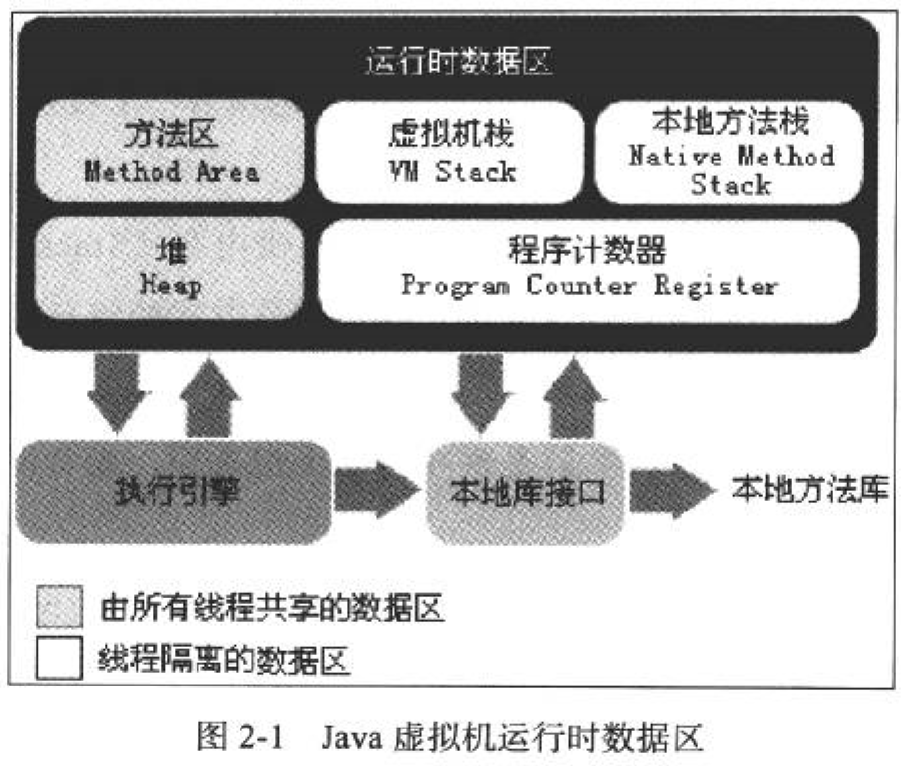
**虚拟机内存**

参考：<https://blog.csdn.net/qian520ao/article/category/7378094>

问题[StackOverflowError的产生](https://www.cnblogs.com/stefanking/articles/5434043.html)？

Java虚拟机将把它所管理的内存分为多个数据区域。有的随着进程生而生，死而死，有的随着线程（线程私有内存）生而生，死而死。

* **运行是数据区域**

****

* **程序计数器**

程序计数器是一块较小的内存区域，它可以看作是当前线程执行字节码的行号指示器。字节码解释器工作时通过这个改变程序计数器的值来选取执行字节码的下一条指令。

每个线程都有一个独立的程序计数器，各个线程之间的程序计数器互不影响，独立存储，所以它属于线程私有内存。

如果这个线程执行的是一个Java方法，则这个计数器记录的是执行字节码指令的地址；如果执行的是一个Native方法则这个计数器值为空。

此区域是唯一一个虚拟机规范中没有任何OutOfMemoryError情况的区域。

* **虚拟机栈**

虚拟机栈也是线程私有的，它的生命周期与线程相同。

虚拟机栈描述的是执行Java方法的内存模型:当线程执行一个方法的同时，会在虚拟机栈中创建一个**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。每一个方法从调用到执行完的过程就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

注意：**栈帧**这个概念，栈帧是一个数据结构存储上面说的内容，他是虚拟机栈的元素。一个虚拟机栈中存在多个栈帧，因为当一个线程执行一个方法的时候会产生一个栈帧，此时这个栈帧位于栈顶，当这个方法中调用了另一个方法的时候，又会在虚拟机栈中产生一个栈帧，此时新产生的这个栈帧会位于栈顶。只有位于栈顶的栈帧才会有效，称为“当前栈帧”，与这个栈帧相关联的方法称为“当前方法”。

在虚拟机栈中我们最需要关注的就是由栈帧存储的局部变量表：所有的基本数据类型（包括值）、引用型、returnAddress（指向了一条字节码指令地址）。局部变量表在编译期间就已经确定好所占空间大小，在运行期间不会改变局部变量表所占空间的大小。

对于这块区域虚拟机规范给出了两个异常情况：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;如果虚拟机可以动态扩展，但是在扩展的时候没有申请到足够的空间，将抛出OutOfMemoryError异常。

* **本地方法栈**

本地方法栈与虚拟机栈的作用一致，只不过本地方法栈服务于Native方法，而虚拟机栈服务于Java方法。

本地方法栈也属于线程私有内存，生命周期与线程一样。

与虚拟机栈一样他也会StackOverflowError异常和抛出OutOfMemoryError异常。

* **堆**

Java堆是虚拟机管理的最大一块内存内存区域，也是被所有线程共享的一块内存区域，它随着虚拟机的创建而创建。堆内存中存储了几乎所有的实例对象。

Java堆是垃圾收集器管理的主要区域。从内存回收的角度看，由于收集器基本都采用分带收集算法，所以Java堆可分为：新生代、老生代；从内存分配的角度看，Java对可划分出多个线程私有的分配缓冲区（TLAB）。无论怎么划分都与Java存放内容无关，无论哪个区域Java堆存放的永远是对象和数组。

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。在Java堆中若没有足够的内存完成实例分配，并且也不能在扩展Java堆时，就会抛出OutOfMemoryError异常。

* **方法区**

方法区用来存储已被虚拟机加载的类的信息、静态变量、常量、即时编译器编译后的代码等数据。它也是各个线程共享的区域，是程序中永远唯一的元素存储区。

和堆一样，方法区可以选择固定大小，也可以选择拓展大小，还可以选择不实现垃圾回收。

垃圾回收在方法区是比较少见的，并不是说在方法区的数据是“永久”存在的。方法区的垃圾回收主要是针对类型的卸载和常量池的回收。当方法区无法满足内存分配需求时将会抛出OutOfMemoryError异常。

* **运行时常量池**

**运行时常量池**是方法区的一部分。Class常量池中的内容将在类加载后进入方法区的运行时常量池中，除了保存Class文件中的符号引用外，也会把翻译出来的直接引用也存储在运行时常量池中。运行时常量池是方法区的一部分，也受到方法区的内存限制，当常量池无法在申请到内存时会抛出OutOfMemoryError异常。

**Class常量池：**Class文件除了有类的版本、字段、方法、接口等描述字段外，还有一项信息是常量池，用于存储编译器生成的各种字面量和符号引用。

字面量：表示一个固定的、不可变的值，包括int、double.boolean、char、String、null、Integer。值得注意的是，字面量指的是一个值，比如“小明”、“aa”、4.5、1等等。不管这些值出现在方法内，或者是类的成员变量的值，这些字面量都会进入Class常量池，并且进入到运行时常量池。

符号引用：属于编译原理方面的概念，包含三类常量：类和接口的全限定名、字段的名称和描述符、方法的名称和描述符

运行时常量池相对于Class常量池另一个特性就是具备动态性。Java语言并不要求常量只有编译期才能产生，也就是并非预置到Class常量池的内容才能进入到运行时常量池中，运行期间也可以将新的常量放入到运行时常量池中，用的比较多的是String类的intern()方法。

* **对象创建**

这里讨论的对象创建是指普通的对象创建，不包括数组和Class对象等。在Java语言层面创建一个对象只需要一个new关键字，但是在虚拟机中必须经过如下步骤：

**1、类加载检查：**当虚拟机遇到一个new指令时首先要检查这个指令的参数能否在当前类的Class常量池中定位到这个类的符号引用。并且检查这个符号引用代表的类是否被加载、解析、初始化过，如果没有，则必须先加载这个类。

**2、分配内存：**经过类加载检查后，虚拟机需要为新对象分配内存。对象所需内存的大小在类加载后就可以完全确定。为对象分配内存就等于把一块确定好大小的内存从Java堆里划分出来，其中包括两种分配方式：

**指针碰撞：**假设Java堆中的内存是绝对工整的，即所有使用的内存在一边，空闲的内存在另一边，中间是一个分界指针，那么为对象分配内存就只是将分界指针向空闲的一边移动与新对象大小相等的距离即可，这种分配方式叫指针碰撞。

**空闲列表：**假如Java堆内存不是绝对规整的，使用的内存和空闲内存相互交错，这时候就不能用指针碰撞的方式来分配内存，此时需要虚拟机维护一个列表来记录那些空闲内存，在内存分配的时候就从列表中选取一块足够大的内存来创建对象，并更新列表上的记录。

虚拟机给对象分配内存的方式由Java堆是否规整来决定，Java堆是否规整又由虚拟机的垃圾收集器是否带有压缩整理功能决定。

**3、初始化内存空间为零值：**内存分配完成后，虚拟机需要将分配的对象内存空间初始化为零值（不包括对象头）。这一步保证了对象的实例字段（成员变量）在Java代码中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。

**4、对象设置：**上一步不会设置对象头，而接下来虚拟机要对这个对象进行必要的设置，其中就包括对象头信息：这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的GC分代年龄等信息，这些信息都是存放在对象头中的。

**5、实例构造器的init方法：**上面步骤执行完后，即虚拟机执行完new指令后，会继续执行<init>方法（包括成员变量赋值、普通语句块执行、构造函数执行等）。这样一个真正可用的对象才算完全生产出来。

创建对象分配Java堆中的内存并不是线程安全的，可能出现正在给对象A分配内存，指针还没来得及修改，这时候另一个线程创建对象B使用原来的指针来分配内存的情况，解决这种并发问题有两种解决方案：

**同步处理**：虚拟机对内存分配动作进行同步处理，即采用CAS配上失败重试的方式来保证原子性操作。

**本地线程分配缓冲（TLAB）：**每个线程在Java堆中预先分配一块内存，称为本地线程分配缓冲（TLAB），哪个线程需要分配内存就在相应线程的TLAB上进行分配的动作。当相应线程的TLAB空间使用完了之后，虚拟机会为这个线程分配新的TLAB，这时候才需要进行同步锁定操作。这样在创建对象时，就把虚拟机分配内存的动作按照线程划分到Java堆不同的区域，而保证了线程安全。

* **对象的内存布局**

在HotSpot虚拟机中，一个实例对象在Java堆中的布局分为3部分：对象头、实例数据、对齐填充。

* **对象头**

对象头包括两部分信息：运行时数据和类型指针。

**运行时数据：**如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有锁、偏向线程ID、偏向时间戳等。这部分数据的长度在32位和64位的虚拟机中分别为32bit和64bit，官方称为“Mark Word”。

**类型指针：**即对象指向它的类元数据的指针，虚拟机通过这个指针可以知道这个对象是哪个类的实例。（但是并不是所有的虚拟机实现都要在对象的头部存储类型指针，话句话说，并不是只能通过对象才能查找到类型的元数据信息）。

**例外：**如果对象是个数组，那么在头部还需要一块空间来记录数组的长度，因为通过数组的元数据无法确定具体某个数组的大小。但是，普通的对象的类元数据可以确定某个实例的大小。

* **实例数据**

实例数据中存储的是对象的真正有效信息，也就是对象的各种类型的字段内容。无论是父类继承下来的还是，子类中定义的都要记录下来。

* **对齐填充**

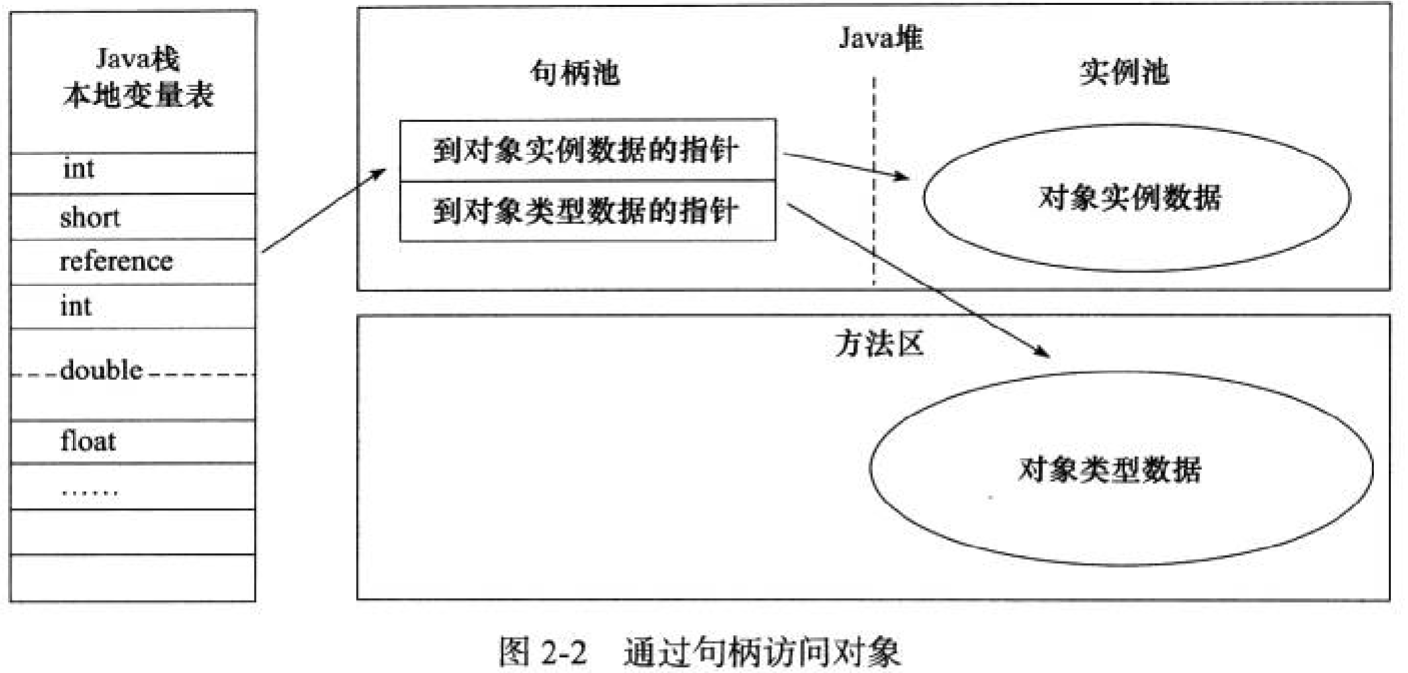
最后一部分对齐填充不是必然存在的，也没有特殊的含义，它只起到一个占位符的作用。由于虚拟机内存管理系统要求对象的起始地址必须是8字节的整数倍，话句话说就是，对象的大小必须是8字节的整数倍。而对象的头部正好是8字节的整数倍（1倍或者2倍），当对象的实例数据部分不是8字节的整数倍，就需要多分配一些内存来对齐填充。

* **对象的访问定位**

创建对象是为了使用对象，在虚拟机规范中线程本地栈中的引用（reference）类型变量指向一个对象，但是他没有规定这个引用以何种方式去访问和定位堆中的对象的具体位置，所以具体的访问方式取决于虚拟机的实现。目前主流的访问方式有两种：句柄访问和直接指针。

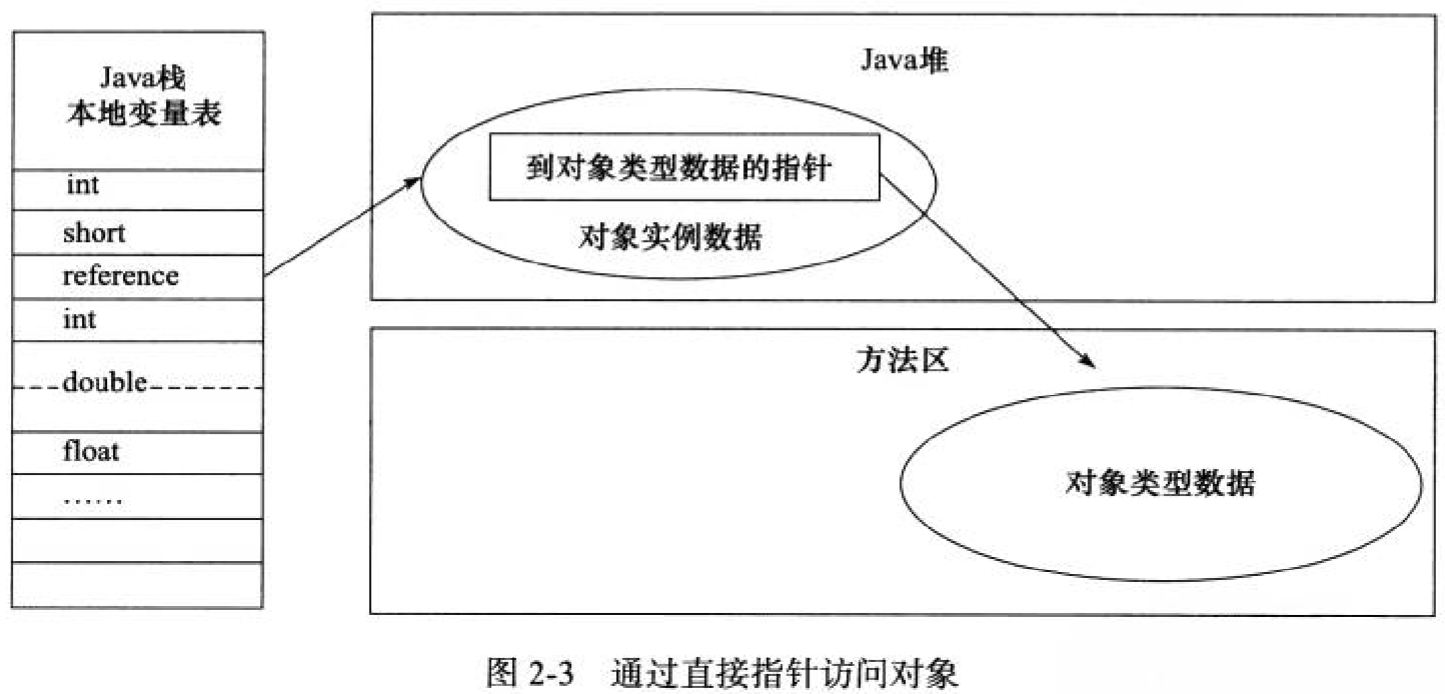
* **句柄访问**

如果是句柄访问，那么Java堆会划分出一块区域作为句柄池，线程本地栈中的引用存储的就是句柄的地址，句柄中又存储了具体事例对象的地址和类型数据的地址。（这样的实例对象头部可以不存储类型指针）



* **直接访问**

直接访问的话，则线程栈中的引用直接存储对象的地址，然后对象的头部存储类型指针，然后通过类型指针找到类型数据。



**优缺点：**

**句柄访问：**使用句柄访问的好处就是线程栈中的引用是稳定的，当Java堆中对象被移动的时候（当垃圾收集时对象被移动是非常普遍的）只需要修改句柄中对象的指针，而线程栈中的引用不需要修改。

**直接引用**：直接引用方式访问对象只需要一次指针定位，不但访问速度更快，还节省了对象访问的开销。由于在Java中访问对象是一个非常频繁的事情，因此频繁的访问对象也是一项可观的开销。