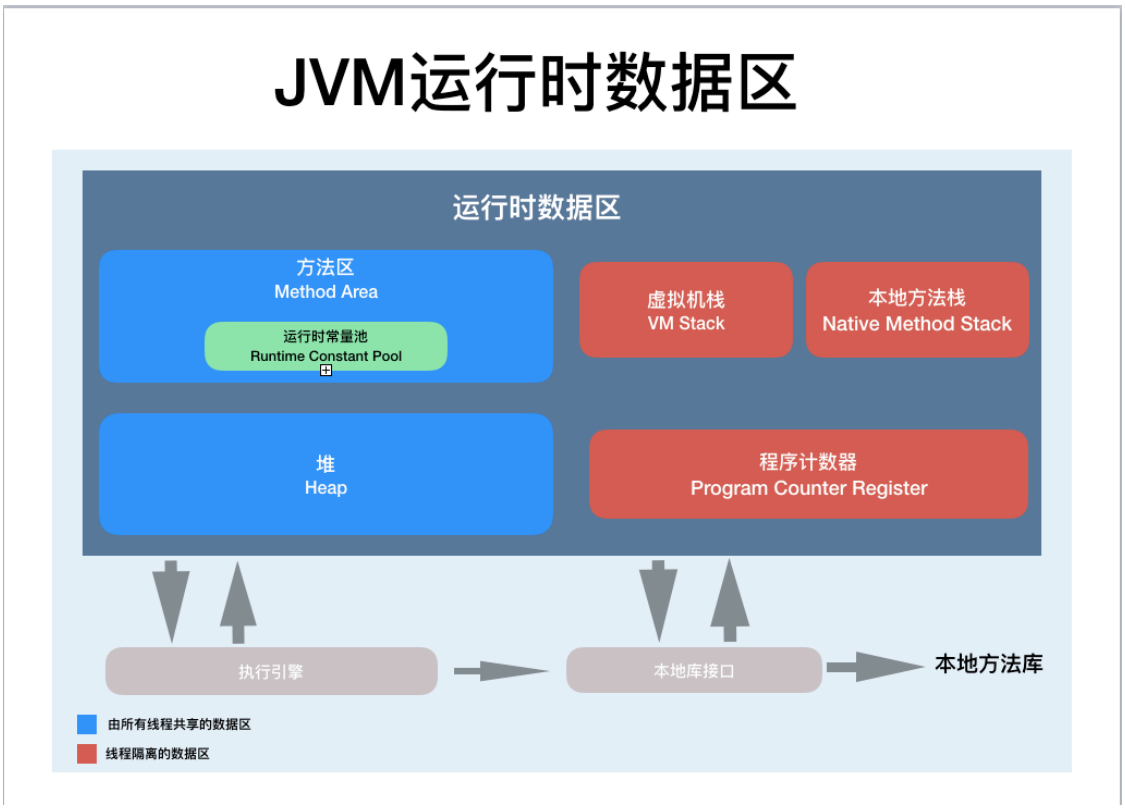
JVM 内存区域

JVM 运行时内存划分

Java 虚拟机在执行Java程序的过程中会把他所管理的内存划分为若干个不同数据区域。这些区域都有自己的用途，以及创建和销毁的时间有的区域随着虚拟机的启动而存在，有些区域则依赖用户线程的启动和结束而建立和销毁。

Java 7 Java虚拟机运行时划分：



Question

- 内存划分的用途?
- 各个内存用于存储什么?

程序计数器

Question

- 什么是程序计数器?
- 程序计数器作用是什么?
- 为什么程序计数器是线程私有内存?

程序计数器 （Program Counter Register） 是一块较小的内存空间，他可以看作是当前线程所执行的字节码的信号指示器。

字节码解释器工作时是通过这个计数器的值选取下一条需要执行的字节码指令，分支、循环、异常处理、线程恢复等基础功能都需要依赖这个计数器完成。

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，一个处理器都会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，每条线程都需要一个独立的程序计数器，各个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执行地址。如果正在执行的是一个native方法，这个计数器值为空（Undefined）。

Java虚拟机栈

Java虚拟机栈（Java Virtual Machine Stacks）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，诶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出栈的过程。

本地方法栈

本地方法栈（Native Method Stack ）与虚拟机栈作用非常相似，区别不过是虚拟机栈为虚拟机执行Java方法服务，而本地方法栈则为虚拟机使用到的native方法服务。

虚拟机规范对本地方法栈中方法使用的语言，使用方式与数据结果没有强制规定。

Java 堆

Question

- Java堆什么时候创建的?
- Java堆用于存储什么?

Java堆 （Java Heap）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分配内存。**

> The heap is the runtime data area from thich memory for all class interface and arrays is allocated

但是随着**JIT编译器**的发展与**逃逸分析**技术逐渐成熟，**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对堆上也渐渐变得不是那么“绝对”了。

注：

1. JIT编译技术请参考《深入JVM系列-JIT编译详解》
2. 逃逸分析、同步省略、栈上分配、标量替换请参考《深入JVM系列-逃逸分析、同步省略、栈上分配、标量替换》

Java 堆是垃圾收集器管理的主要区域，因此很多时候也被称为“GC堆”。从内存回收的角度来看，由于现在收集器基于 **分代收集算法**,所以 Java 堆中还可以细分为：新生代和老年代。 再细致一点的有 Eden 空间 、From Survivor 空间、To Survivor 空间等 从内存分配的角度来看，线程共享的 Java 堆中可能分出多个线程私有的分配缓冲区 (Thread Local Allocation Buffer TLAB)。

不过无论如何划分，都与存储内容无关，无论那个区域，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，或者更快的分配内存。

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。当前主流的虚拟机都是按照可拓展来实现的通过 虚拟机参数 **(-Xmx 和 -Xms)** 控制。如果堆中的内存没有完成实例分配，而且堆也无法在拓展时，会抛出 OutOfMemoryError 错误。

方法区

Question

- 方法区属于永久代吗?

方法区 （Method Area） 与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、即时编辑器编译后的代码等数据。

方法区本质上和永久代并不等价，仅仅是因为 Hotspot 设计团队选择把 GC 分代收集拓展至方法区，这样 Hotspot 的垃圾收集器可以像管理Java堆一样管理这部分内存，能够省去专门为方法区编写内存管理代码工作。对于其他虚拟机（如 BEA JRickit、IBM J9 等）来说就不存在永久代的概念的。

永久代有 -XX:MaxPermSize的上限， JDK 1.7的 Hotspot 中已经把原来放在永久代的字符串常量池移出。

Java虚拟机对方法区的限制非常宽松，除了和Java 堆一样不需要连续得内存和可已选择固定大小或者可拓展外，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对类型的卸载，一般来说，这个区域的回收“成绩”比较难以令人满意，尤其是类型的卸载，条件相对苛刻，但是这部分区域的回收是确实必要的。

根据Java虚拟机规范规定，当方法区无法满足内存分配需求时，将抛出 OutOfMemoryError 错误。

运行时常量池

运行时常量池（Runtime Constant Pool）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（Constant Pool Table）,用于存放编译期生成的各种字面量和符合引用，这部分内容将在类加载后放入方法区的运行时常量池中存放。

运行时常量池相对于Class文件常量池的另外一个重要特征是具备动态性，**Java语言并不要求一定只有编译期才能产生，也就是并非预置如Class文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量池放入池中**，这种特性被开发人员用的最多便是 String 类中 intern() 方法。

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虚拟机规范定义的内存区域，但是这部分内存也被频繁使用，而且可能导致 OutOfMemoryError 异常出现。

在 JDK 1.4 中添加的 NIO 类引入一种基于通道（Channel） 和 缓冲区 (Buffer) 的 I/O 方式，他可以使用 Native 函数直接分配堆外内存，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提高性能，避免在 Java 堆和 Native 堆中来回复制数据。

虽然，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 肯定还是受到本机总内存，（包含 RAM 以及 SWAP 区或者分页文件）大小及处理器寻址空间的限制。如果忽略这部分空间，使得各个内存区域总和大于物理内存限制，从而会导致动态扩展时会出现 OutOfMemoryError 异常出现。

HotSpot 虚拟机对象揭秘

轻思考以下问题

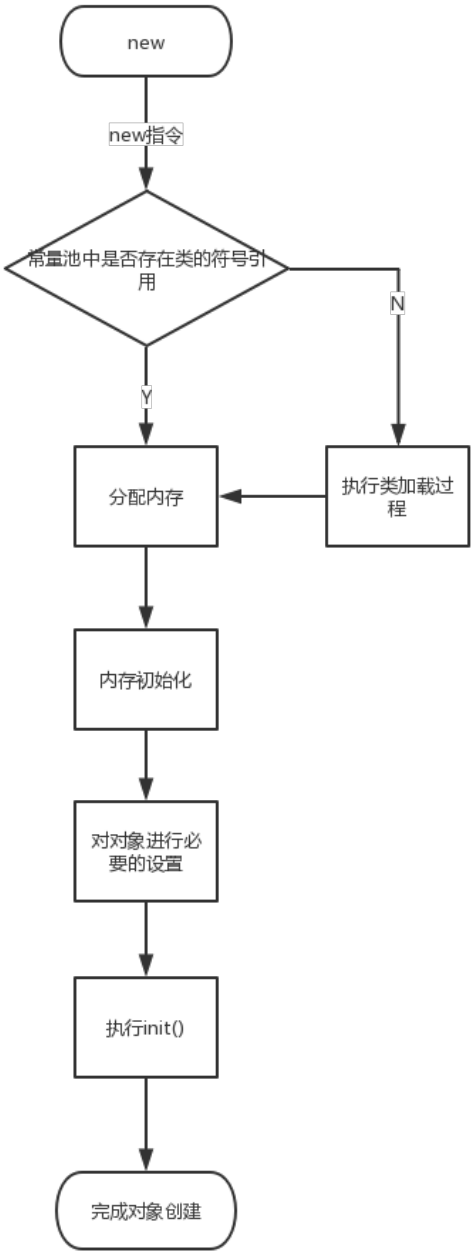
Question

- 对象是如何在虚拟机上分配、布局、访问的?

接下来我们以最常用的虚拟机 Hotspot 和 常用的内存区域 Java 堆为例 深入探讨 Hotspot虚拟机在 Java堆上如何对象分配、布局和访问的全过程。

对象的创建

对象创建的流程图:



下面我们详细讲解一下每一个步骤：

常量池检查过程

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。

分配内存

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。

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。

对对象设置

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,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。

执行init()

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。

HotSpot解释器的代码片段:

```
    //确保常量池中存放的是已解释的类
    if (!constants ->tag_at (index) .is_unresolved_klass()) {
        //断言确保是klassOop和instanceKlassOop
        oop
        entry = (klassOop) *constants ->obj_at_addr (index) ;
        assert (entry ->is_klass(), "Should be resolved klass") ;
        klassOop
        k_entry = (klassOop) entry;
        assert (k_entry ->klass_part() ->oop_is_instance(), "Should be
instanceKlass") ;
        instanceKlass * ik = (instanceKlass *) k_entry ->klass_part();
        //确保对象所属类型已经经过初始化阶段
        if (ik ->is_initialized()&&ik ->can_be_fastpath_allocated())
        {
            //取对象长度
            size_t
            obj_size = ik ->size_helper();
            oop
            result = NULL;
            //记录是否需要将对象所有字段置零值
            bool need_zero = !ZeroTLAB;
            //是否在TLAB中分配对象
            if (UseTLAB) {
                result = (oop) THREAD ->tlab().allocate (obj_size) ;
            }
            if (result == NULL) {
```

```
                need_zero = true;
                //直接在eden中分配对象
                retry:
                HeapWord * compare_to = *Universe:heap() ->top_addr();
                HeapWord * new_top = compare_to + obj_size;
                /*cmpxchg是x86中的CAS指令，这里是一个C++方法，通过CAS方式分配
空间，如果并发失败，
                    转到retry中重试，直至成功分配为止*/
                if (new_top<=*Universe:heap() ->end_addr()) {
                    if (Atomic:
                    cmpxchg_ptr (new_top, Universe:heap() ->
top_addr(), compare_to) !=compare_to) {
                        goto retry;
                    }
                    result = (oop) compare_to;
                }
            }
            if (result != NULL) {
                //如果需要，则为对象初始化零值
                if (need_zero) {
                    HeapWord * to_zero = (HeapWord *) result +
sizeof (oopDesc) /oopSize;
                    obj_size -= sizeof (oopDesc) /oopSize;
                    if (obj_size>0) {
                        memset (to_zero, 0, obj_size * HeapWordSize) ;
                    }
                }
                //根据是否启用偏向锁来设置对象头信息
                if (UseBiasedLocking) {
                    result ->set_mark (ik ->prototype_header()) ;
                }else{
                    result ->set_mark (markOopDesc:
                    prototype()) ;
                }
                result ->set_klass_gap (0) ;
                result ->set_klass (k_entry) ;
                //将对象引用入栈，继续执行下一条指令
                SET_STACK_OBJECT (result, 0) ;
                UPDATE_PC_AND_TOS_AND_CONTINUE (3, 1) ;
            }
        }
    }
```

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）。

对象头

HotSpot虚拟机的对象头包括两部分信息，第一部分是对象的自身数据，第二部分是类型指针。

对象自身数据

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

类型指针

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。

实例数据

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参数（FieldsAllocationStyle）和字段在Java源码中定义顺序的影响。HotSpot虚拟机默认的分配策略为longs/doubles、ints、shorts/chars、bytes/booleans、oops（Ordinary Object Pointers），从分配策略中可以看出，相同宽度的字段总是被分配到一起。 在满足这个前提条件的情况下，在父类中定义的变量会出现在子类之前。如果CompactFields参数值为true（默认为true），那么子类之中较窄的变量也可能会插入到父类变量的空隙之中。

对齐填充

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。而对象头部分正好是8字节的倍数（1倍或者2倍），因此，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。

对象的访问和定位

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。

句柄

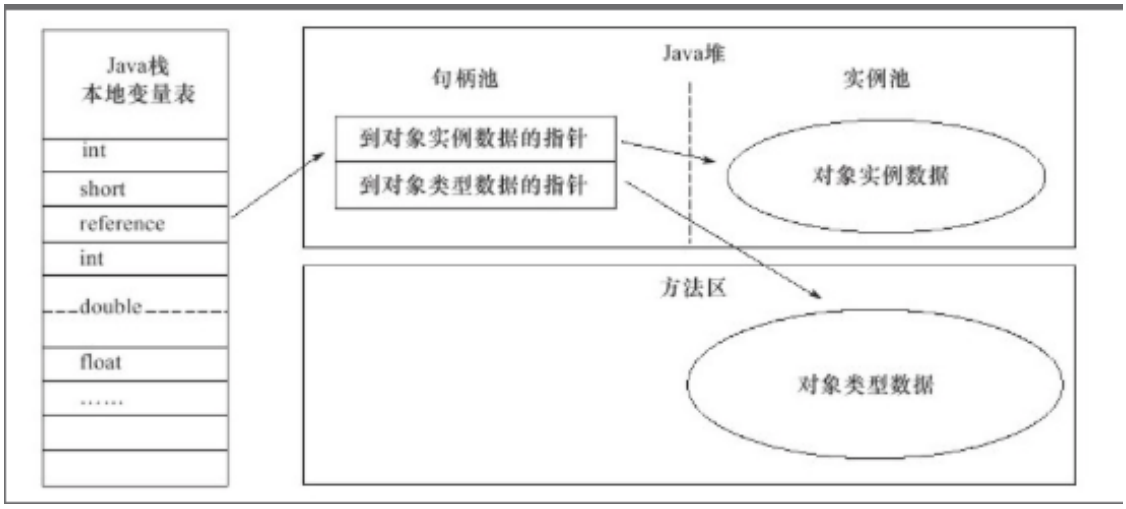
Question

- 什么是句柄?

句柄

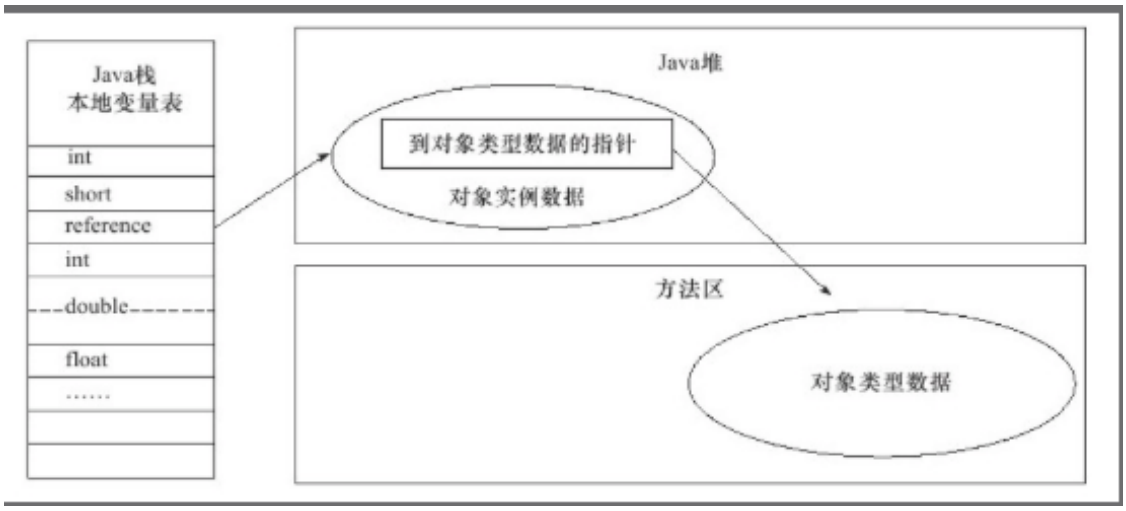
关于句柄参看 句柄文章

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地址信息



指针

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。



对比

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。

实战篇

实战篇分别介绍和探讨内存溢出和内存泄漏

内存溢出

在Java虚拟机规范的描述中，除了程序计数器外，虚拟机内存的其他几个运行时区域都有发生OutOfMemoryError（下文称OOM）异常的可能

Java堆溢出

Java堆用于存储对象实例，只要不断地创建对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收机制清除这些对象，那么在对象数量到达最大堆的容量限制后就会产生内存溢出异常。

例：

```
/**
 *VM Args:-Xms20m-Xmx20m-XX:+HeapDumpOnOutOfMemoryError
 *@author zzm
 */
public class HeapOOM{
 static class OOMObject{
}

public static void main (String[]args) {
List<OOMObject>list=new ArrayList<OOMObject>();

    while (true) {
    list.add (new OOMObject()) ;
    }
    }
}
```

```
java.lang.OutOfMemoryError:Java heap space
Dumping heap to java_pid3404.hprof.....
Heap dump file created[22045981 bytes in 0.663 secs]
```

代码限制Java堆的大小为20MB，不可扩展（将堆的最小值-Xms参数与最大值-Xmx参数设置为一样即可避免堆自动扩展），通过参数-XX:+HeapDumpOnOutOfMemoryError可以让虚拟机在出现内存溢出异常时Dump出当前的内存堆转储快照以便事后进行分析

如果是内存泄露，可进一步通过工具查看泄露对象到GC Roots的引用链。于是就能找到泄露对象是通过怎样的路径与GC Roots相关联并导致垃圾收集器无法自动回收它们的。掌握了泄露对象的类型信息及GC Roots引用链的信息，就可以比较准确地定位出泄露代码的位置。

虚拟机栈和本地方法栈溢出

如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常。

如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出OutOfMemoryError异常。

例：

```
public class TestError {

    private int stackLength = 1;

    public void stackLeak() {
        stackLength++;
        stackLeak();
    }

    public static void main(String[] args) throws Throwable {
        TestError oom = new TestError();
        try {
            oom.stackLeak();
        } catch (Throwable e) {
            System.out.println("stack length:" + oom.stackLength);
            throw e;
        }
    }
}
```

运行结果：

```
stack length:18984
Exception in thread "main" java.lang.StackOverflowError
    at error.TestError.stackLeak(TestError.java:10)
    at error.TestError.stackLeak(TestError.java:10)
    at error.TestError.stackLeak(TestError.java:10)
    // ....后续异常堆栈信息省略
```

在单个线程下，无论是由于栈帧太大还是虚拟机栈容量太小，当内存无法分配的时候，虚拟机抛出的都是StackOverflowError异常。

如果测试时不限于单线程，通过不断地建立线程的方式倒是可以产生内存溢出异常，如代码清单2-5所示。但是这样产生的内存溢出异常与栈空间是否足够大并不存在任何联系，或者准确地说，在这种情况下，为每个线程的栈分配的内存越大，反而越容易产生内存溢出异常。

其实原因不难理解，操作系统分配给每个进程的内存是有限制的，譬如32位的Windows限制为2GB。虚拟机提供了参数来控制Java堆和方法区的这两部分内存的最大值。剩余的内存为2GB（操作系统限制）减去Xmx（最大堆容量），再减去MaxPermSize（最大方法区容量），程序计数器消耗内存很小，可以忽略掉。如果虚拟机进程本身耗费的内存不计算在内，剩下的内存就由虚拟机栈和本地方法栈“瓜分”了。每个线程分配到的栈容量越大，可以建立的线程数量自然就越少，“建立线程时就越容易把剩下的内存耗尽。

例：

```
public class TestMultThread {

    private void dontStop() {
        while (true) {
        }
    }

    public void stackLeakByThread() {
        while (true) {
            Thread thread = new Thread(new Runnable() {
                @Override
                public void run() {
                    dontStop();
                }
            });
            thread.start();
        }
    }

    public static void main(String[] args) throws Throwable {
        TestMultThread oom = new TestMultThread();
        oom.stackLeakByThread();
    }
}
```

运行结果：

```
Exception in thread"main"java.lang.OutOfMemoryError:unable to create new
native thread
```

内存泄漏

内存泄露是指程序中间动态分配了内存，但在程序结束时没有释放这部分内存，从而造成那部分内存不可用的情况，重启计算机可以解决，但也有可能再次发生内存泄露，内存泄露和硬件没有关系，它是由软件设计缺陷引起的。

内存优化建议

Q: 如何避免内存泄露、溢出?

- 尽早释放无用对象的引用。

好的办法是使用临时变量的时候，让引用变量在退出活动域后自动设置为null，暗示垃圾收集器来收集该对象，防止发生内存泄露。

- 程序进行字符串处理时，尽量避免使用String，而应使用StringBuffer。

- 尽量少用静态变量。

因为静态变量是全局的，GC不会回收。

- 避免集中创建对象尤其是大对象，如果可以的话尽量使用流操作。

- 尽量运用对象池技术以提高系统性能。

生命周期长的对象拥有生命周期短的对象时容易引发内存泄漏，例如大集合对象拥有大数据量的业务对象的时候，可以考虑分块进行处理，然后解决一块释放一块的策略。

- 不要在经常调用的方法中创建对象，尤其是忌讳在循环中创建对象。

可以适当的使用hashtable，vector创建一组对象容器，然后从容器中去取那些对象，而不用每次new之后又丢弃。

参考

《周志明.深入理解Java虚拟机：JVM高级特性与最佳实践》

http://hg.openjdk.java.net/

https://blog.csdn.net/sinat_35512245/article/details/54866068