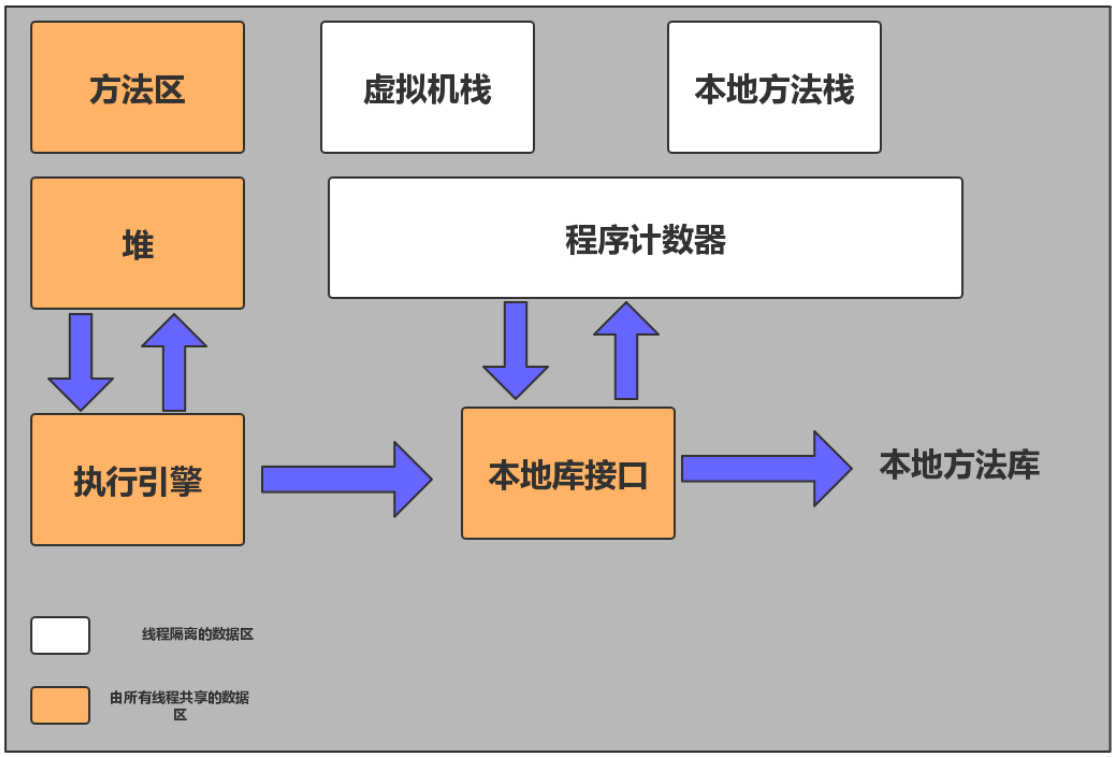
JAVA内存区域

JAVA虚拟机运行时数据区



- 从上图可以看出，这里其实主要就是上面的四大部分，包括堆、方法区、栈和程序计数器
- 其中还分为线程是否共享的区域，另外注意这里栈有些语言可能还会把栈分为虚拟机栈和本地方法栈，但一般来说其实两者功能是类似的

1.程序计数器

- 程序计数器是一块较小的内存空间，可以看作是当前线程执行的字节码的行号指示器.一般每个处理器在每个时刻只能执行一条线程中的指令，因此为了 线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器
-
- **此区域是唯一一个在JAVA内存规范中没有规定任何OutOfMemoryError情况的区域(OOM,也就是内存溢出)**

2.Java虚拟机栈与本地方法栈

- Java虚拟机栈和本地方法栈都是线程私有的，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。每一个方法从调用直到执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

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，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）。

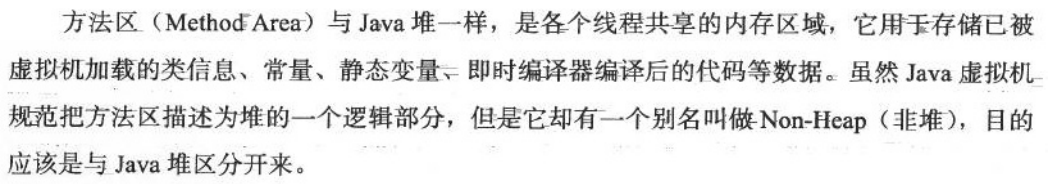
本地方法栈（Native Method Stack）与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。在虚拟机规范中对本地方法栈中方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如 Sun HotSpot 虚拟机）直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出 StackOverflowError 和 OutOfMemoryError 异常。

3.Java堆

Java堆是被线程共享的一块区域

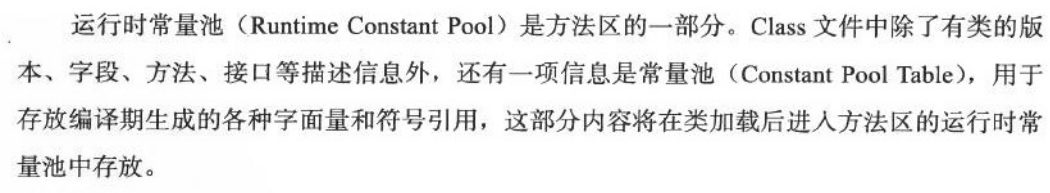
- Java堆主要存放的是对象实例，所以这里也是垃圾回收机制管理的主要区域，现在GC基本都采用分代收集算法(也就是把Java堆分代，按照对象的生存周期分为新生代、老年代等区域，其中新生代细分可以分为Eden、From Suvivor和To Suvivor空间)
- Java堆虽然是线程共享的，但是从内存分配的角度来看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区(TLAB)。另外这里也会有OOM的出现。

4.方法区



- 注意，虽然这块区域在线程结束的时候也会跟着销毁，但是并不代表这里的内容不会进行GC,这块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。
- 很多书会把方法区称为"永久代"，这种说法其实是不准确的，为了突破可用内存的上限，Java8中将原本由Jvm管理内存的方法区移到了虚拟机之外的计算机本地，并将此称为元空间，也就是不再受虚拟机大小限制了。

运行时常量池



- **常量池: 存放字面量和符号引号**
- 符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能够无歧义的定位到目标即可.
- 例如，在Class文件中它以CONSTANT_Class_info、CONSTANT_Fieldref_info、CONSTANT_Methodref_info等类型的常量出现。符号引用与虚拟机的内存布局无关，引用的目标并不一定加载到内存中。在Java中，一个java类将会编译成一个class文件。在编译时，java类并不知道所引用的类的实际地址，因此只能使用符号引用来代替。比如org.simple.People类引用了org.simple.Language类，在编译时People类并不知道Language类的实际内存地址，因此只能使用符号org.simple.Language（假设是这个，当然实际中是由类似于CONSTANT_Class_info的常量来表示的）来表示Language类的地址。各种虚拟机实现的内存布局可能有所不同，但是它们能接受的符号引用都是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中

```
   在java代码中，一个类可能使用另外类或者接口的字段或者调用另外一个类的方法。
在编译的时候，class文件中是通过叫做"符号引用"的方式来实现的。
如下面的例子
public interface Intf {
    public static String str = "abcde";
    public static int ival = new Random().nextInt();

}
public class T {
    public static int tint = Intf.ival;
}
class T编译成class 文件之后，利用javap -verbose 查看class文件，可以看到
在初始化static int tint的时候，从常量池的第12项取值
static {};
  Code:
   Stack=1, Locals=0, Args_size=0
   0:    getstatic    #12; //Field Intf.ival:I
   3:    putstatic    #17; //Field tint:I
   6:    return
  LineNumberTable:
   line 6: 0
   line 3: 6
常量池的第12项是一个符号引用，指向Intf的ival字段。ival的值只能在运行的时候才能确定。
const #12 = Field    #13.#15;    //  Intf.ival:I
const #13 = class    #14;    //  Intf
在运行的时候会将符号引用解析为指向Intf内存地址的直接引用。

如果将class T改为如下，引用的是Intf的一个在编译时候就可以确定的常量值。
通过查看编译后的class文件，发现在T的class文件中，是将str = "abcde";的值
"abcde"直接复制了一份放到了常量池中，没有符号引用。
public class T {
```

```
    public static String tstr = Intf.str;
}
在初始化static int tint的时候，从常量池的第12项取值
static {};
  Code:
   Stack=1, Locals=0, Args_size=0
   0:    ldc    #12; //String abcde
   2:    putstatic    #14; //Field tstr:Ljava/lang/String;
   5:    return
  LineNumberTable:
   line 5: 0
   line 3: 5
常量池的第12项直接为"abcde"字符串。
const #12 = String    #13;    //  abcde
const #13 = Asciz    abcde;
----------------------
作者：IT农夫
来源：CSDN
原文：https://blog.csdn.net/kkdelta/article/details/17752097
版权声明：本文为博主原创文章，转载请附上博文链接！
```

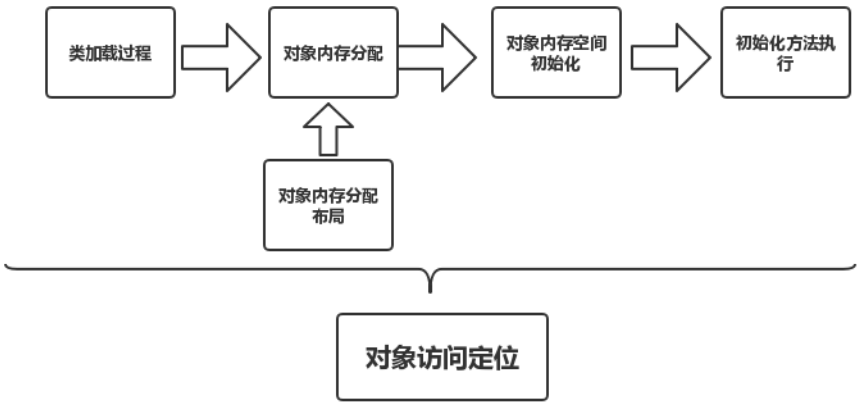
5.直接内存

直接内存（Direct Memory）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。但是这部分内存也被频繁地使用，而且也可能导致 OutOfMemoryError 异常出现，所以我们放到这里一起讲解。

在 JDK 1.4 中新加入了 NIO（New Input/Output）类，引入了一种基于通道（Channel）与缓冲区（Buffer）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 堆中的 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，肯定还是会受到本机总内存（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。服务器管理员在配置虚拟机参数时，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（包括物理的和操作系统级的限制），从而导致动态扩展时出现 OutOfMemoryError 异常。

Java对象创建过程



1.类加载过程

Java 是一门面向对象的编程语言，在 Java 程序运行过程中无时无刻都有对象被创建出来。在语言层面上，创建对象（例如克隆、反序列化）通常仅仅是一个 new 关键字而已，而在虚拟机中，对象（文中讨论的对象限于普通 Java 对象，不包括数组和 Class 对象等）的创建又是怎样一个过程呢？

虚拟机遇到一条 new 指令时，首先将去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，并且检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没有，那必须先执行相应的类加载过程，本书第 7 章将探讨这部分内容的细节。

2.对象内存分配

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。对象所需内存的大小在类加载完成后便可完全确定（如何确定将在 2.3.2 节中介绍），为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。假设 Java 堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，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空间那边挪动一段与对象大小相等的距离，这种分配方式称为“指针碰撞”（Bump the Pointer）。如果 Java 堆中的内存并不是规整的，已使用的内存和空闲的内存相互交错，那就没有办法简单地进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，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，这种分配方式称为“空闲列表”（Free List）。选择哪种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。因此，在使用 Serial、ParNew 等带 Compact 过程的收集器时，系统采用的分配算法是指针碰撞，而使用 CMS 这种基于 Mark-Sweep 算法的收集器时，通常采用空闲列表。

除如何划分可用空间之外，还有另外一个需要考虑的问题是对象创建在虚拟机中是非常频繁的行为，即使是仅仅修改一个指针所指向的位置，在并发情况下也并不是线程安全的，可能出现正在给对象 A 分配内存，指针还没来得及修改，对象 B 又同时使用了原来的指针来分配内存的情况。解决这个问题有两种方案，一种是对分配内存空间的动作进行同步处理——实际上虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性；另一种是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（Thread Local Allocation Buffer，TLAB）。哪个线程要分配内存，就在哪个线程的 TLAB 上分配，只有 TLAB 用完并分配新的 TLAB 时，才需要同步锁定。虚拟机是否使用 TLAB，可以通过 -XX:+/-UseTLAB 参数来设定。

3.对象内存初始化

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），如果使用 TLAB，这一工作过程也可以提前至 TLAB 分配时进行。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，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、对象的哈希码、对象的 GC 分代年龄等信息。这些信息存放在对象的对象头（Object Header）之中。根据虚拟机当前的运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。关于对象头的具体内容，稍后再做详细介绍。

4.对象初始化方法执行

在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚刚开始——<init> 方法还没有执行，所有的字段都还为零。所以，一般来说（由字节码中是否跟随 invokespecial 指令所决定），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，这样一个真正可用的对象才算完全产生出来。

5.对象的内存布局

在 HotSpot 虚拟机中，对象在内存中存储的布局可以分为 3 块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

HotSpot 虚拟机的对象头包括两部分信息，第一部分用于存储对象自身的运行时数据，如哈希码（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，这部分数据的长度在 32 位和 64 位的虚拟机（未开启压缩指针）中分别为 32bit 和 64bit，官方称它为“Mark Word”。对象需要存储的运行时数据很多，其实已经超出了 32 位、64 位 Bitmap 结构所能记录的限度，但是对象头信息是与对象自身定义的数据无关的额外存储成本，考虑到虚拟机的空间效率，Mark Word 被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息，它会根据对象的状态复用自己的存储空间。例如，在 32 位的 HotSpot 虚拟机中，如果对象处于未被锁定的状态下，那么 Mark Word 的 32bit 空间中的 25bit 用于存储对象哈希码，4bit 用于存储对象分代年龄，2bit 用于存储锁标志位，1bit 固定为 0，而在其他状态（轻量级锁定、重量级锁定、GC 标记、可偏向）下对象的存储内容见表 2-1。

表 2-1　HotSpot 虚拟机对象头 Mark Word

存储内容	标 志 位	状　　态
对象哈希码、对象分代年龄	01	未锁定
指向锁记录的指针	00	轻量级锁定
指向重量级锁的指针	10	膨胀（重量级锁定）
空，不需要记录信息	11	GC 标记
偏向线程 ID、偏向时间戳、对象分代年龄	01	可偏向

对象头的另外一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。并不是所有的虚拟机实现都必须在对象数据上保留类型指针，换句话说，查找对象的元数据信息并不一定要经过对象本身，这点将在 2.3.3 节讨论。另外，如果对象是一个 Java 数组，那在对象头中还必须有一块用于记录数组长度的数据，因为虚拟机可以通过普通 Java 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却无法确定数组的大小。

6.对象的访问定位

建立对象是为了使用对象，我们的 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 reference 类型在 Java 虚拟机规范中只规定了一个指向对象的引用，并没有定义这个引用应该通过何种方式去定位、访问堆中的对象的具体位置，所以对象访问方式也是取决于虚拟机实现而定的。目前主流的访问方式有使用句柄和直接指针两种。

- 如果使用句柄访问的话，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与类型数据各自的具体

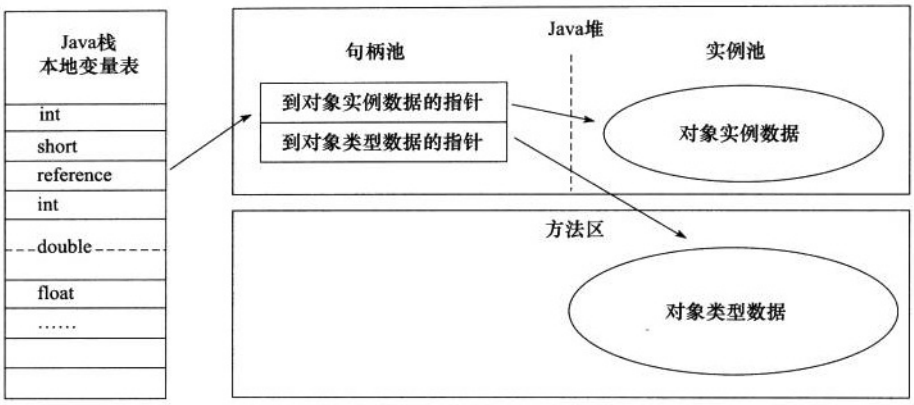


图 2-2　通过句柄访问对象

❑ 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。

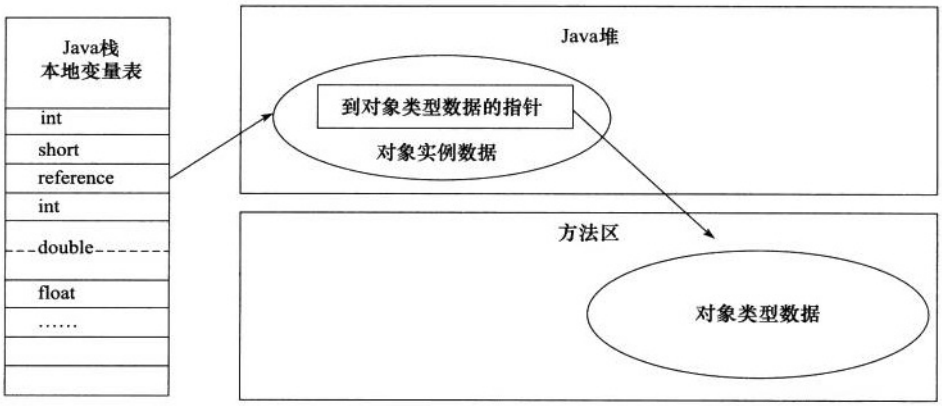


图 2-3　通过直接指针访问对象

这两种对象访问方式各有优势，使用句柄来访问的最大好处就是 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 本身不需要修改。

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。就本书讨论的主要虚拟机 Sun HotSpot 而言，它是使用第二种方式进行对象访问的，但从整个软件开发的范围来看，各种语言和框架使用句柄来访问的情况也十分常见。