运行时数据区域

Jvm在执行java程序的过程中会把它所管理的内存区域划分为若干个不同的数据区域，这些区域各有各的用途。

Jvm管理的运行时内存包含以下数据区域

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

程序计数器：是一个块较小的内存空间，可以看做是当前线程锁执行的字节码的行号指示器。

字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。

Jvm的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的。

每条线程都有一个独立的程序计时器，各条线程之间计数器互不影响，独立存储。这被称为线程私有内存。

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自己吗指令的地址，如果正在执行的是native方法，这个计数器的值为空。

此内存区域是唯一一个在jvm规范中没有规定任何的OutOfMemaryError情况的区域。

Java虚拟机栈

Java虚拟机栈也是线程私有的，它的声明周期与线程相同。它描述的是java方法执行的内存模型。

每个方法在执行的时候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，注意这里存放的不是真正的方法中的对象，而是对对这些运行时对象的引用或句柄，当然基本数据类型存储的就是其本身，这里也看出java方法是按值传递。

方法被调用到执行完成对应着一个栈帧在jvm中入栈到出栈的过程。

局部变量表存放编译期可知的各种基本数据类型、对象引用和returnAddress类型。

局部变量表所需的内存空间在编译期完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。

两种异常：栈溢出StackOverflowError和内存溢出OutOfMemaryError

本地方法栈：与java虚拟机栈非常相似，只是本地方法栈是为jvm使用到的Native方法服务。

本地方法栈也会抛出栈溢出StackOverflowError和内存溢出异常

Java堆：java堆是被所有线程贡献的一块内存区域，是jvm所管理的内存中最大的一块，这块区域在虚拟机启动时即创建。Java堆内存的唯一作用是存放对象实例。

几乎所有的对象实例和数组都在这里分配内存。

Java堆是GC收集管理的主要区域。

从内存收集的角度来看，由于现在收集器基本都采用分代收集算法，所以java堆中还可以细分为新生代和老生代。

方法区：与java堆一样，是各个线程共享的内存区域，它用于存储已经被虚拟机加载的类信息、常量、静态变量、即使编译后的代码等区域。

很多人都愿意将方法区成为永久代，本质上两者并不等价。

在jdk1.7 的hotspot中，已经将原本放在永久代的字符串常量池移出。

垃圾收集在这个区域是比较少出现的。

这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载。

该区域会抛出内存溢出异常OutOfMemaryError

运行时常量池：是方法区的一部分。Class文件信息除了有类的版本、字段、方法、接口等信息外，还有一项信息是常量池，用于存放编译期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。

运行时常量池相对于class文件常量池的另外一个重要特征是具备动态性，运行期间也可能有新的常量放入池中。

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，会抛出内存溢出异常。

直接内存：并不不是jvm运行时数据区域的一部分，也不是jvm规范中定义的内存区域，即它不在jvm管理的范围内。该区域是NIO时使用，直接分配的堆外内存。

该区域受计算机内存大小的限制，会抛出内存溢出异常。

对象的创建：

对象所需的内存大小在类加载完成后便看完全确定。

分配内存的一种方式，分为已分配内存和未分配内存，使用指针来作为分界点的指示器，当创建对象分配内存时，移动指针以分配指定大小的内存空间，这叫做指针碰撞。

另一种分配内存的方式是空闲列表法。

选择哪种分配方式要依据java堆是否规整，而这又由GC是否带有压缩整理功能 决定。

并发情况下的内存分配：一种方案是jvm采用CAS配上失败重试的方式保证更新操作的原子性；另一种方案是，每个线程在java对中预先分配一小块内存，称为本地线程分配缓冲。

初始化：内存分配完成后，jvm需要将分配到的内存空间都初始化为零值。这一步操作保证了对象的实例字段在java代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

接下来jvm需要对对象进行必要的设置，即设置对象头，包括对象所属类信息，对象的类的元数据信息，对象的hashcode，对象的GC分代年龄等信息。

完成上述过程后，从jvm的角度来看一个对象已经产生了，但从程序的角度来看，对象的创建才刚刚开始：<init>方法还没有执行，所有的字段都还为零。

因此执行new指令之后，会接着执行<init>方法，把对象按照程序员的意愿进行初始化，这时一个真正的对象才建立起来。

对象的内存布局：

1. 对象头
2. 实例数据
3. 对齐填充

对象头包含：

1. 用于存储对象自身的运行时数据，如hashcode，GC分代年龄，锁状态标志，线程持有的锁，偏向线程ID，偏向时间戳等。在32位和64位机器上分别为32bit和64bit。
2. 类型指针：即对象指向它的类元数据的指针
3. 如果对象是数组，那么还包括一块用于记录数组长度的数据。

Jvm可以根据普通对象的元数据信息确定java对象的大小，但是从数组的元数据中无法确定数组的大小，所以在数组的头部记录了数组的长度。

实例数据：该部分是对象真正存储的有效信息，也是在程序代码中所定义的各种类型的字段内容。无论是从父类继承下来的还是子类中定义的都需要记录起来。对象中字段的存储顺序受jvm和字段定义的顺序的影响。

对齐填充：该部分并不是必然存在的，也没有特别的含义，仅仅起到了占位符的作用。

对象的访问定位

Java程序需要通过栈上的reference数据来操作堆上的具体对象。

由于jvm规范没有规定，对象的访问方式取决于jvm的具体实现，目前主流的访问方式有两种：

1. 使用句柄
2. 直接指针

如果使用句柄，那么java堆中将会划分出一块内存来作为句柄池，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域类型数据各自的具体地址信息。

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。

第二种方式操作更快，因为直接得到的就是对象在堆中的地址，hotspot采用第二种方式进行对象的访问。

//============20171027