jvm运行时数据区域

VM主要由类加载器子系统、运行时数据区（内存空间）、执行引擎以及与本地方法接口等组成。其中运行时数据区又由**方法区**、**堆**、**Java**
组成。其中**本地方法栈**与**直接内存**由操作系统提供，jvm负责调用

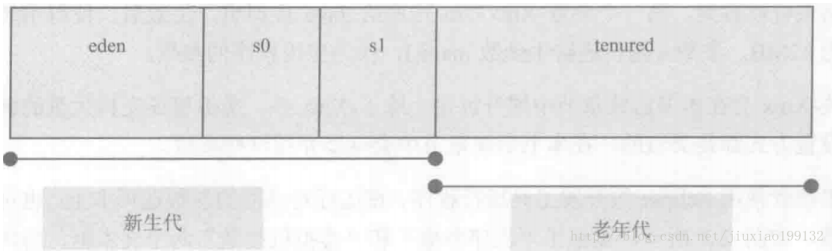
PC寄存器

PC寄存器又称作程序计数器，每一个线程都是私有的。用于保存当前线程正在执行的位置，由于Java是支持多线程执行的，所以程序执行的
多个线程交叉执行时，被暂停的线程的程序当前执行到的位置必然要保存下来，以便用于被暂停的线程恢复执行时再按照被暂停时的指令地

堆（Heap）

java堆是Java虚拟机所管理的内存中最大的一块。Java堆是所有线程共享的一块内存区域，在虚拟机启动的时候创建。此内存区域的唯一目
都存在这个内存空间中（随着JIT编译器的发展和逃逸分析技术的逐渐成熟，栈上分配，标量替换优化技术的出现，所以并不是绝对的所有

Java堆是自动进行管理的，通过垃圾回收系统自动的清理java堆中的垃圾对象，所以有时候被称为‘GC堆’。
从内存回收的角度来看Java堆的分类可以细分为新生代和老年代。在细致一点新生代分为eden区，s0区，s1区

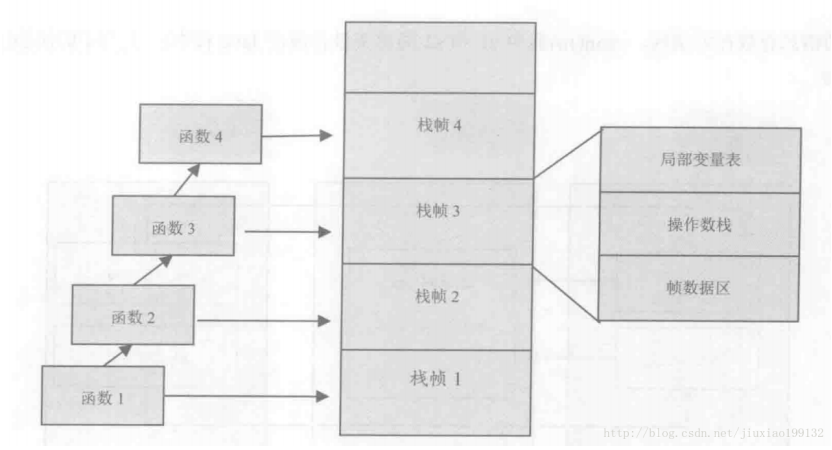


从内存分配的角度来看，线程共享的Java堆中可能划分出来多个线程私有的分配缓冲区。不过无论如何划分，无论哪个区域，存储的都是对内存，或者更快的分配内存

Java堆可以是物理上不连续的内存空间，只需要逻辑上连续即可。Java堆的分配上是扩展的通过-Xmx和-Xms来设定最大值和最小值来控制候会抛出异常OutOfMemoryError

Java虚拟机栈（Stack）

Java虚拟机栈是线程私有的，它的生命周期与线程相同。它所描述的是Java方法执行的内存模型，每个方法执行的时候都会创建一个栈帧（作栈、动态链接、方法出口等信息。每一个方法被调用直到执行完成的过程，就对应着一个栈帧在Java虚拟机栈中从入栈到出栈的过程



如图：方法1对应栈帧1，方法2,3,4分别对应栈帧2,3,4
方法1执行时，栈帧1入栈
方法1调用方法2时，栈帧2入栈
方法2调用方法3时，栈帧3入栈
方法3调用方法4，栈帧4入栈
方法4执行完毕，栈帧4出栈
方法3执行完毕，栈帧3出栈
方法2执行完毕，栈帧2出栈
方法1执行完毕，栈帧1出栈

Java出栈有两种方式，一种正常执行完毕，另外一种就是抛出异常，但是不管怎么样都会出栈。
Java栈是一块内存区域，是有空间大小限制的。栈帧的局部变量都会申请内存空间，超过了内存的限制就会抛出异常OutOfMemoryError。允许的深度会抛出异常Stack Overflow（-Xss表示来指定线程最大的栈空间）

本地方法栈

本地方法栈与虚拟机栈发挥的作用非常类似，区别在于Java虚拟机栈调用的是Java方法，本地方法栈调用的本地（Native）方法

方法区(永久区)

方法区和堆一样，也是一块共享内存空间，它用于存储已经被Java虚拟机加载的类的信息，常量、静态变量、字段，方法等。方法区的大小
类太多，也会抛出内存溢出异常（OutOfMemoryError）

在HotSpot版本的Java虚拟机上方法区被称为“永久区（Perm）”，是因为HotSpot虚拟机的设计团队把GC分代扩展到了方法区，这块区域
是其他版本的Java虚拟机来说不存在永久区的概念。并且在Hotspot版本的jdk1.8以后由“元数据区”取代了永久区

永久区通过参数-XX:permSize和-XX:MaxPermSize来设定，默认最大值为64M，如果程序中有动态代理来动态的生成类，那么需要指定一
出

元数据区参数通过-XX:MaxMetaspaceSize来指定最大元数据区大小，如果不指定，它会不断扩展，直到耗尽操作系统的内存

直接内存

直接内存并不是虚拟机中的内存区域，而是操作系统所分配的内存区域。jdk1.4中加入了NIO，引入基于管道（Channel）与缓冲区（Buffer
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。这样能够提高性能，因为避免

对象访问

在Java语言中，对象访问无处不在。最普通的也是最简单的访问，也要涉及到Java栈，Java堆，方法区三个最重要的内存区域

```
1 Object obj = new Object();
```

obj 是一个引用（Reference）存储在Java栈
new Object() 是一个实例，存储在Java堆

在Java虚拟机规范中并没有定义这个引用类型（Reference）通过哪种方式定位，以及访问Java堆中对象的具体位置，因此不同的虚拟机实
接指针

句柄访问

Java堆中会划分出来一块内存作为句柄池，Reference中存储的是对象的句柄地址，而句柄中包含了对象的实例数据和类型数据各自的地址

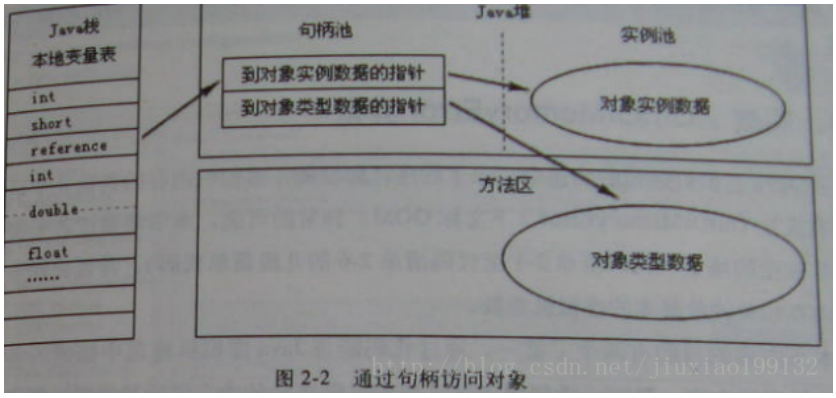


图 2-2 通过句柄访问对象

直接使用指针

Java堆对象的布局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

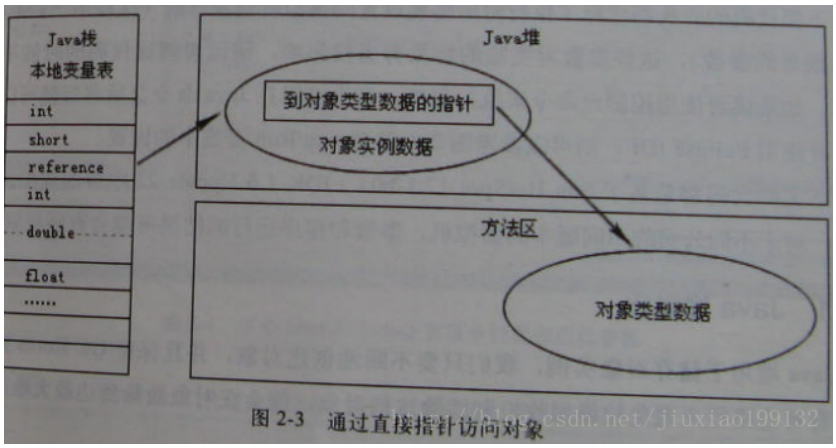


图 2-3 通过直接指针访问对象

使用句柄的方式最大好处就是Reference存储的是稳定的地址，对象被移动（垃圾收集时新生代，老生代移动是非常普遍的行为）时只会改变
Reference本身不需要更改。

使用直接指针的方式最大的好处就是速度更快，它节省了一次指针定位的时间开销。

Hotspot使用的是第二种方式

扩展：内存溢出

栈溢出 —— StackOverflowError

```
import org.junit.Test;

public class StackOverflowError {

    private int count = 0;

    public void countAdd(){
        count++;
        countAdd();
    }

    @Test
    public void test(){
        try {
            countAdd();
        } catch (Throwable e) {
            System.out.println("stack 的 深度 为 ： " + count);
            e.printStackTrace();
        }
    }
}
```

默认输出

```
stack 的 深度 为 ： 17703
java.lang.StackOverflowError
    at com.jx.note.jvm.oom.StackOOM.countAdd(StackOOM.java:11)
```

```
    at com.jx.note.jvm.oom.StackOOM.countAdd(StackOOM.java:11)
```

改变栈的空间最大值（这是对每个线程的）

```
-Xss5M
```

改变参数后输出

```
stack 的 深度 为 ： 58439
java.lang.StackOverflowError
```

可以看到如果改变了栈的空间大小，是会递归次数的。

改变栈帧的大小

同样将栈的大小固定为 5M

```
-Xss5M
```

改变函数，多增加一些局部变量

```
public void countAdd() {
    long a1 = 0L;
    long b1 = 0L;
    long c1 = 0L;
    long d1 = 0L;
    long e1 = 0L;
    long f1 = 0L;
    long g1 = 0L;
    long h1 = 0L;
     long i1 = 0L;
     long j1 = 0L;
     long qw = 0L;
     long d = 0L;
     long qqe = 0L;
     long w = 0L;
     long s = 0L;
     long g = 0L;
     count++;
     countAdd();
}
```

最终会发现递归次数减少，这是由于函数的 栈帧变大了。栈的区域大小固定，栈帧变大，递归次数变少

栈帧的组成部分：返回值，参数，局部变量等

栈溢出 —— OutOfMemoryError（略过）

当栈申请的内存空间不够用的时候会发生

堆溢出 - OutOfMemoryError

```
/**
 * -Xms20m -Xmx20m
 */
public class HeapOOM {
```

```
5
6       public static void main(String[] args) {
7           List<Object> list = new LinkedList<Object>();
8           while(true){
9               list.add(new Object());
10          }
11      }
12  }
```

永久区溢出

JDK8中已经完全移除了永久带。这项工作是在这个bug：https://bugs.openjdk.java.net/browse/JDK-6964458推动下完成的。JDK8中，P
除了

在移除了Perm区域之后，JDK 8中使用MetaSpace来替代，这些空间都直接在堆上来进行分配。 在JDK8中，类的元数据存放在native堆中
中给元数据区添加了一些新的参数。

-XX:MetaspaceSize=<NNN> <NNN>是分配给类元数据区（以字节计）的初始大小（初始高水位），超过会导致垃圾收集器卸载类。
高水位的时候，下一个高水位是由垃圾收集器来管理的

-XX:MaxMetaspaceSize=<NNN> <NNN>是分配给类元数据区的最大值（以字节计）。这个参数可以用来限制分配给类元数据区的大小

-XX:MinMetaspaceFreeRatio=<NNN>，<NNN>是一次GC以后，为了避免增加元数据区（高水位）的大小，空闲的类元数据区的容量的

-XX:MaxMetaspaceFreeRatio=<NNN>，<NNN>是一次GC以后，为了避免减少元数据区（高水位）的大小，空闲的类元数据区的容量的

```
1   /**
2    * 设置永久区最大值
3    * -XX:MaxPermSize=7M
4    */
5   public class PermanentOOM {
6
7       public void addToPermanent() {
8           for (int i = 0; i < Integer.MAX_VALUE; i++) {
9               String s = String.valueOf(i).intern();
10          }
11      }
12
13      public static void main(String[] args) {
14          new PermanentOOM().addToPermanent();
15      }
16  }
```