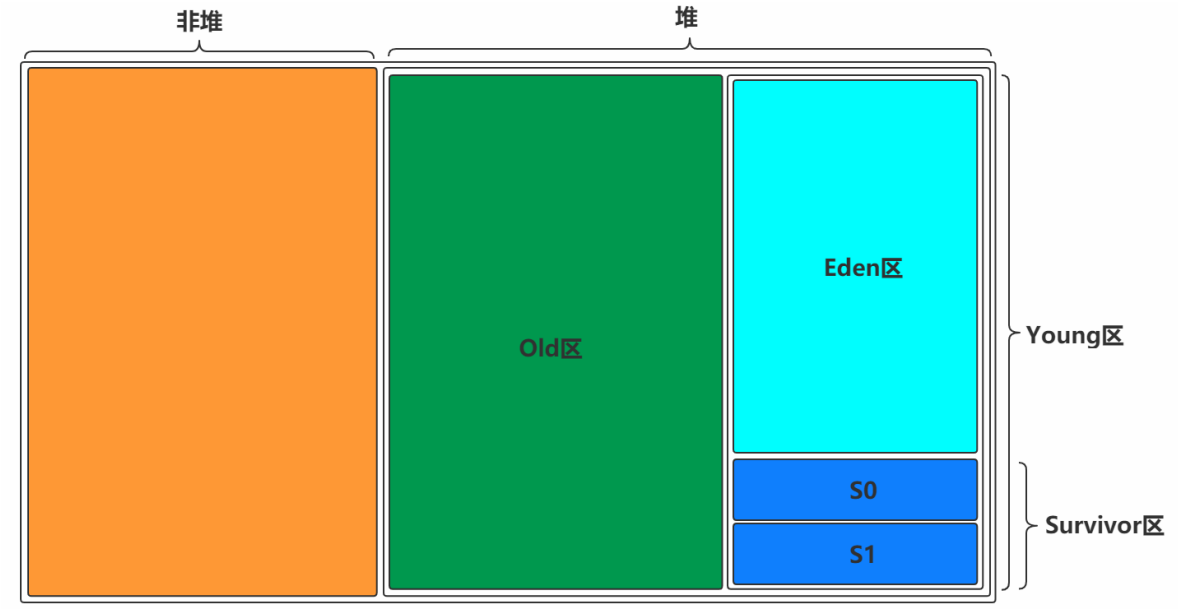
上面对运行时数据区描述了很多，其实重点存储数据的是堆和方法区(非堆)，所以内存的设计也着重从这两方面展开(注意这两块区域都是线程共享的)。

对于虚拟机栈，本地方法栈，程序计数器都是线程私有的。

可以这样理解，JVM运行时数据区是一种规范，而JVM内存模式是对该规范的实现

2.4.2 图形展示

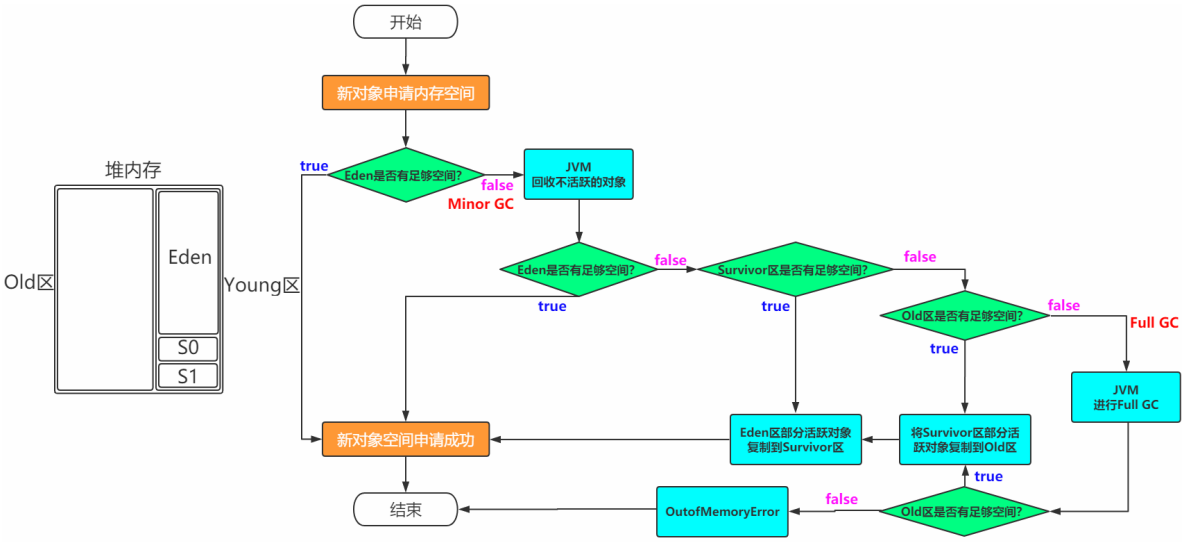
```
一块是非堆区，一块是堆区
堆区分为两大块，一个是Old区，一个是Young区
Young区分为两大块，一个是Survivor区（S0+S1），一块是Eden区
S0和S1一样大，也可以叫From和To
```



2.4.3 对象创建过程

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。

```
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。
```



2.4.4 常见问题

- 如何理解各种GC

```
Partial GC
Partial其实也就是部分的意思.那么翻译过来也就是回收部分GC堆的模式,他并不会回收我们整个堆.而我们的young GC以及我们的Old GC都属于这种模式
young GC
只回收young区
old GC
只回收Old区
full GC
实际上就是对于整体回收
```

- 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
现代的商业虚拟机都采用这种收集算法来回收新生代，IBM公司的专门研究表明，新生代中的对象大概98%是“朝生夕死”的
```

- 堆内存中都是线程共享的区域吗?

```
JVM默认为每个线程在Eden上开辟一个buffer区域，用来加速对象的分配，称之为TLAB，全称:Thread Local Allocation Buffer。
对象优先会在TLAB上分配，但是TLAB空间通常会比较小，如果对象比较大，那么还是在共享区域分配。
```

2.4.5 体验与验证

2.4.5.1 使用visualvm

visualgc插件下载链接： https://visualvm.github.io/pluginscenters.html

选择对应JDK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

2.4.5.2 堆内存溢出

- 代码

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

> 记得设置参数比如-Xmx20M -Xms20M

- 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
```

2.4.5.3 方法区内存溢出

> 比如向方法区中添加Class的信息

- 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;
    }
}
```

- 代码

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

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

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

- 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]
```

2.4.5.4 虚拟机栈

- 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);
    }
}
```

- 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)
```

- 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。