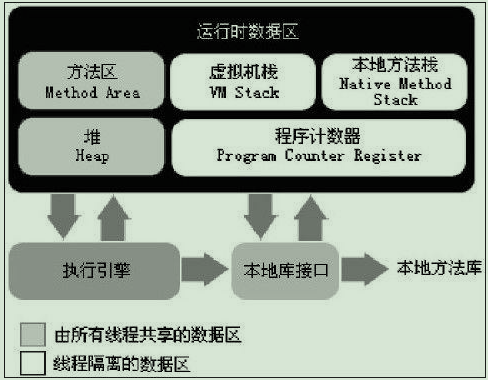
**Java技术体系包括**：

1. Java程序设计语言
2. 各种硬件平台上的Java虚拟机
3. Class文件格式
4. Java API类库
5. 第三方Java类库

JDK包括1，2，4

JRE包括Java SE API和JVM

**自动内存管理机制：**



Java虚拟机管理的内存包括以上几个运行时数据区

**程序计数器**：

线程私有。是一块较小的内存空间，它可以看作是当前线程所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，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。每条线程都有一个独立的程序计数器，各条线程之间计数器互不影响，独立存储。

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（Undefined）。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。

**JVM栈：**

线程私有。生命周期与线程相同。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（Stack Frame）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至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

在Java虚拟机规范中，对这个区域规定了两种异常状况：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；如果虚拟机栈可以动态扩展（当前大部分的Java虚拟机都可动态扩展，只不过Java虚拟机规范中也允许固定长度的虚拟机栈），如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

**本地方法栈**：

线程私有。JVM栈为虚拟机执行Java方法（也就是字节码）服务，本地方法栈则为虚拟机使用到的Native方法服务。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。

**堆：**

所有线程共享。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和数组都在这里分配内存。堆是垃圾收集器管理的主要区域，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。在实现时，既可以是固定大小的，也可以是可扩展的，不过当前主流的虚拟机都是按照可扩展来实现的（通过-Xmx和-Xms控制）。如果在堆中没有内存可供实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常。

**方法区：**

所有线程共享。用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。不需要连续的内存，可以选择固定大小或者可扩展，还可以选择不实现垃圾收集。这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。根据Java虚拟机规范的规定，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：**

是方法区的一部分，受到方法区内存的限制。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。运行时常量池有动态性，可以在运行期间将新的常量放入池中，例如String类的intern（）方法。当常量池无法再申

请到内存时会抛出OutOfMemoryError异常。

**直接内存：**

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。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。当无法满足内存分配需求时，将抛出OutOfMemoryError异常。

**HotSpot虚拟机在Java堆中对象分配、布局和访问的全过程：**

**对象的创建：**

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。

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，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。如果java堆中内存是绝对规整的，采用“指针碰撞”（Bump the Pointer）分配方式。如果不规整，采用“空闲列表”（Free List）分配方式。

分配内存的线程安全性考虑。一种是对分配内存空间的动作进行同步处理，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**）。虚拟机是否使用TLAB，可以通过-XX：+/-UseTLAB参数来设定。

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）。对象头存放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。

执行new指令之后接着执行＜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。

**对象的内存布局：**

对象在内存中存储的布局分为3块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

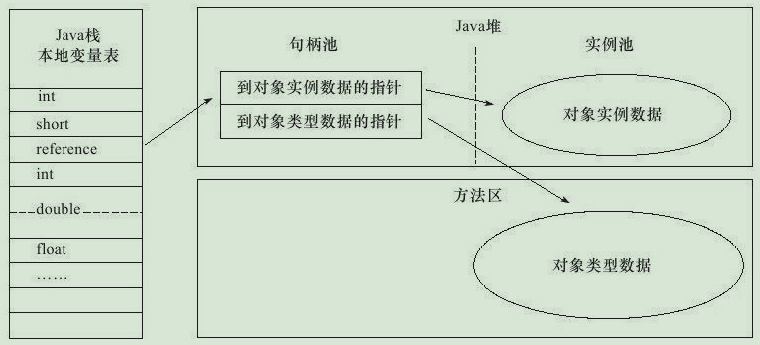
对象头包括两部分信息，第一部分用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等。对象头的另外一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。如果对象是一个Java数组，那在对象头中还必须有一块用于记录数组长度的数据。

实例数据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，还是在子类中定义的，都需要记录起来。这部分的存储顺序会受到虚拟机分配策略（FieldsAllocationStyle）和字段在Java源码中定义顺序的影响。

对齐填充并不是必然存在的，也没有特别的含义，它仅仅起着占位符的作用。由于HotSpot VM的自动内存管理系统要求对象起始地址必须是8字节的整数倍，换句话说，就是对象的大小必须是8字节的整数倍，当对象实例数据部分没有对齐时，就需要通过对齐填充来补全。

**对象的访问：**

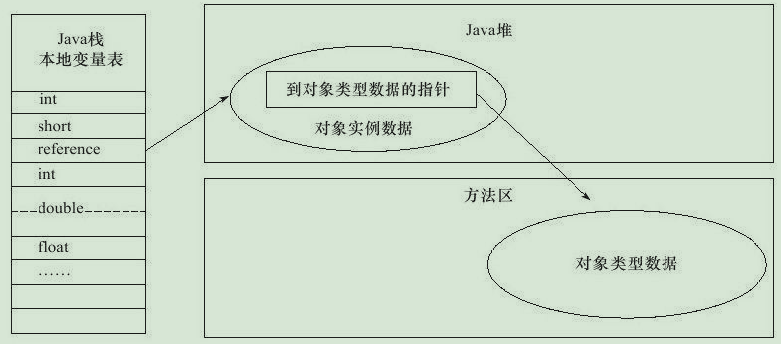
目前主流的访问方式有使用句柄和直接指针两种。



通过句柄访问对象，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地址信息。这种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本身不需要修改。



通过指针访问对象，reference中存储的直接就是对象地址，这种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。Sun HotSpot采用这种方式访问对象。