第2章 Java内存区域与内存溢出异常

2016年3月3日 20:29

概要：从概念上介绍JVM内存的各个区域，并讲解这些区域的作用，面向对象，以及可能会产生的问题

第2章目录

2.2运行时数据区域

有的随着线程开启与结束而创建与销毁，有的会随着JVM启动就创建

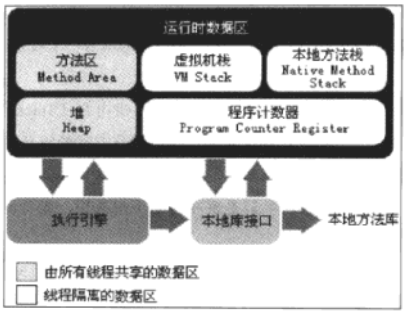

图 2-1 Java 虚拟机运行时数据区

2.2.3 本地方法栈

功能与虚拟机栈类似，只不过虚拟机栈是服务于java方法，本地方法栈是服务于调用本地方法时（native method）。抛出异常与虚拟机栈一样。

2.2.1 程序计数器

较小的一块内存空间。用于指示当前执行的字节码的行号。字节解释器就是通过改变计数器从而改变下一条要执行的字节码指令。

在Java中多线程是轮流切到时间片的，因而每个线程都应该有一个自己的程序计数器，好让拿到CPU时间切回来执行时可以知道执行到哪。因此程序计数器是线程私有的内存空间。

如果是java方法，计数器记录的是当前执行的字节码指令的地址，如果是native方法，则为0.此内存区域是唯一在JVM规范中没有OOM异常的区域。

2.2.2 JAVA虚拟机栈

虚拟机栈也是线程私有的，生命周期与线程一样。描述的是Java方法的执行的内存模型：每个方法执行时，都会创建一个栈帧，用于存储局部变量，操作数栈，动态链接，方法出口等。方法从调用到执行完成，对应着栈帧从入栈到出栈的过程。（我的理解是：一个线程对应着一个虚拟机栈，系统分给虚拟机栈空间很有限的，每执行一个方法就创建一个栈帧，用于保存方法的局部变量，出口信息等等）

局部变量表放的是编译器可知的各种基本数据类型，对象引用等。局部变量表所需空间在编译期即可决定，在运行期中并不会改变。

可能出现的异常：当线程请求的栈大于虚拟机规定的栈空间时，跑出stackoverflow异常；如果JVM栈空间可扩展，当线程申请不到足够的内存时，就会抛出OOM异常。

第2章 Java内存区域

2016年3月3日 21:08

2.2.4 JAVA堆

Java堆是JVM内存管理中内存最大的一块。
在虚拟机启动时创建，被所有线程共享，用于存放对象实例。几乎所有的对象都在这里分配内存。JVM规范描述过：所有的对象和数组的都应该在分配在堆上。
Java堆是垃圾收集的主要区域，因此很多时候被称为GC堆。
从内存回收角度看，现在收集器基本采用分代收集算法，堆中被分为老年代和新生代。细致一点有Eden空间，formSurvivor，toSurvivor空间等。
根据虚拟机规范，堆可以在物理存储上不连续，在逻辑上连续即可，像我们的磁盘空间一样。在实现时，可固定空间也可扩展（通过-Xmx，-Xms），当可扩展时堆中内存不够分配给对象实例时，就抛出OOM异常

2.2.5

方法区与堆一样，都被所有线程共享的。方法区存储着虚拟机加载了的类的信息，常量，静态变量，即时编译器编译了的代码（什么是即时编译器）。

在hotspot虚拟机上，很多人将方法区称为永久代，然而两者并不等价，只不过是hotspot用了永久代来实现方法区而已，用管理堆的方法来管理方法区，这样就不用写实现方法区的代码。方法区和堆一样，可固定大小也可扩展，还可以选择不进行垃圾收集。当方法区无法满足内存分配需求时，就抛出OOM异常。

2.2.6 运行时常量

运行时常量池是方法区的一部分。Class文件中除了类的版本，字段，方法，接口的信息，还有一项是常量池，用于存储编译期生成的各种字面量和符号引用(什么是符号引用)，这部分内容将在类加载进方法区时存进运行时常量池。

运行时常量池相对于class文件中的常量池的一个重要特征是具备动态性，java语言并不要求常量只有在编译期才能产生，并非只有class文件中常量池的常量才能进入运行时常量池，在运行时也能产生常量，并存进常量池。

既然运行时常量池是方法区的一部分，自然受到方法区内存的限制，当运行时常量池没法申请到内存时，就会报OOM异常。

2.2.7 直接内存

直接内存不是虚拟机运行时数据一部分，也不是jvm规范中的内存区域。但是这部分内存被频繁使用，可能导致OOM异常。

本机直接内存的分配不受到java堆大小的限制，但既然是内存，肯定受到本机内存的限制。当配置虚拟机的堆大小时，会根据实际内存设置-Xmx，但是会忽略了直接内存，从而导致各个内存区域总和大于物理内存限制。从而导致扩展内存时OOM异常

对象的创建，布局，访问

2016年3月3日　23:49

2.3 在一个具体的虚拟机HotSpot和常用的区域堆上讲解对象如何创建，如何布局，如何访问

2.3.1 对象的创建

检查：虚拟机遇到一条new指令时，首先会去检查这个指令的参数能不能在常量池定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经加载，解析和初始化。如果没有，就要先执行类的加载。

检查通过后：虚拟机为新生对象分配内存。所需内存在类被加载后就可以确定，为对象分配内存的任务就等于在堆中划一块确定大小的内存出来。

假设堆内存是绝对规整的，空闲和使用各占一边，中间放一个指针作为分界点，那分配内存的行为就只是将指针挪动与对象内存大小相等的距离即可，这种分配叫“指针碰撞”。如果里面内存不是规整的，就无法进行指针碰撞，就需要维护一个列表记录空闲空间与已用空间，在列表中空闲空间分配一块内存给对象后就更新列表，这种分配称为“空闲列表”。选哪种方式取决于内存是否规整，而内存是否规整又跟收集算法是否有压缩功能有关。因此使用不同收集器用不同的分配方式。

有一点需要考虑的是，创建对象是非常频繁的行为，当A对象正拿到内存后指针还没修改时，对象B拿了这个指针来分配内存。解决这种情况有两种方法：1是采用同步机制确保操作的原子性，2是内存分配在不同空间进行，即每个线程都在堆中分配一小块内存用作本地线程分配缓冲，TLAB，哪个线程要分配就在哪个线程的TLAB上分配内存给对象。虚拟机是否用TLAB，可通过参数-XX：+UseTLAB设定

内存分配后，虚拟机需要将分配到的空间初始化为零（使用TLAB则在分配时就进行），所以这才保证了对象在实例化时可以不赋值直接使用。

最后是对对象头的设置。（例如这个对象是哪个类的实例，怎样找到类的元数据，对象的哈希码等）

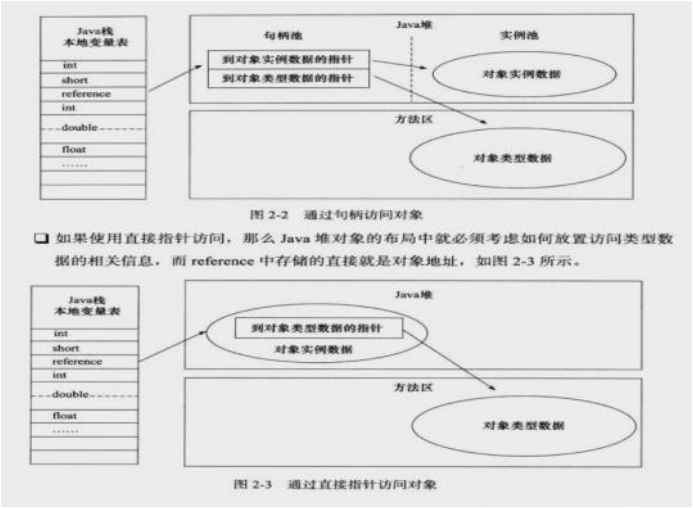

图 2-2　通过句柄访问对象

❑ 如果使用直接指针访问，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的相关信息，而 reference 中存储的直接就是对象地址，如图 2-3 所示。

图 2-3　通过直接指针访问对象

2.3.2 对象内存布局

对象在内存中的存储可以分为3个区域：对象头，实例数据，对齐补充

对象头:包含两方面信息,第一部分是存储自身运行时数据，如哈希码，GC年龄，锁状态等另一部分是类型指针，对象指向它的类元数据（什么是类元数据）的指针，用于确定对象是哪个类的实例。

实例数据：对象存储的真正有效信息。也就是代码中定义的字段内容。

对齐补充：并步一定存在，仅仅起占位符的作用

2.3.3对象的访问定位

为了使用对象，我们需要通过栈上的reference数据来操作堆上的具体对象。具体怎样取到堆上对象，是各个虚拟机的自己实现。目前主流的有使用句柄和直接指针两种，

实战OOM异常

2016年3月4日 0:30