http://visualvm.github.io/pluginscenters.html visualVM插件下载地址

1 内存分配

昨天我们讲了JVM的内存结构

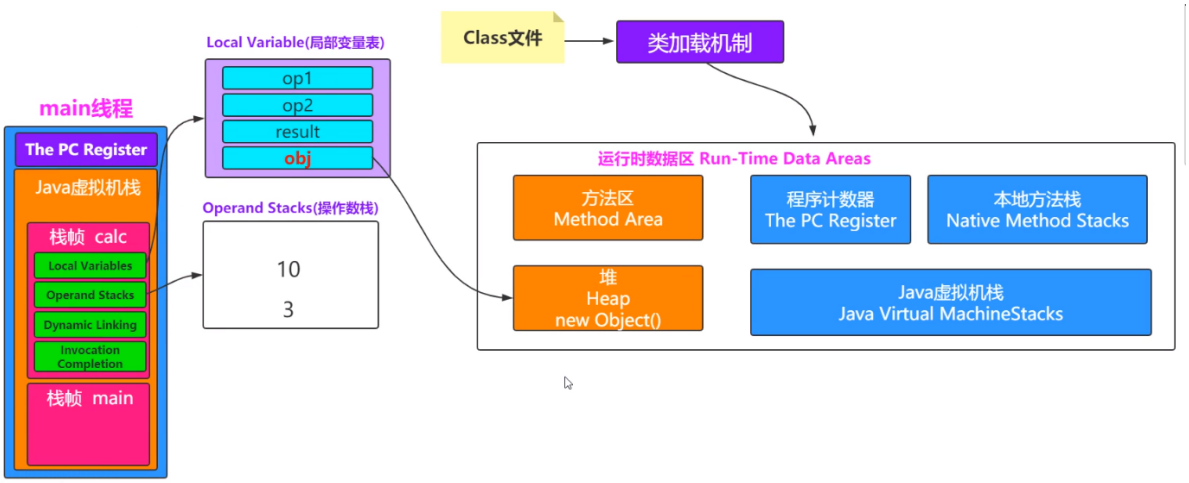
1.1程序计数器

1.2本地方法栈

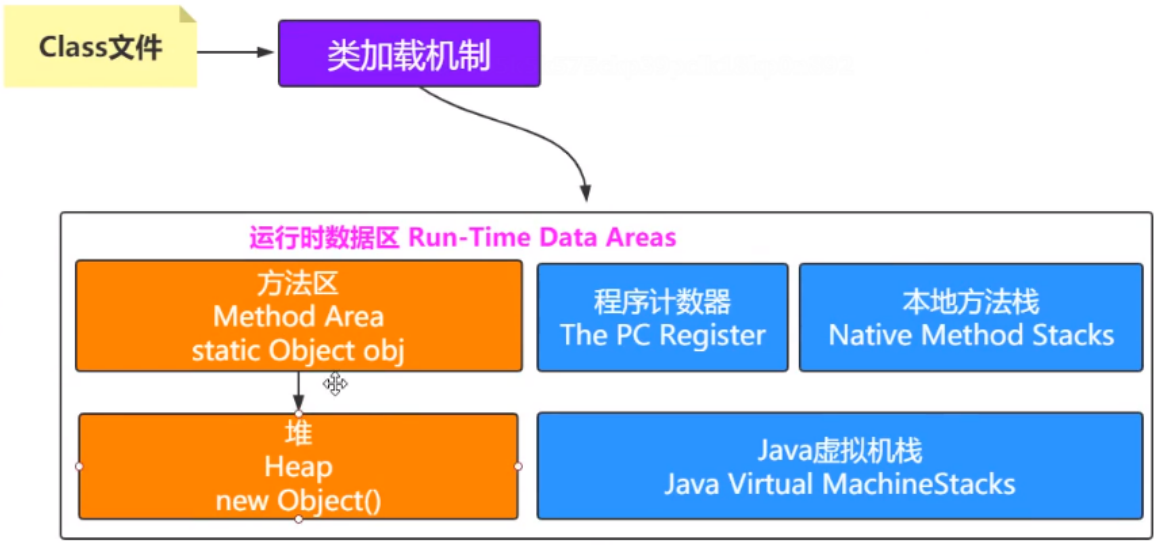
1.3虚拟机栈

栈帧（Stack Frame）是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈（Virtual Machine Stack）的栈元素。它包括局部变量表、操作数栈、动态链接、方法返回地址。

Object obj = new Object()：栈指向堆

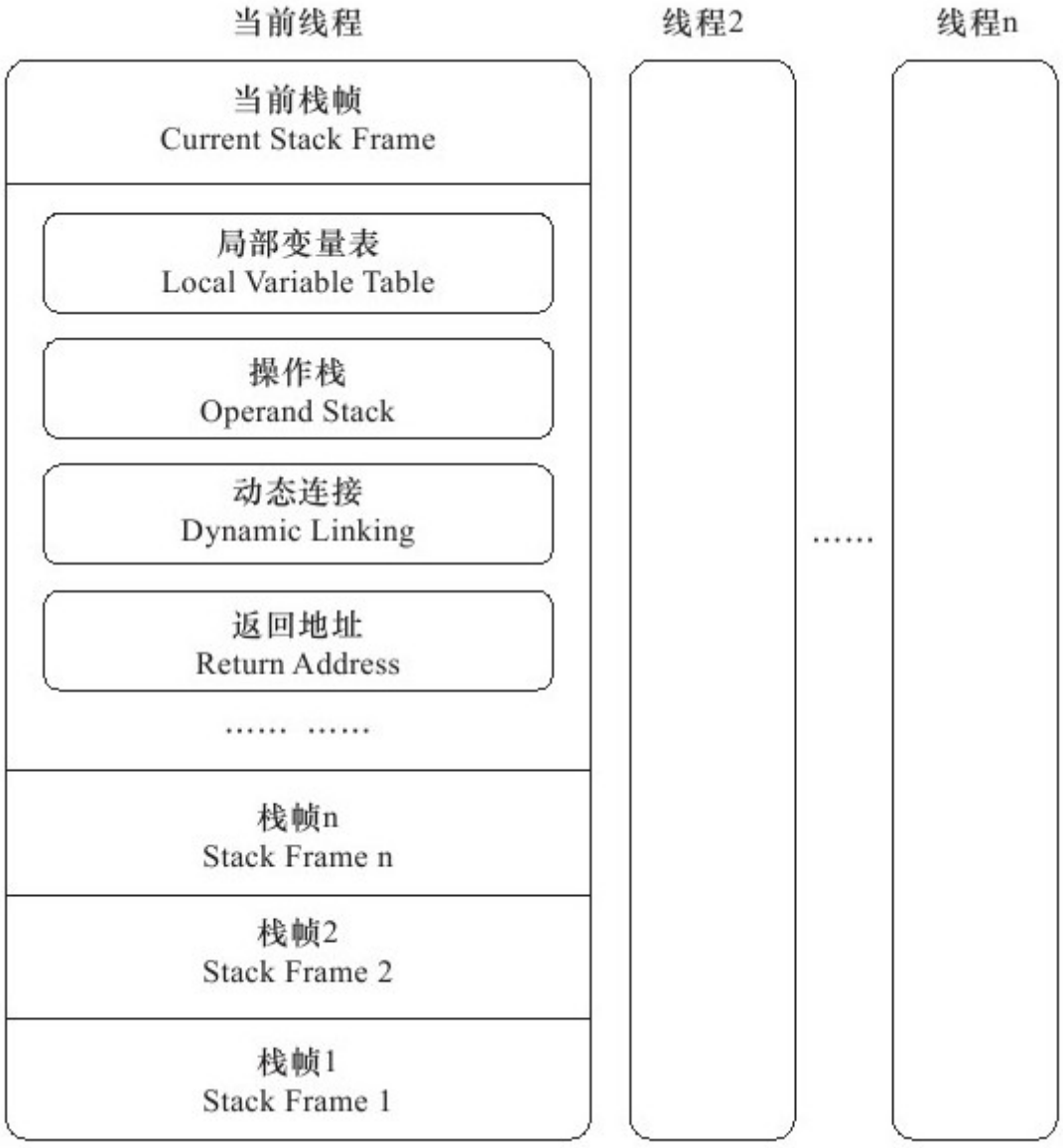


static Object obj = new Object()：方法区指向堆

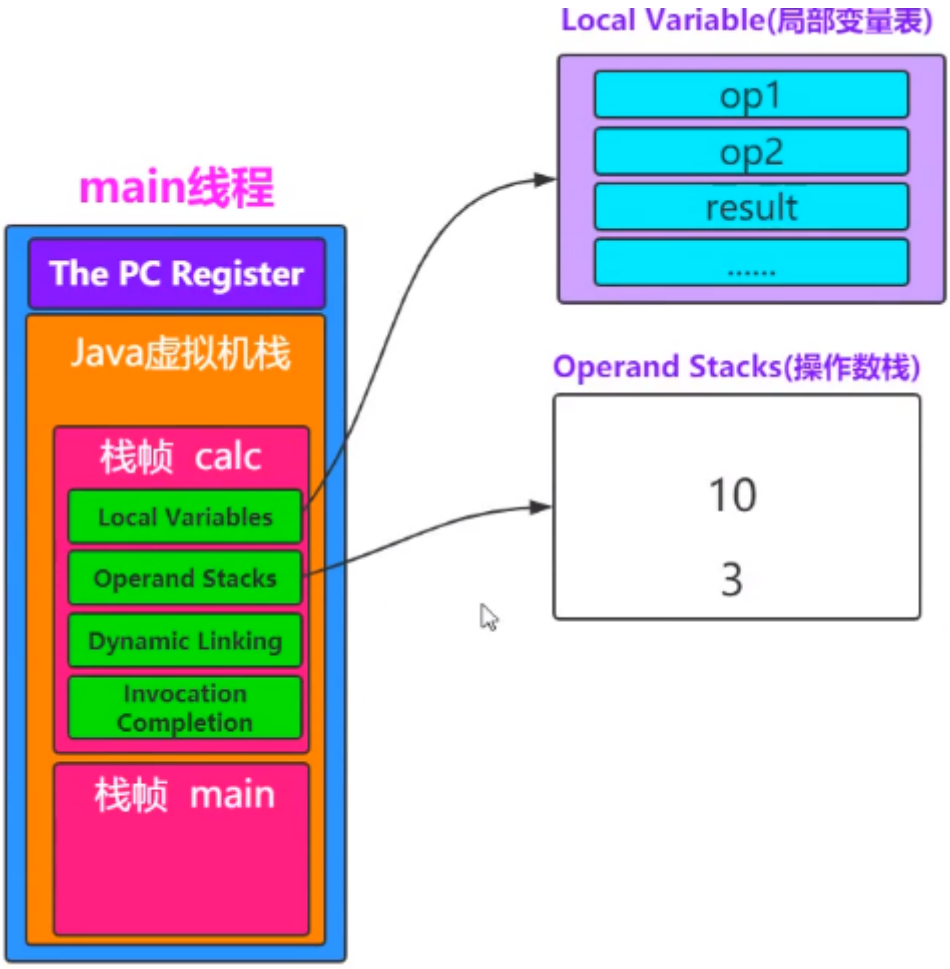


对象中存在指向类信息的类元数据，堆指向方法区

Java对象内存布局



查看反编译字节码：javap -c Person.class >Person.txt




```
public static int calc(int, int);
    Code:
       0: iconst_3 //把常数3放入操作数栈
       1: istore_0 //弹出操作数栈中的数据给局部常量表中的第一个
       2: iload_0  //将局部变量表中的第一个数据压入操作数栈
       3: iload_1  //将局部变量表中的第二个数据压入操作数栈
       4: iadd     //对操作数栈中的两个数据相加
       5: istore_2 //弹出操作数栈中的数据给局部常量表中的第三个
       6: iload_2  //将局部变量表中的第三个数据压入操作数栈
       7: ireturn  //返回
```

每一个方法从调用开始至执行完成的过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。每一个栈帧都包括了局部变量表、操作数栈、动态连接、方法返回地址和一些额外的附加信息。在编译程序代码的时候，栈帧中需要多大的局部变量表，多深的操作数栈都已经完全确定了，并且写入到方法表的Code属性之中，因此一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。

一个线程中的方法调用链可能会很长，很多方法都同时处于执行状态。对于执行引擎来说，在活动线程中，只有位于栈顶的栈帧才是有效的，称为当前栈帧（Current Stack Frame），与这个栈帧相关联的方法称为当前方法（Current Method）。执行引擎运行的所有字节码指令都只针对当前栈帧进行操作，在概念模型上，典型的栈帧结构如图所示。

1.3.1局部变量表

局部变量表（Local Variable Table）是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。在Java程序编译为Class文件时，就在方法的Code属性的max_locals数据项中确定了该方法所需要分配的局部变量表的最大容量。

1.3.2动态连接

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）。我们知道Class文件的常量池中存有大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用作为参数。这些符号引用一部分会在类加载阶段或者第一次使用的时候就转化为直接引用，这种转化称为静态解析。另外一部分将在每一次运行期间转化为直接引用，这部分称为动态连接。（静态分派，动态分派）（多态举例）

1.3.3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。

前面的3个都是根据线程的生命周期自动生或灭的！所以我们不需要管，所以只有方法区跟堆才是我们需要考虑GC的点！

~~为什么Jdk1.8要把方法区从JVM里（永久代）移到直接内存（元空间）？~~

1. ~~字符串存在永久代中，容易出现性能问题和内存溢出。~~
2. ~~类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难。~~
3. ~~永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。~~

1.4堆

1.4.1查看方法区跟堆大小

设置大小

-Xms20M -Xmx20M

打印分配的大小

-XX:+PrintGCDetails

```
long initialMemory = Runtime.getRuntime().totalMemory()/1024/1024;
long maxMemory = Runtime.getRuntime().maxMemory()/1024/1024;
```

默认堆大小 ： 物理电脑内存大小的 1/64 最大的内存大小：物理电脑内存大小的 1/4

1.4.2新生代跟老年代

那么我们堆里面的内存到底是怎么样的呢？我们一个一个看

首先，我们想到的是堆就预分配了一块内存！！所有的对象都在一块，当我们要进行GC的时候，会去找所有的对象是否需要回收！！

但是我们的对象大部分都是朝生夕死的，那么我们就想着，能不能把大部分新创建的对象单独放到某一个地方，减少我们GC的范围，那些少部分的非朝生夕死的对象，或者很大的对象放到我们的另外一个地方！！这样我们就有了2个区域，old区跟young区！！比例old占三分之二，young占三分之一

年龄：一次垃圾回收，没被回收的对象年龄加1,年龄大于15了就移到老年代。

老年代：对象大小特别大。

这样我们就能保证一般情况下只会对我们的young进行GC，那么这个GC的过程叫做young GC或者MinorGC

什么时候进老年代

既然区分了2个代，那么老年代肯定也是要放数据的，不然就没用了，那什么时候进入老年代！！新创建的对象在新生代，那么进入老年代肯定是需要条件的！

1.大对象 这个也是可以设置的，目的是为了减少频繁的youngGC，大对象比较占内存

2.年限达到15岁，默认15，可以设定，但是不能超过15，**-XX:MaxTenuringThreshold=16**就会报错，无法创建JVM，这个是因为java的对象头是4bit来保存分代年龄

肯定，老年代也是需要去进行内存清理的，那么老年代的GC叫做Major GC，Major GC一般会伴随至少一次MinorGC

那么full GC就是我的整个堆、元空间等全局范围的GC

1.4.3Survivor 区域

我们现在分为老年代跟新生代，比我们之前在一起的时候，GC速率已经快不少了，但是我们又发现了一个问题！！空间碎片，我回收完了之后，有太多碎片，保存不了超过碎片的数据，又会浪费！

所以，又想了个办法，把young区再分Eden区、From、To三个区域！那么他们是怎么解决碎片问题的呢

1.我们数据只会存在Eden区、from或者To 2个区域的一个区域，也就是我From.跟To肯定有一个是空的！

2.最开始，我们的数据假如都在Eden或From区，当Eden满了或者没有足够的连续空间保存我的对象时，会触发MinorGC，所有对象的年龄+1，这个时候会把我们Eden区的对象赋值到To区，之前From区的数据就会根据他们的年龄来决定去向，到了去老年代的阀值就去老年代，不然就跟着一起去To区！

3.这样，我们的Eden区跟From区又空了，就不会有碎片，如此循环！解决碎片问题，就是牺牲我一个小区的空间解决空间碎片问题

Eden：from：to=8:1:1

1.4.3.1 对象动态年龄判断

当前放对象的Survivor区域里（其中一块区域，放对象的那块S区），一批对象的总大小大于这块Survivor区域内存大小的50%（-XX:TargetSurvivorRatio可以指定），那么此时大于等于这批对象年龄最大值的对象，就可以直接进入老年代了，例如Survivor区域里现在有一批对象，年龄1+年龄2+年龄n的多个年龄对象总和超过了Survivor区域的50%，此时就会把年龄n（含）以上的对象都放入老年代。这个规则其实是希望那些可能是长期存活的对象，尽早进入老年代。**对象动态年龄判断机制一般是在minor GC之后触发。**

1.4.3.2 老年代空间分配担保机制

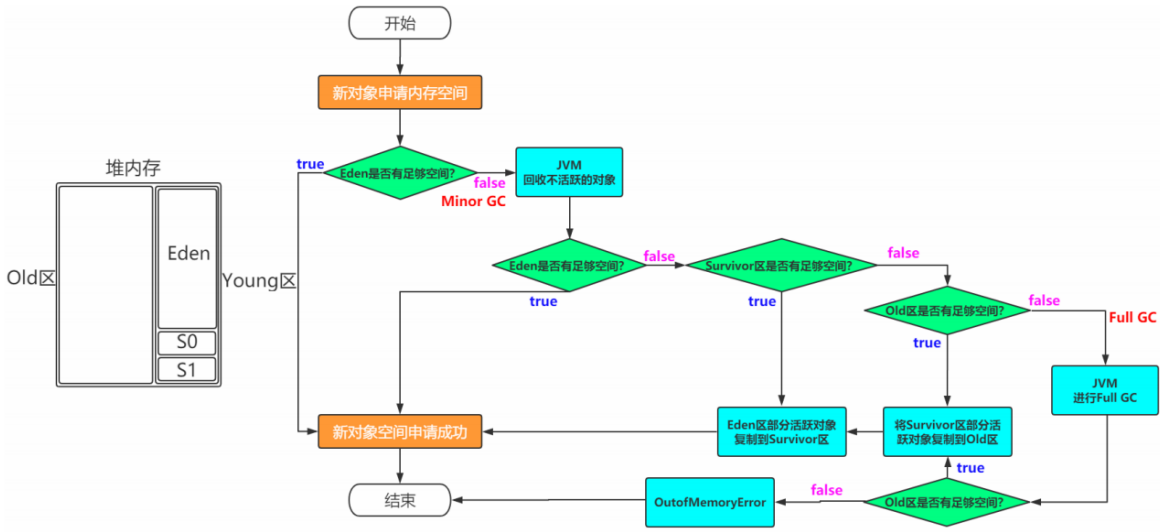
年轻代每次minor GC之前JVM都会计算下老年代**剩余可用空间**

1. 如果这个可用空间小于年轻代里现有的所有对象大小之后（**包括垃圾对象**），就会看一个“-XX:-HandlerPromotionFailure"(jdk1.8默认是设置了）的参数是否设置了。
2. 如何有这个参数，就会看看老年代的可用内存大小，是否大于之前每一次minor GC后进入老年代的对象的平均大小。
3. 如果上一步结果是小于或者之前说的参数没有设置，那么就会触发一次Full GC，对老年代和年轻代一起回收一次垃圾。
4. 如果回收完还是没有足够空间存放新的对象就会发生“OOM”
5. 当然，如果minorGC之后剩余存活的需要挪动到老年代的对象大小还是大于老年代可用空间，那么也会触发FullGC，FullGC完之后如果还是没有空间放minor GC之后的存活对象，则也会发生“OOM”。

1.4.4对象一辈子理解

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。



1.4.5常见问题

- 如何理解Minor/Major/Full GC

Minor GC:新生代
Major GC:老年代 ，MajorGC通常会伴随着MinorGC，也就意味着会触发FullGC
Full GC:新生代+老年代
FullGC：STW（Stop The World） ，会消耗时间

- 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,并且没有年龄限制的话，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:S0:S1为什么是8:1:1?

首先S0和S1必须一样大，不一样大的话，复制的时候会出问题
第二个为啥Eden区要占大部分，因为我们对象大部分都是“朝生夕死”，对象的生命周期比较短，然后直接可以在Eden去创建，如果Eden区小的话，会经常触发minorGC

1.5方法区：永久代

1.6 内存区域的一个问题

```
public class test{
    public int i = 10;//i作为成员变量在类加载的时候会存在方法区存字段的描述信息，10作为字面量会保存在方法区的常量池中。当new Test对象只要，i和10会随着对象一起保存在堆中。
    public static int j = 11;//j作为静态变量，会保存在方法区中，11作为字面量会保存在常量池中

    public String s1 = "123";//s1作为成员变量在类加载的时候会存在方法区存字段的描述信息，“123”作为字面量会保存在方法区的常量池中。当new Test对象时，s1以及“123”的真实地址会随着对象一起保存在堆中。

    public void add(){
        int k = 12;//k作为基本数据类型，会被存在虚拟机栈中，12作为字面量会被存在常量池中。
        String s2 = "321";//s2作为引用类型，会被存在虚拟机栈中，“321”作为字面量会被存在常量池中。
    }

}
```

2 体验与验证

2.1堆内存溢出

2.1.1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

2.1.2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
```

2.2方法区内存溢出

比如向方法区中添加Class的信息

2.2.1asm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;
    }
}
```

2.2.2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
```

2.2.3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]
```

2.3虚拟机栈内存溢出

2.3.1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);
    }
}
```

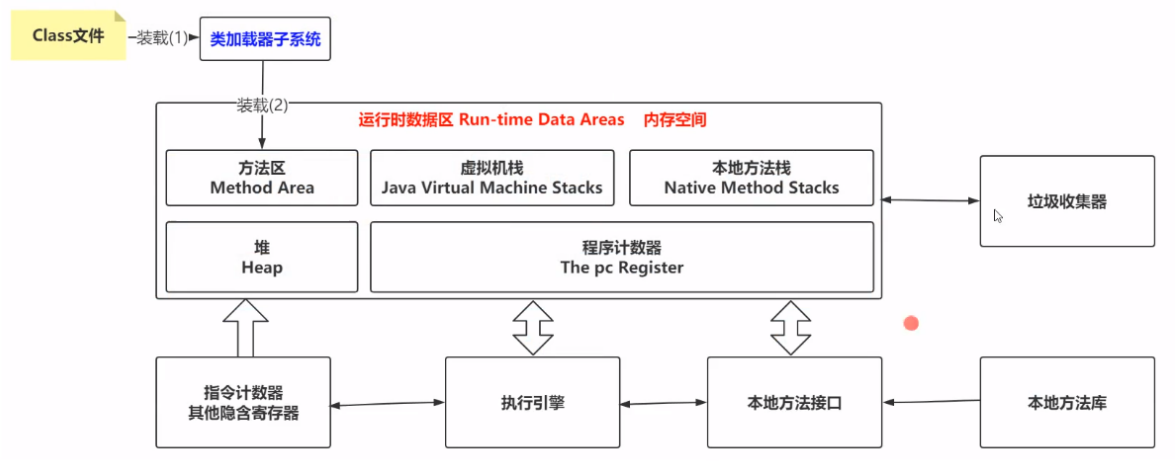
2.3.2运行结果

```
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)
```

2.3.3理解和说明

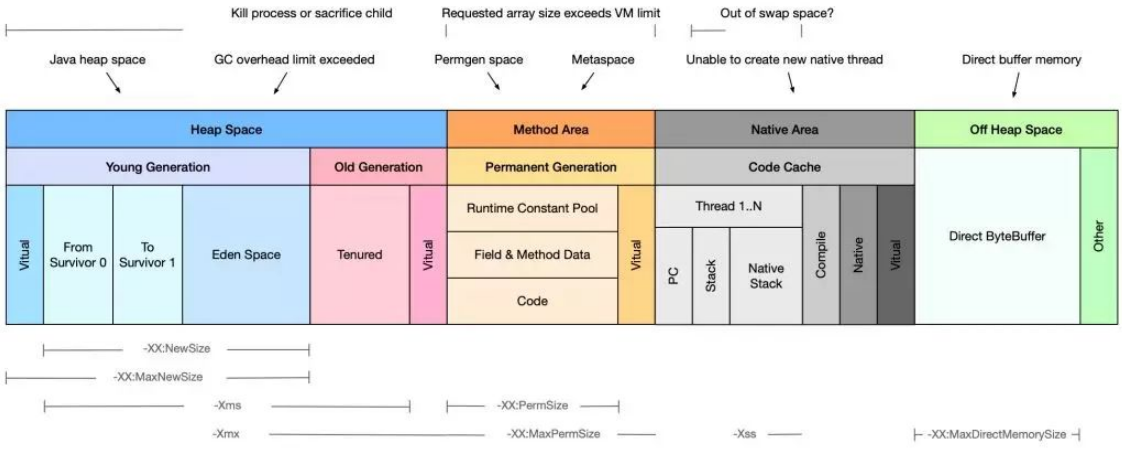
Stack Space用来做方法的递归调用时压入Stack Frame(栈帧)。所以当递归调用太深的时候，就有可能耗尽Stack Space，爆出StackOverflow的错误。
-Xss128k：设置每个线程的堆栈大小。JDK 5以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。根据应用的线 程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有 限制的，不能无限生成，经验值在3000~5000左右。
线程栈的大小是个双刃剑，如果设置过小，可能会出现栈溢出，特别是在该线程内有递归、大的循环时出现溢出的可能性更 大，如果该值设置过大，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。

3 JVM全貌



聊完了内存区域，接下来要聊垃圾回收了

4 补充



Java 内存模型 & OOM 错误

4.1 heap space

当堆内存（Heap Space）没有足够空间存放新创建的对象时，就会抛出 `java.lang.OutOfMemoryError:Javaheap space` 错误（根据实际生产经验，可以对程序日志中的 OutOfMemoryError 配置关键字告警，一经发现，立即处理）。

原因分析

`heap space` 错误产生的常见原因可以分为以下几类：

1、请求创建一个超大对象，通常是一个大数组。

2、超出预期的访问量/数据量，通常是上游系统请求流量飙升，常见于各类促销/秒杀活动，可以结合业务流量指标排查是否有尖状峰值。

3、过度使用终结器（Finalizer），该对象没有立即被 GC。

4、内存泄漏（Memory Leak），大量对象引用没有释放，JVM 无法对其自动回收，常见于使用了 File 等资源没有回收。

解决方案

针对大部分情况，通常只需要通过 `-Xmx` 参数调高 JVM 堆内存空间即可。如果仍然没有解决，可以参考以下情况做进一步处理：

1、如果是超大对象，可以检查其合理性，比如是否一次性查询了数据库全部结果，而没有做结果数限制。

2、如果是业务峰值压力，可以考虑添加机器资源，或者做限流降级。

3、如果是内存泄漏，需要找到持有的对象，修改代码设计，比如关闭没有释放的连接。

4.2 GC overhead limit exceeded

当 Java 进程花费 98% 以上的时间执行 GC，但只恢复了不到 2% 的内存，且该动作连续重复了 5 次，就会抛出 `java.lang.OutOfMemoryError:GC overhead limit exceeded` 错误。简单地说，就是应用程序已经基本耗尽了所有可用内存， GC 也无法回收。

此类问题的原因与解决方案跟 `Javaheap space` 非常类似，可以参考上文。

4.3 Permgen space

该错误表示永久代（Permanent Generation）已用满，通常是因为加载的 class 数目太多或体积太大。

原因分析

永久代存储对象主要包括以下几类：

1、加载/缓存到内存中的 class 定义，包括类的名称，字段，方法和字节码；

2、常量池；

3、对象数组/类型数组所关联的 class；

4、JIT 编译器优化后的 class 信息。

PermGen 的使用量与加载到内存的 class 的数量/大小正相关。

解决方案

根据 Permgen space 报错的时机，可以采用不同的解决方案，如下所示：

1、程序启动报错，修改 `-XX:MaxPermSize` 启动参数，调大永久代空间。

2、应用重新部署时报错，很可能是没有应用没有重启，导致加载了多份 class 信息，只需重启 JVM 即可解决。

3、运行时报错，应用程序可能会动态创建大量 class，而这些 class 的生命周期很短暂，但是 JVM 默认不会卸载 class，可以设置

`-XX:+CMSClassUnloadingEnabled` 和 `-XX:+UseConcMarkSweepGC` 这两个参数允许 JVM 卸载 class。

如果上述方法无法解决，可以通过 jmap 命令 dump 内存对象 `jmap-dump:format=b,file=dump.hprof<process-id>` ，然后利用 Eclipse MAT https://www.eclipse.org/mat 功能逐一分析开销最大的 classloader 和重复 class。

4.4 Metaspace

JDK 1.8 使用 Metaspace 替换了永久代（Permanent Generation），该错误表示 Metaspace 已被用满，通常是因为加载的 class 数目太多或体积太大。

此类问题的原因与解决方法跟 `Permgenspace` 非常类似，可以参考上文。需要特别注意的是调整 Metaspace 空间大小的启动参数为 `-XX:MaxMetaspaceSize`。

4.5 Unable to create new native thread

每个 Java 线程都需要占用一定的内存空间，当 JVM 向底层操作系统请求创建一个新的 native 线程时，如果没有足够的资源分配就会报此类错误。

原因分析

JVM 向 OS 请求创建 native 线程失败，就会抛出 `Unableto createnewnativethread`，常见的原因包括以下几类：

1、线程数超过操作系统最大线程数 ulimit 限制；

2、线程数超过 kernel.pid_max（只能重启）；

3、native 内存不足；

该问题发生的常见过程主要包括以下几步：

1、JVM 内部的应用程序请求创建一个新的 Java 线程；

2、JVM native 方法代理了该次请求，并向操作系统请求创建一个 native 线程；

3、操作系统尝试创建一个新的 native 线程，并为其分配内存；

4、如果操作系统的虚拟内存已耗尽，或是受到 32 位进程的地址空间限制，操作系统就会拒绝本次 native 内存分配；

5、JVM 将抛出 `java.lang.OutOfMemoryError:Unableto createnewnativethread` 错误。

解决方案

1、升级配置，为机器提供更多的内存；

2、降低 Java Heap Space 大小；

3、修复应用程序的线程泄漏问题；

4、限制线程池大小；

5、使用 -Xss 参数减少线程栈的大小；

6、调高 OS 层面的线程最大数：执行 `ulimia-a` 查看最大线程数限制，使用 `ulimit-u xxx` 调整最大线程数限制。

ulimit -a .... 省略部分内容 ..... max user processes (-u) 16384

4.6 Out of swap space

该错误表示所有可用的虚拟内存已被耗尽。虚拟内存（Virtual Memory）由物理内存（Physical Memory）和交换空间（Swap Space）两部分组成。当运行时程序请求的虚拟内存溢出时就会报 `Outof swap space` 错误。

原因分析

该错误出现的常见原因包括以下几类：

1、地址空间不足；

2、物理内存已耗光；

3、应用程序的本地内存泄漏（native leak），例如不断申请本地内存，却不释放。

4、执行 `jmap-histo:live<pid>` 命令，强制执行 Full GC；如果几次执行后内存明显下降，则基本确认为 Direct ByteBuffer 问题。

解决方案

根据错误原因可以采取如下解决方案：

1、升级地址空间为 64 bit；

2、使用 Arthas 检查是否为 Inflater/Deflater 解压缩问题，如果是，则显式调用 end 方法。

3、Direct ByteBuffer 问题可以通过启动参数 `-XX:MaxDirectMemorySize` 调低阈值。

4、升级服务器配置/隔离部署，避免争用。

4.7 Kill process or sacrifice child

有一种内核作业（Kernel Job）名为 Out of Memory Killer，它会在可用内存极低的情况下“杀死”（kill）某些进程。OOM Killer 会对所有进程进行打分，然后将评分较低的进程“杀死”，具体的评分规则可以参考 Surviving the Linux OOM Killer。

不同于其他的 OOM 错误，`Killprocessorsacrifice child` 错误不是由 JVM 层面触发的，而是由操作系统层面触发的。

原因分析

默认情况下，Linux 内核允许进程申请的内存总量大于系统可用内存，通过这种“错峰复用”的方式可以更有效的利用系统资源。

然而，这种方式也会无可避免地带来一定的“超卖”风险。例如某些进程持续占用系统内存，然后导致其他进程没有可用内存。此时，系统将自动激活 OOM Killer，寻找评分低的进程，并将其“杀死”，释放内存资源。

解决方案

1、升级服务器配置/隔离部署，避免争用。

2、OOM Killer 调优。

4.8 Requested array size exceeds VM limit

JVM 限制了数组的最大长度，该错误表示程序请求创建的数组超过最大长度限制。

JVM 在为数组分配内存前，会检查要分配的数据结构在系统中是否可寻址，通常为 `Integer.MAX_VALUE-2`。

此类问题比较罕见，通常需要检查代码，确认业务是否需要创建如此大的数组，是否可以拆分为多个块，分批执行。

4.9 Direct buffer memory

Java 允许应用程序通过 Direct ByteBuffer 直接访问堆外内存，许多高性能程序通过 Direct ByteBuffer 结合内存映射文件（Memory Mapped File）实现高速 IO。

原因分析

Direct ByteBuffer 的默认大小为 64 MB，一旦使用超出限制，就会抛出 `Directbuffer memory` 错误。

解决方案

1、Java 只能通过 ByteBuffer.allocateDirect 方法使用 Direct ByteBuffer，因此，可以通过 Arthas 等在线诊断工具拦截该方法进行排查。

2、检查是否直接或间接使用了 NIO，如 netty，jetty 等。

3、通过启动参数 `-XX:MaxDirectMemorySize` 调整 Direct ByteBuffer 的上限值。

4、检查 JVM 参数是否有 `-XX:+DisableExplicitGC` 选项，如果有就去掉，因为该参数会使 `System.gc()` 失效。

5、检查堆外内存使用代码，确认是否存在内存泄漏；或者通过反射调用 `sun.misc.Cleaner` 的 `clean()` 方法来主动释放被 Direct ByteBuffer 持有的内存空间。

6、内存容量确实不足，升级配置。

推荐工具&产品

JVM 内存分析工具 mat

1、Eclipse Memory Analyzer

https://www.eclipse.org/mat

阿里云 APM 产品，支持 OOM 异常关键字告警

2、ARMS

https://help.aliyun.com/document_detail/42966.html?spm=a2c4g.11174283.6.685.d69b668cuztvff

阿里 Java 在线诊断工具 Arthas(阿尔萨斯)

3、alibaba Arthas

https://github.com/alibaba/arthas