**一图解千愁，jvm内存从来没有这么简单过！**

<https://mp.weixin.qq.com/s/mB1TwKVULY7gSqXTW9NEoA>

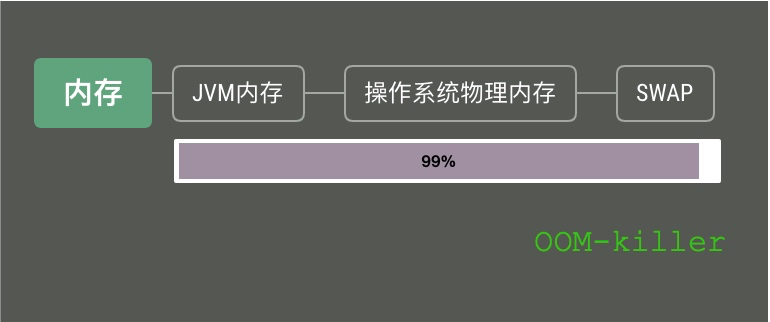
看到这张图的同学，千万不要到处分享。我们仅限于小范围讨论，因为这张图威力很大，是我花了10年时间才画出来的！

了解了这张图，会让你对JVM内存的划分有更深入的理解，而不仅限于什么虚拟机栈、程序计数器等比较浅显的认知。

那么这张图有什么用呢？在进行内存排查的时候，我们需要了解到底是哪一个部分除了问题。如果你找不对地方，肯定切入就比较困难，这会耗费你大量的精力。

一台4GB的机器，一般使用Xmx分配给JVM的，肯定不能太多。比如3.5GB之类的。这就太贪婪了，很容易造成JVM异常死亡。这是为什么呢？

这个比较好理解，因为在操作系统上，运行的不仅仅你的JVM应用，还会有其他一些守护进程，比如各种日志收集工具、监控工具、安全工具等。它们虽然占用的内存不是很多，但累加起来还是比较可观的。JVM内存和操作系统的剩余内存是一个此消彼长的关系，这些小内存挤占了JVM的发挥空间，就容易出问题。



JVM是我们的主体，所以要把它放在主人公的位置。这种划分方式，就可以把整个内存搞成JVM内存、操作系统物理内存、SWAP三个部分。

当JVM和其他程序占满了物理内存，接着占满了SWAP内存（交换分区一般不开，这个一会在说），当在需要申请内存空间的时候，操作系统发现：**完蛋了，没有可用的内存空间了。**

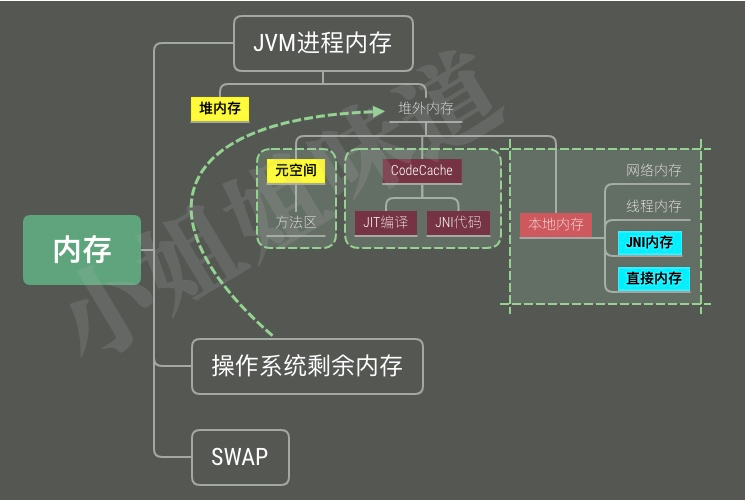
这个时候，Linux会启动oom-killer，杀死占用内存最大的进程，这个时候大概率是你可爱的JVM宝贝进程。

这里的oom，指的是操作系统的，而不是JVM的。所以你会发现：**你的java进程死了，但是什么都没有留下。就这么静悄悄的去了。**

这些信息，只能通过dmesg命令找到，属于操作系统范畴。

那么接下来，我们就上一下最主要的一张图，然后解释一下这十几部分都是干什么的。

我们依然把内存分为上面的三部分，但是对JVM的进程内存进行更细致的划分。



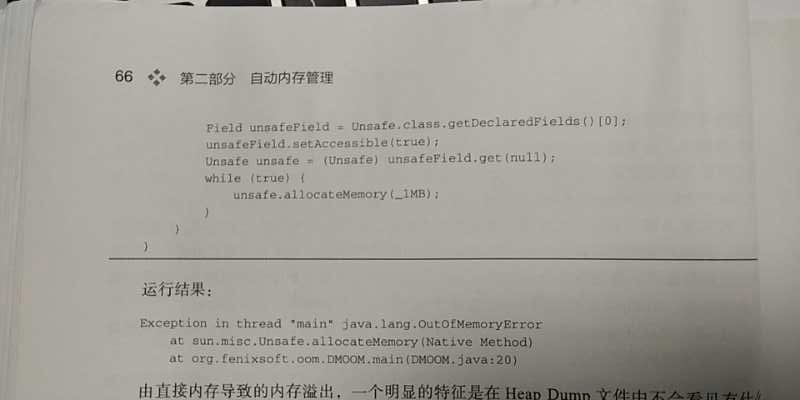
首先，对于JVM的内存，有堆内内存和堆外内存之分。

对于堆内内存，是我们平常打交道最多的地方，因为我们大部分Java对象，都是在堆上分配的。一旦有溢出问题，使用jmap + mat等一系列猛如虎的操作，就可以方便快捷的发现问题。

这是一个Java好手都能掌握的技能。

关键就是堆外内存那一部分，就十分的蛋疼了。因为杂七杂八的东西都在这里，很容易搞混。

可以看到，对于这部分的内存问题，即使是JVM界最权威的周老师的书籍，依然也有相关的错误。



这段代码的运行结果其实是错误的，这里的unsafe，并不是直接内存。

那我们就盘点一下里面都有些啥。

**第一，元空间**

元空间是jdk8以后才加入的，用来替换原来的永久代。也就是说，原perm区（永久代）中的方法区，也在这里。从它原来的名字就可以看出来，永久代指的就是那些变动很少的数据，稳定为主。比如我们在jvm启动时，加载的那些class文件；以及在运行时，动态生成的代理类。

比较坑的是，元空间的大小，默认是没有上限的。极端情况下，会一直挤占操作系统的剩余内存。

**第二、CodeCache**

很多文章对着一部分的介绍非常少，但其实这也是非常重要的一个非堆区域。因为JIT是JVM一个非常重要的特性，CodeCahe存放的，就是即时编译器所生成的二进制代码。当然，JNI的代码也是放在这里的。

这个空间在不同的平台，大小都是不一样的，但一般够用了。也有同学手贱把这个区域调的非常的小，这种情况下，JVM不会溢出，这个区域也不会溢出，但是会退化成解释型执行模式，速度和JIT不可同日而语，慢个数量级也是可能的。

**本地内存**

其实，在聊天的时候，我们相互谈到的堆外内存，大部分指的是这里，大部分出问题的，也是这里。它有更细致的划分。

**（1）网络内存**

网络连接也是要占用很多内存的。这个连接就非常有意思，你可以认为它是操作系统内核所占用的内存，也可以认为是JVM进程占用的内存。

如果你的系统并发非常高，这部分内存的占用也是比较多的。因为连接一般对应着网卡的数据缓冲区，还有文件句柄的耗费。

**（2）线程内存**

同样的，如果你造的线程非常多，JVM除了占用Thread对象本身很小的一部分堆内存，大部分是以轻量级进程的方式存在于操作系统。

这同样是一个积少成多的内存区域，但一般不会发生问题。

**（3）JNI内存**

上面谈到CodeCache存放的JNI代码，JNI内存就是指的这部分代码所malloc的具体内存。

比如Java的zip库，就不是在JVM的堆里完成的，而是开辟了一个堆外的缓冲池进行运算。

**（4）直接内存**

直接内存，指的是使用了Java的直接内存API，进行操作的内存。这部分内存可以受到JVM的管控，比如ByteBuffer类所做的事情。

ByteBuffer底层是用的unsafe,但unsafe是不受直接内存的管控的，它们不是一个东西。

上面提到的书中直接使用unsafe程序，并不会造成JVM直接内存溢出，反而会造成操作系统内存溢出。

那这些内存我们如何看到呢？

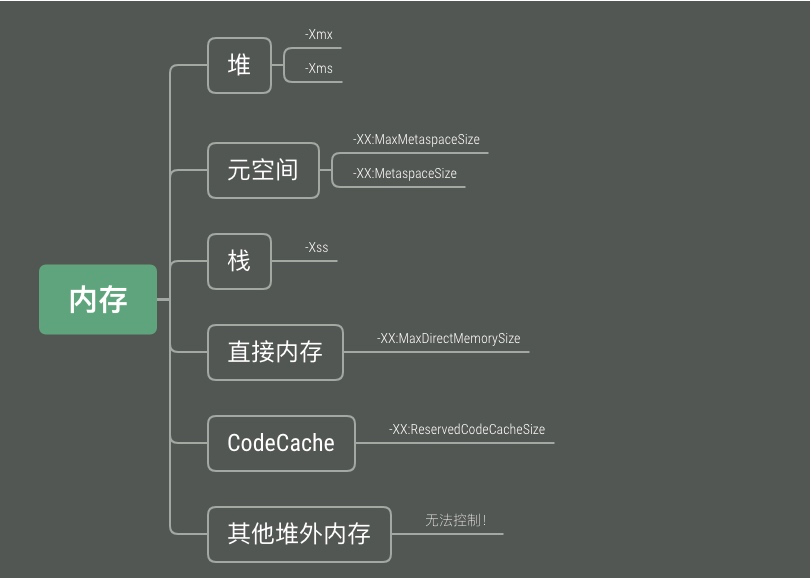
linux下有一个命令lsof，可以看到JVM进程所关联的所有句柄信息，一般可作为参考。

近一步，使用pmap函数，即可观测到具体的内存分布。但是不要怕，有很多是共享内存。

这个具体的过程，可以参见之前写的一篇堆外内存排查的文章。

如果你了解了图中这些内存划分，就会很容易了解，为什么NMT工具无法显示JNI内存的统计。

接下来，我们总结一下，这些内存区域，哪些参数能够控制它们。



* 堆  -Xmx  -Xms
* 元空间 -XX:MaxMetaspaceSize  -XX:MetaspaceSize
* 栈 -Xss
* 直接内存  -XX:MaxDirectMemorySize
* JIT编译后代码存放 -XX:ReservedCodeCacheSize
* 其他堆外内存 无法控制！随缘吧。

可以看到，堆外内存的占用，其实还是比较多的。如果你太贪婪，整个内存很容易就玩玩。

**一般的，我们使用操作系统的2/3作为堆空间，是比较合理的。**这是一个经验值。比如6GB的内存，你分配给JVM的，最好不要超过4GB。

还有，我们上面谈到的swap交换分区，在高并发应用中，一般是关掉的。因为它会造成频繁的页交换，在GC的时候，会引起严重的卡顿。

但要辩证的思维看待问题。对于低频的，对内存大小有非常大的依赖的情况下，SWAP不仅要开，还要开的大一些。