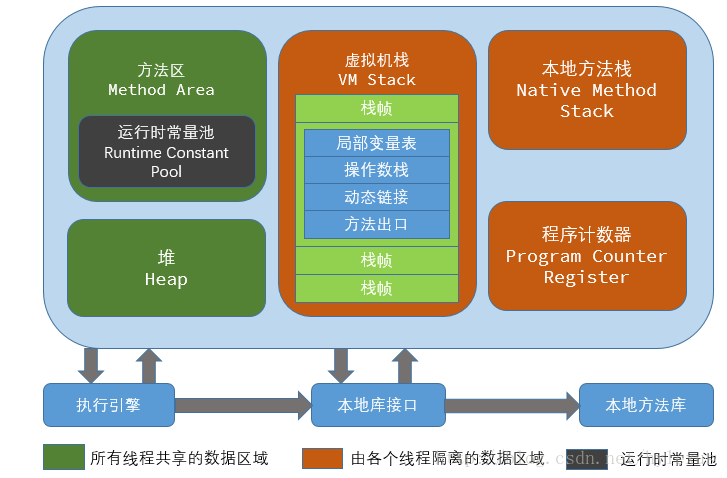
# Java内存区域与内存溢出异常

1. 运行时数据区域

Java虚拟机在执行java程序的过程中会把它所管理的内存划分为若干个不同的数据区域，这些区域都有自己各自的用途，以及创建和销毁的时间，有的区域随着虚拟机进程的启动而存在，有些区域则是依赖用户线程的启动和结束而建立和销毁。

Java虚拟机所管理的内存将会包括以下几个运行时数据区域



* 程序计数器

程序计数器是一块较小的内存空间，它的作用可以看作是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，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，分支，循环，跳转，异常处理，线程恢复等基础功能都需要依赖这个计数器来完成。

由于java虚拟机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，在任何一个确定的时刻，一个处理器只会执行一条线程中的指令。因此，为了线程切换后能够恢复到正确的位置，每条线程都需要一个独立的程序计数器，各个线程之间的计数器互不影响，独立存储，这类内存区域被称为“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，这个计数器值则为空，此内存区域是唯一一个在java虚拟机规范中没有规定任何OutOfMemoryError情况的区域

* Java虚拟机栈

与程序计数器一样，Java虚拟机栈也是线程私有的，它的生命周期与线程相同。虚拟机描述的是Java方法执行的内存模型：

每个方法被执行的时候都会同时创建一个栈帧，用于存储局部变量表，操作栈，动态链接，方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机中从入栈到出栈的过程

|  |
| --- |
| 有关栈帧：摘录<http://www.cnblogs.com/noKing/p/8167700.html>  **什么是栈帧？**  栈帧是一种数据结构，用于虚拟机进行方法的调用和执行  栈帧是虚拟机的栈元素，也就是入栈和出栈的一个单元  其他答案：  栈帧是用来存储数据和部分过程结果的数据结构，同时也被用来处理动态链接，方法返回值和异常分派  **栈帧在什么地方？**  栈帧所在位置：内存->运行时数据区->某个线程对应的虚拟机栈->栈帧  **栈帧的含义？**  每个方法的执行和结束都对应着栈帧的入栈和出栈  入栈表示被调用，出栈表示执行完毕或者返回异常  一个虚拟机栈对应一个线程，当前CPU调度的那个线程叫做活动线程；一个栈帧对用一个方法，活动线程的虚拟机里最顶部的栈帧代表了当前正在执行的方法，而这个栈帧也被叫做“当前栈帧”  **栈帧既然是个数据结构，都有哪些数据？’**  局部变量表，操作数栈，动态链接，方法返回地址，附加信息  **栈帧的大小是在什么时候确定的？**  编译程序代码的时候，就已经确定了局部变量表和操作数栈的大小，而且在方发表的code属性中写好了，不会受到运行数据的影响  **什么事局部变量表？**  局部变量表就是一片连续的内存空间，最小单位是Slot，用来存放方法参数和方法内部定义的局部变量  虚拟机里没有明确指明一个Slot的内存空间大小。但是boolean，byte，char，short，int，float，reference，returnAddress类型的数据都可以用32位空间或更小的内存来存放，这些类型占用一个Slot。Java中的long和double类型都是64位，占用两个Slot（只有double和long是jvms里明确规定的64位数据类型）  **虚拟机如何调用这个局部变量表？**  局部变量表是有索引的，就像数组一样，从0开始，到表的最大索引，也就是Slot的数量  -1。要注意的是：方法参数的个数+局部变量的个数≠Slot的数量，因为Slot的空间是可以复用的，当pc计数器的值已经超出了某个变量的作用域时，下一个变量不必使用新的Slot空间，可以去覆盖前面的那个空间  特别的：  JVMS7中：  在一个实例方法的调用时，局部变量表的第0位是一个指向当前对象的引用，也就是java里的this  **什么是操作数栈**  每个栈帧都包含一个被叫做操作数栈的后进先出的栈，叫操作栈，或者叫做操作数栈  通常情况下，操作数栈指的就是当前栈帧的操作数栈  **操作数栈有什么用？**   1. 栈帧刚创建时，里面的操作数时空的 2. Java虚拟机提供指令来让操作数栈对一些数据进行入栈操作，比如可以把局部变量表的数据，实例的字段等数据入栈 3. 同时也有指令来支持出栈操作 4. 向其他方法传参的参数，也存在操作数栈中 5. 其他方法返回的结果，返回时存在操作数中   **操作数栈本身就是一个普通的栈吗？**  栈就是栈，再加上数据结构所支持的一些指令和操作  但是，这里的栈有约束  操作数栈区分类型，操作数栈中严格区分类型，而且指令和类型也得严格匹配  **栈帧和栈帧之间是完全独立的吗？**  本来栈帧作为虚拟机栈的一个单元，应该是栈帧之间完全独立的  但是，虚拟机进行了一些优化：为了避免过多的方法之间参数的复制传递，方法返回值的复制传递等一些操作，就让一部分数据进行栈帧之间的共享  **什么是动态链接？**  一个方法调用另一个方法，或者一个类使用另一个类的成员变量时，需要知道被调用者的名字，可以不认识它，但是调用它就需要知道它的名字，符号引用就相当于名字，这些被调用者的名字就存放在java字节码文件里  名字知道以后，java真正在运行起来的时候，不是真的就能靠名字（符号引用）就能找到相应的类和方法的  需要解析成相应的直接引用，利用直接引用来准确地找到  举个例子，就相当于我在0X0300H这个地址存入了一个数526，为了方便编程，我给这个地址起了个别名为A，以后我编程的时候（运行之前）可以用别名A来暗示访问这个空间的数据，但是其实程序运行起来后，实质上还是去寻找0X0300H这片空间来获取526这个数据的  这样的符号引用和直接引用在运行时进行解析和链接的过程，叫做动态链接  **动态链接的前提？**  每一个栈帧内部都要包含一个指向运行时常量池的引用，来支持动态链接的实现  **有关运行时常量池：摘录自：**<https://www.cnblogs.com/xiaotian15/p/6971353.html>  **运行时常量池简介**  运行时常量池，它是方法区的一部分，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，用于存放编译期间生成的各种字面量和符号引用，这部分内容将在类加载后存放发到常量池中  运行时常量是相对于常量来说的，它具备一个重要特征：动态性，值相同的动态常量与我们通常说的常量来说只是来源不同，但是都存储在池内同一块内存区域，java语言并不要求常量一定只能在编译期产生，运行期间也可能产生新的变量，这些常量被放在运行时常量池中。  这里说的常量包括：基本类型包装类（包装类不管理浮点型，整型只会管理-128到127）和String（也可以通过String.intern()方法强制将String放入常量池中）  **Class文件中的信息常量池**  在Class文件结构中，最头的4个字节用于存储Megic Number，用于确定一个文件是否能被JVM接受，再接着4个字节用于存储版本号，前两个字节存储次版本号，后两个存储主版本号，再接着是用于存放常量的常量池，由于常量的数量是不固定的，所以常量池的入口放置一个U2类型的数据存储常量池容量计数值  常量池主要用于存放两大类常量：字面量和符号引用量，字面量相当于java语言层面常量的概念，如文本字符串，声明为final的常量值等。符号引用则属于编译原理方面的概念，包括了如下三种类型的常量：   * 类的接口和全限定名 * 字段名称和描述符 * 方法名和描述符   **常量池的好处**  常量池是为了避免频繁的创建和销毁对象而影响系统性能，其实现了对象的共享。例如字符串常量池，在编译阶段就把所有的字符串文字放到一个常量池中   * 节省内存空间：常量池中所有相同的字符串常量被合并，只占用一个空间 * 节省运行空间：比较字符串时，==比equals()更快，对于两个引用变量，只用==判断引用是否相等，也就可以判断实际值是否相等   双等号==的含义： 基本数据类型之间应用双等号，比较的是他们的数值  符合数据类型（类）之间应用双等号，比较的是他么在内存中的存放地址  **方法正常调用完成/方法返回地址**  返回一个值给调用它的方法，方法正常完成发生在一个方法执行过程中遇到了方法返回的字节码指令的时候，使用哪种返回指令取决于方法返回值的类型（如果有返回值的话）  Jvm虚拟机根据不同数据类型有不同的底层return指令，当被调用方法执行某条return指令时，会选择相应的return指令来让值返回（如果该方法有返回值的话）  在这种情况下，当前栈帧就被用来恢复调用者的状态，恢复局部变量表，操作数栈，和程序计数器（pc指针）,而这个程序计数器要适当的增加，来指向下一条指令（也就是调用函数的下一句），使调用者方法能够正常的继续执行下去，而且返回值push到了调用方法的操作数栈中  **方法异常调用完成**  异常时不会返回值给调用者  所以return和抛出异常会导致方法结束，也就是栈帧数据出栈 |

-----end

* （续）java虚拟机栈

经常有人把java内存区分为堆内存（Heap）和栈内存（Stack），其实这是比较粗糙的分法，java内存区域的划分实际上远比这复杂，其中所指的“堆”会在之后详说，而所指的“栈”就是虚拟机栈，或者说是虚拟机栈中的局部变量表部分

局部变量表存放了编译期可知的各种基本数据类型（boolean,byte,char,short,int,float,long,double）,对象引用（reference类型，它不等同于对象本身，根据不同的虚拟机实现，它可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或者其他与此对象相关的位置）和returnAddress类型（指向了一条字节码指令的地址）

其中64位长度的long和double类型的数据会占用2个局部变量空间（Slot），其余的数据类型只会占用1个。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在栈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

在java虚拟机规范中，对虚拟机栈区域规定了两种异常状况：

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. 如果虚拟机栈可以动态扩展（当前大部分的java虚拟机都可以动态扩展，只不过java虚拟机规范中也允许固定长度的虚拟机栈），当扩展时无法申请到足够的内存时会抛出OutOfMemoryError异常

方法在没有被调用前存在于方法区的运行时常量池里，符号引用

被调用了以后存在于虚拟机栈的局部变量表的栈帧里，操作数栈

* 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，区别不过是虚拟机栈为虚拟机执行java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的native方法服务

Native方法参考

<https://www.jianshu.com/p/1eb6d859175d>

和<https://www.cnblogs.com/HDK2016/p/7226840.html?utm_source=itdadao&utm_medium=referral>

虚拟机规范中对本地方法中的方法使用语言，使用方式，与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它，甚至有的虚拟机（比如Sun HotSpot虚拟机）直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区也会抛出StackOverflowError异常和OutOfMemoryError异常

* Java堆

对于大多数应用来说，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内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里被分配内存。

Java虚拟机规范中的相关描述为：所有的对象实例以及数组都要在堆上分配，但是随着JIT编译器的发展与逃逸分析技术的逐渐成熟，栈上分配，标量替换优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也渐渐变得不是那么的绝对了

Java堆是垃圾回收收集器的管理的主要区域，因此很多时候也被称作“GC堆”，如果从内存回收的角度来看，由于现在的收集器基本都是采用的分代收集算法，所以java堆中还可以细分为新生代，老年代再细致一点的有Eden空间，From Survivor空间，To Survivor空间等。如果从内存分配的角度看，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（TLAB）。无论如何划分，都与存放内容无关，无论哪个区域，存储的都仍然是对象实例，进一步划分的目的是为了更好的回收内存，或者更快地分配内存。

根据Java虚拟机规范的规定，java堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像磁盘空间一样，在实现时，既可以实现成固定大小的，也可以是扩展的额，不过当前主流的虚拟机都是暗战可扩展来实现的（通过-Xmx和-Xms控制）。如果堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常

* 方法区

方法区域java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息，常量，静态变量，及时编译器编译后的代码等数据。

对于习惯在HotSpot虚拟机上开发和部署程序的开发者来说，很多人愿意把方法去称为“永久代”，本质上两者并不等价，仅仅是因为HotSpot虚拟机的设计团队选择把GC分代收集扩展至方法权益，或者使用永久代来实现方法区，对于其他虚拟机（如BEA JRockit，IBM J9等）来说不存在永久代的概念

Java虚拟机规范对于这个区域的限制非常宽松，除了和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。相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入方法区就如永久代的名字一样“永久”存在了。

这个区域的内存回收目标主要是针对常量池的回收和堆类型的卸载

根据java虚拟机的规范，当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常

* 运行时常量池，见上
* 直接内存

直接内存并不是虚拟机运行时数据区的一部分，也不是java虚拟机规范中定义的内存区域，但是这部分内存被频繁使用，而且也可能导致OutOfMemoryError异常出现。

在JDK1.4中新加入了NIO（New Input/Output）类，引入了一种基于通道与缓冲区的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆里面的DirectByteBuffer对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在java堆和Native堆中来回复制数据

显然，本机的直接内存的分配并不会收到Java堆大小的限制，但是，既然是内存，则肯定还是会受到本机总内存（包括RAM及SWAP区或者分页文件）的大小及处理器寻址空间的限制，服务器管理员配置虚拟机参数时，一般会根据实际内存设置-Xmx等参数信息，但经常会忽略掉直接内存，使得各个内存区域的总和大于物理内存限制（包括物理上的和操作系统级的限制），从而导致动态扩展时出现OutOfMemoryError异常

* 对象访问

对象访问在java语言中无处不在，是最普通的程序行为，当即使是最简单的访问，也会涉及java栈，java堆，方法区这三个最重要的内存区域之间的关联联系

例如：

Object obj=new Object();

假设这句代码出现在方法体中，那么“Object obj”这部分的语义将会反应到java栈的本地变量表中，作为一个reference类型数据出现。而“new Object()”这部分语义将会反映到java堆中，形成一块存储了Object类型所有实例数据值（Instance Data，对象中各个实例字段的数据）的结构化内存，根据具体类型以及虚拟机实现的对象内存布局的不同，这块内存的长度是不固定的。另外，在java堆中，还必须包含能查找到此对象类型数据（如，对象类型，父类，实现的接口，方法等）的地址信息，这些类型数据则存储在方法区中

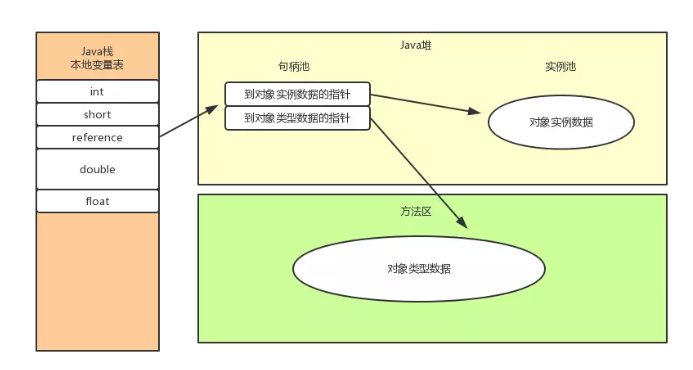
所以

方法在没有被调用前存在于方法区的运行时常量池里，符号引用

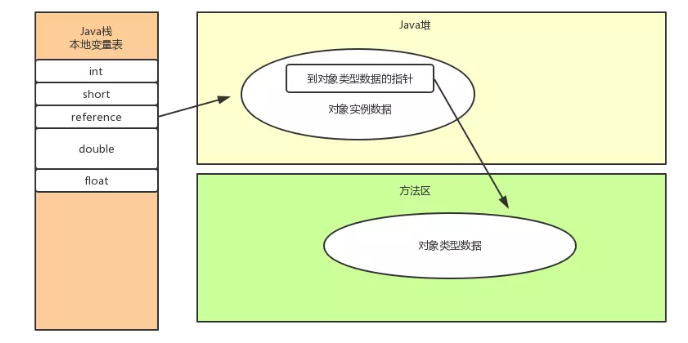
被调用了以后存在于虚拟机栈的局部变量表的栈帧里，操作数栈

主流的定位访问方式有： 句柄访问和直接指针访问

1. 句柄访问方式，java堆中将会划分出一块内存作为句柄池，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和类型数据各自的具体地址信息



1. 如果使用直接指针访问方式，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，reference中直接存储的就是对象地址



这两种对象的访问方式各有优势，使用句柄访问方式的最大好处就是reference中存储的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而reference本身不需要更改

使用直接指针访问方式的最大好处就是速度更快，它节省了每一次指针定位的时间开销，由于对象的访问在java中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本

就Sun HotSpot而言，它是使用第二种方式进行对象访问的