java内存区域与内存异常

笔记本：	jvm		
创建时间：	2018/9/11/周二 14:27	**更新时间：**	2018/9/15/周六 16:27
作者：	1634896520@qq.com		
URL：	https://blog.csdn.net/pfnie/article/details/52766204		

java内存区域与内存异常

一、内存中到底放了什么？

Java虚拟机所管理的内存包括：

1. 程序计数器
2. Java虚拟机栈
3. 本地方法栈
4. Java堆
5. 方法区
6. 运行时常量池
7. 直接内存

（一）程序计数器

这是一块**较小**的内存空间，它可以看作是当前线程所执行的字节码的**行号指示器**。每条线程都需要一个独立的程序计数器，各条线程之间计数器不受影响，独立存储，我们称这类内存区域为“**线程私有**”的内存。
如果线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址：如果正在执行的是Native方法，那么这个计数值为空。

此内存区域是唯一一个在Java虚拟机中没有任何 **OutofMemoryError**情况的内存区域。

（二）Java虚拟机栈

Java虚拟机栈也是**线程私有**的，它们的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型，每个方法在执行的同时都会创建一个栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每个方法从调用直至执行完成的过程，就对应一个栈帧在虚拟机栈中入栈到出栈的过程。

两种异常情况：① 如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常：
② 如果扩展时无法申请到足够的内存，就会抛出OutofMemoryError异常。

（三）本地方法栈

本地方法栈与虚拟机栈所发挥的作用是非常相似的，与虚拟机栈一样，本地方法栈区域也会抛出StackOverfloeError和OutofMemoryError ，**不同之处在于**：

虚拟机栈为虚拟机执行Java方法，也就是字节码服务，而本地方法栈则为虚拟机使用到的Native方法服务。在虚拟机规范中对本地方法栈中方法使用的语言、使用方法与数据结构并没有强制规定，因此具体的虚拟机可以自由的实现它。

（四）Java堆

Java堆是Java虚拟机所管理的**内存中最大**的一块，Java堆是被所有**线程共享**的一块区域，在虚拟机启动时候创建，此内存区域的唯一目的就是**存放对象实例。**

Java堆是垃圾收集管理的主要区域，因此也被成为“GC堆”。

从内存回收的角度来看： 由于现在收集器基本都是采用分代收集算法，所以Java堆中还可以细分为 新生代 和老年代。
从内存分配的角度来看： 线程共享的Java堆中可能划分出多个线程私有的分配缓冲区。

Java堆是可扩展的，通过（ -Xmx 和 -Xms 控制）
如果在堆中没有内存完成实例分配，并且也无法再扩展时，将会抛出 OutofMemoryError 异常

（五）方法区

方法区和Java堆一样，是各个**线程共享**的内存区域。用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据，针对常量池的回收和类型的加载。

和Java堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。

当方法区无法满足内存分配需求的时候，将会抛出OutofMemoryError

（六）运行时常量池

Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译器生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

它相对于class文件常量池的另外一个重要特征是具有动态性。运行期间也可将新的常量放入池中。

当常量池无法申请到内存时会抛出OutofMemoryError异常。

（七）直接内存

它并不是虚拟机运行时数据去的一部分，也不是Java虚拟机规范中定义的内存区域。本机直接内存的分配不会受到Java堆大小的限制。但是既然是内存，救护收到本机总内存（包括RAM以及SWAP区或者分页文件）大小以及处理器寻址空间的限制。服务器在给虚拟机配置参数的时候，会根据实际内存设置-Xmx等参数信息，但经常忽略直接内存，使得各个内存区域总和大于物理内存限制，从而导致动态扩展时出现OutofMemoryError异常

二、HotSpot虚拟机对象探秘

（一）对象的创建过程

虚拟机在遇到一条new指令时：

①检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并检查这个符号引用代表的类是否被加载、解析和初始化过。

若没有，则必须先加载此类。

②在类加载检查通过后，接下来虚拟机将为新生对象分配内存。

（这里有一个知识点叫作“指针碰撞”：假设Java堆中的内存是绝对规整的，所有用过的内存放在一边，空闲的内存放在一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲那边挪动一段与对象大小相等的距离，这种分配方式就叫作“指针碰撞”）但如果Java堆中的内存不是规整的，那就不能指针碰撞，这个时候就需要用到“空闲列表”的分配方式（：在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式就叫作“空闲列表”）

举例：在使用Serial\parNew等带Compact过程的收集器时，通常采用指针碰撞，而使用CMS这种基于Mark-Sweep算法的收集器时，采用空闲列表。

③内存分配完成后，虚拟机需要将分配搭配的内存空间都初始化为零值。

④接下来，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例，如何才能找到类的元数据信息，对象的hash码，对象的GC分代年龄等信息。

⑤最后，执行new 指令之后会接着执行<init>方法，把对象按照程序员的一员进行初始化。

```
*：如何解决对象创建在虚拟机中频繁的行为？也就是说，在并发条件下，如何分配内存。
    第一种方案：对分配内存空间的动作进行同步处理，也就是虚拟机采用CAS配上失败重试的方式保证更新操作的原子性。
    第二种方案：把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在java堆中预先分配一小块内存，称为本地线程分配缓冲（TLAB），可通过-XX:+/UserTLAB参数来设定。
```

（二）对象的内存布局

分为三个区域：对象头、实例数据、对齐填充。

对象头又分为：①用于存储对象自身的运行时数据 Mark Word
②类型指针

实例数据：对象真正存储的有效信息
对齐填充：当对象实例数据部分没有对齐时，就需要通过对齐填充来补充。

（三）对象的访问定位
句柄：如果使用句柄访问的话，那么Java堆将会划分出一块内存来作为句柄驰，reference中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。
优点：reference中存储的是稳定的句柄地址。

直接指针：如果使用指针访问，那么Java堆对象的布局中就必须考虑如何放置访问类型数据相关信息，而reference中存储的直接就是对象地址。
优点：速度更快

三、OutofMemoryError异常

在Java虚拟机规范中，除了程序计数器不会发生此异常外，其他内存区域都会有可能发生此异常。

（一）Java堆溢出

Java堆用于存储对象实例，只要不断地创建对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收机制清除这些对象，那么在对象数量到达最大堆时就会产生此异常。
比如：
在下面程序中设置参数：

```
-Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError
```

```
import java.util.ArrayList;
import java.util.List;

public class JVMtest
      static class HeapOOM

      public static void main String args
            List<HeapOOM> list = new ArrayList<HeapOOM> ;
            while  true
                  list.add new HeapOOM ;
```

控制台run configration输出：

```
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid45532.hprof ...
Heap dump file created [28074773 bytes in 0.108 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
      at java.util.Arrays.copyOf(Arrays.java:3210)
      at java.util.Arrays.copyOf(Arrays.java:3181)
      at java.util.ArrayList.grow(ArrayList.java:261)
      at java.util.ArrayList.ensureExplicitCapacity(ArrayList.java:235)
      at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:227)
      at java.util.ArrayList.add(ArrayList.java:458)
      at JVMtest.main(JVMtest.java:11)
```

解决方案：

先通过内存映像分析工具（Eclipse Memory Analyzer）对Dump出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，也就是要分清到底是出现了内存泄漏（Memory leak），还是内存溢出（Memory Overflow），如果是内存泄漏，可进一步通过工具查看泄漏对象到GC Roots的引用链。于是就能找到泄漏对象是通过怎样的路径与GC Roots相关联并导致垃圾收集器无法自动回收它们的。掌握了泄漏对象的类型信息，以及GC Roots引用链的信息，就可以比较准确地定位出泄漏代码的位置。如果不存在泄漏，换句话说就是内存中的对象确实都还必须存活着，那就应当检查虚拟机的堆参数（-Xmx与-Xms），与机器物理内存对比看是否还可以调大，从代码上检查是否存在某些对象生命周期过长、持有状态时间过长的情况，尝试减少程序运行期的内存消耗。

（二）虚拟机栈和本地方法栈溢出

在单个线程下，无论是由于栈帧太大，还是虚拟机栈容量太小，当内存无法分配的时候，虚拟机抛出的都是 StackOverflowError 异常，而不是OOM。

多线程的情况下，是会出现OOM的！

为什么多线程的情况下就会产生OOM？

有一个操作系统为2G，我们分配给两个线程，每个800M，也就还剩400M，这样的话，有一个线程不够用的话，就会在400里边申请，所以如果剩下的越多，出现OOM的可能性越小，如果每个分配950M，这样就剩100M，这样的话出现OOM的可能性就更大。如果在增加线程，系统对每一个线程非配的内存是一定的，所以剩下的内存更少，这样的话，出现OOM的可能更大，但这都是相对而说。

如何解决OOM问题？

如果是建立过多线程导致内存溢出，在不能减少线程数或者更换64虚拟机的情况下，就只能通过减少最大堆和减少栈容量来换取更多的线程。

（三）方法区和运行时常量池溢出

运行时常量池溢出测试：

如果要向运行时常量池中添加内容，最简单的做法就是使用String.intern()这个Native方法。该方法的作用是：如果字符串常量池中已经包含一个等于此String对象的字符串，则返回代表池中这个字符串的String对象；否则，将此String对象包含的字符串添加到常量池中，并且返回此String对象的引用。

方法区溢出测试：

基本思路是运行时产生大量的类去填满方法区，直到溢出。克通过CGLib直接操作字节码运行时生成大量的动态类。

（四）本机直接内存溢出

由DirectMemory导致的内存溢出，一个明显的特征是在Heap Dump文件中不会看见明显的异常，如果发现OOM之后Dump文件很小，而程序中又直接或者简使用了NIO，那就可以考虑检查一下是不是这个方面的原因。