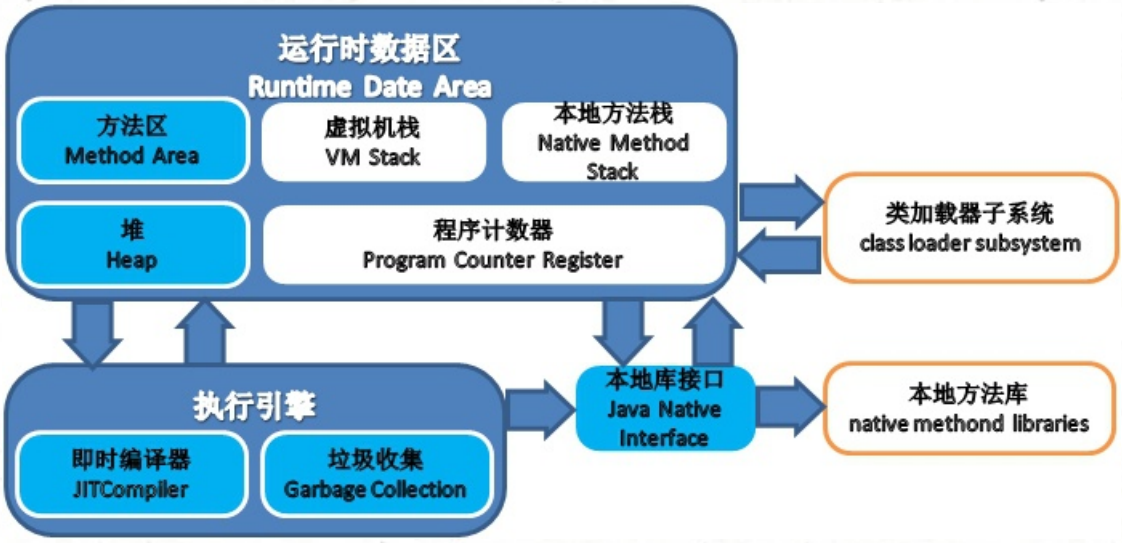
https://github.com/dodola/dodo_algorithm/blob/master/java/jvm/jvmdui_zhan.md
http://blog.csdn.net/L664675249/article/details/51221519



#一、JVM运行时数据区# 一般情况都将JVM运行期的数据区域区分成堆和栈两个区域，可以粗略的认为堆中存放的是对象和数组，栈中存放的是方法中定义的基本数据类型和对象的引用变量。这算是最简单的答案，有的面试官会继续深入的问，其切入点也就是将堆和栈拆分成其他几个具体的区域，主要分为①所有线程都共享的区域包括方法区和堆，②线程隔离的数据区，包括 虚拟机栈、本地方法栈、程序计数器。具体的区分如下图

- 程序计数器 当前线程所执行字节码的行号指示器，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。 作用：JVM是通过轮流执行线程饼分配处理器执行时间的方式来实现的，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计数器，如果执行的是Java方法，计数器记录的是正在执行的字节码的地址，如果执行的是Native方法，计数器的值为空，此内存是唯一一个没有规定OOM情况的区域。
- 虚拟机栈 虚拟机栈描述的是Java方法执行的内存模型，属于线程私有，其声明周期与线程相同。每个方法执行的时候都会同时创建一个栈帧，栈帧里包含了用户存储变量表，操作栈，动态链接，方法出口等信息，每个方法调用直至执行完成的过程就对应着一个栈帧入栈到出栈的过程。我们通常所说的栈指的就是虚拟机栈中的局部变量表。局部变量表存储了编译器可知的基本数据类型(boolean,short,char,int,float,long,double)、对象引用、returnAddress

类型。局部变量表所需的空间在*编译期*中就已经确定。

- 堆 堆是被所有线程共享的一块区域，在虚拟机启动的时候创建，唯一目的就是存放对象实例，所有的对象实例和数组都在堆上分配。
- 方法区
- 运行时常量池, 属于方法区的一部分，存储类信息、常量(final修饰,不可变)、静态变量(static)、即时编译器编译后的代码

##Java 堆## 在 Java 虚拟机中,堆(Heap)是可供各条线程共享的运行时内存区域,也是供所有类实例和数组对象分配内存的区域。

Java 堆在虚拟机启动的时候就被创建,它存储了被自动内存管理系统(Automatic Storage Management System,也即是常说的“Garbage Collector(垃圾收集器)”)所管理的各种对象,这些受管理的对象无需,也无法显式地被销毁。本规范中所描述的 Java 虚拟机并未假设采用什么具体的技术去实现自动内存管理系统。虚拟机实现者可以根据系统的实际需要来选择自动内存管理技术。Java 堆的容量可以是固定大小的,也可以随着程序执行的需求动态扩展,并在不需 要过多空间时自动收缩。Java 堆所使用的内存不需要保证是连续的。

Java 虚拟机实现应当提供给程序员或者最终用户调节 Java 堆初始容量的手段,对于可以动 态扩展和收缩 Java 堆来说,则应当提供调节其最大、最小容量的手段。

Java 堆可能发生如下异常情况: 如果实际所需的堆超过了自动内存管理系统能提供的最大容量,那Java虚拟机将会抛出一个 OutOfMemoryError 异常。

##方法区##

在Java虚拟机中,方法区(Method Area)是可供各条线程共享的运行时内存区域。方法区与传统语言中的编译代码储存区(Storage Area Of Compiled Code)或者操作系统进程 的正文段(Text Segment)的作用非常类似,它存储了每一个类的结构信息,例如运行时常量 池(Runtime Constant Pool)、字段和方法数据、构造函数和普通方法的字节码内容、还包 括一些在类、实例、接口初始化时用到的特殊方法。

方法区在虚拟机启动的时候被创建,虽然方法区是堆的逻辑组成部分,但是简单的虚拟机实现 可以选择在这个区域不实现垃圾收集。

方法区的容量可以是固定大小的,也可以随着程序执行的需求动态扩展,并在不需要过多空间时自动收缩。方法区在实际内存空间中可以是不连续的。

方法区可能发生如下异常情况:如果方法区的内存空间不能满足内存分配请

求,那Java虚拟机将抛出一个 OutOfMemoryError 异常。

##运行时常量池##

运行时常量池(Runtime Constant Pool)是每一个类或接口的常量池(Constant_Pool)的运行时表示形式,它包括了若干种不同的常量:从编译期可知的数值字面量到必须运行 期解析后才能获得的方法或字段引用。运行时常量池扮演了类似传统语言中符号表(Symbol Table)的角色,不过它存储数据范围比通常意义上的符号表要更为广泛。

每一个运行时常量池都在 Java 虚拟机的方法区中,在类和接口被加载到 虚拟机后,对应的运行时常量池就被创建出来。

在创建类和接口的运行时常量池时,可能会发生如下异常情况:当创建类或接口的时候,如果构造运行时常量池所需要的内存空间超过了方法区所能提供的最大值,那 Java 虚拟机将会抛出一个 OutOfMemoryError 异常。

##本地方法栈##

##栈帧## 栈帧(Frame)是用来存储数据和部分过程结果的数据结构,同时也被用来处理动态链接 (Dynamic Linking)、方法返回值和异常分派(Dispatch Exception)。

栈帧随着方法调用而创建,随着方法结束而销毁——无论方法是正常完成还是异常完成(抛出 了在方法内未被捕获的异常)都算作方法结束。

栈帧的存储空间分配在 Java 虚拟机栈之中,每一个栈帧都有自己的局部变量表(Local Variables)、操作数栈(Operand Stack)和指向当前方法所属的类的运行时常量池的引用。

局部变量表和操作数栈的容量是在编译期确定,并通过方法的 Code 属性保存及提供给栈帧使用。因此,栈帧容量的大小仅仅取决于 Java 虚拟机的实现和方法调用时可被分配的内存。

在一条线程之中,只有目前正在执行的那个方法的栈帧是活动的。这个栈帧就被称为是当前栈帧(Current Frame),这个栈帧对应的方法就被称为是当前方法(Current Method),定义这个方法的类就称作当前类(Current Class)。对局部变量表和操作数栈的各种操作,通常都指的是对当前栈帧的对局部变量表和操作数栈进行的操作。

如果当前方法调用了其他方法,或者当前方法执行结束,那这个方法的栈帧就不再是当前栈帧了。当一个新的方法被调用,一个新的栈帧也会随之而创建,并且随着程序控制权移交到新的方法而成为新的当前栈帧。当方法返回的之际,当前栈帧会传回此方法的执行结果给前一个栈帧,在方法返回之后,当前栈帧就随之被丢弃,前一个栈帧就重新成为当前栈帧了。

需要注意的是栈帧是线程本地私有的数据,不可能在一个栈帧之中引用另外一条线程的栈帧(这说明栈帧属于线程安全的区域)。

###局部变量表### 每个栈帧内部都包含一组称为局部变量表(Local Variables)的变量列表。栈帧中局部变量表的长度由编译期决定,并且存储于类和接口的二进制表示之中,既通过方法的 Code 属性保存及提供给栈帧使用。

一个变量槽(slot)可以保存一个类型为 boolean、byte、char、short、float、reference 和 returnAddress 的数据。

两个连续的变量槽(slot)可以保存一个类型为 long 和 double 的数据(需要注意)。

局部变量使用索引来进行定位访问,第一个局部变量的索引值为零,局部变量的索引值是从零至小于局部变量表最大容量的所有整数。 long 和 double 类型的数据占用两个连续的变量槽(slot),这两种类型的数据值采用两个变量槽(slot)中较小的索引值来定位。

例如我们讲一个 double 类型的值存储在索引值为 n 的变量槽(slot)中, 实际上的意思是索引值为 n 和 n+1 的两个变量槽(slot)都用来存储这个值。索引值为 n+1 的变量槽(slot)是无法直接读取的,但是可能会被写入,不过如果进行了这种操作,就将会导致变量槽(slot) n 的内容失效掉。

上文中提及的变量槽(slot) n 的 n 值并不要求一定是偶数,Java 虚拟机也不要求 double 和 long 类型数据采用 64 位对其的方式存放在连续的变量槽(slot)中。虚拟机实现者可以自由地选择适当的方式,通过两个变量槽(slot)来存储一个 double 或 long 类型的值。

类方法(static)被的参数将会传递至从 0 开始的连续的局部变量表位置上, 实例方法的第 0个位置用来存储被调用的实例方法所在的对象的引用(即 Java 语言中的“this” 关键字)。后续的其他参数将会传递至从 1 开始的连续的局部变量表位置上。

###操作数栈###

每一个栈帧内部都包含一个称为操作数栈(Operand Stack)的后进先出(Last-In-First-Out,LIFO)栈。栈帧中操作数栈的长度由编译期决定,并且存储于类和接 口的二进制表示之中,既通过方法的 Code 属性保存及提供给栈帧使用。

在上下文明确,不会产生误解的前提下,我们经常把“当前栈帧的操作数栈”直接简称为“操 作数栈”。 操作数栈所属的栈帧在刚刚被创建的时候操作数栈是空的。

Java虚拟机提供一些字节码指令来从局部变量表或者对象实例的字段中复制常量或变量值到操作数栈中,也提供了一些指令用于 从操作数栈取走数据、操作数据和把操作结果重新入栈。在方法调用的时候,操作数栈也用来准备调用方法的参数以及接收方法返回结果。

举个例子,iadd 字节码指令的作用是将两个 int 类型的数值相加,它要求在执行的之前操作 数栈的栈顶已经存在两个由前面其他指令放入的 int 型数值。在 iadd 指令执行时,2 个 int 值 从操作栈中出栈,相加求和,然后将求和结果重新入栈。在操作数栈中,一项运算常由多个子运算(Subcomputations)嵌套进行,一个子运算过程的结果可以被其他外围运算所使用。

每一个操作数栈的成员(Entry)可以保存一个 Java 虚拟机中定义的任意数据类型的值,包 括 long 和 double 类型。 在操作数栈中的数据必须被正确地操作,这里正确操作是指对操作数栈的操作必须与操作数栈 栈顶的数据类型相匹配,例如不可以入栈两个 int 类型的数据,然后当作 long 类型去操作他们, 或者入栈两个 float 类型的数据,然后使用 iadd 指令去对它们进行求和。有一小部分 Java 虚 拟机指令(例如 dup 和 swap 指令)可以不关注操作数的具体数据类型,把所有在运行时数据区 中的数据当作裸类型(Raw Type)数据来操作,这些指令不可以用来修改数据,也不可以拆散那 些原本不可拆分的数据,这些操作的正确性将会通过 Class 文件的校验过程(§4.10)来强制保障。

在任意时刻,操作数栈都会有一个确定的栈深度,一个 long 或者 double 类型的数据会占用 两个单位的栈深度,其他数据类型则会占用一个单位深度。

###动态链接### 每一个栈帧内部都包含一个指向运行时常量池的引用来支持当前方法的代码实现动态链接(Dynamic Linking)。在 Class 文件里面,描述一个方法调用了其他方法,或者访问其成员变量是通过符号引用(Symbolic Reference)来表示的,动态链接的作用就是将这些符号引用所表示的方法转换为实际方法的直接引用。

类加载的过程中将要解析掉尚未被解析的符号引用,并且将变量访问转化为访问这些变量的存储结构所在的运行时内存位置的正确偏移量。

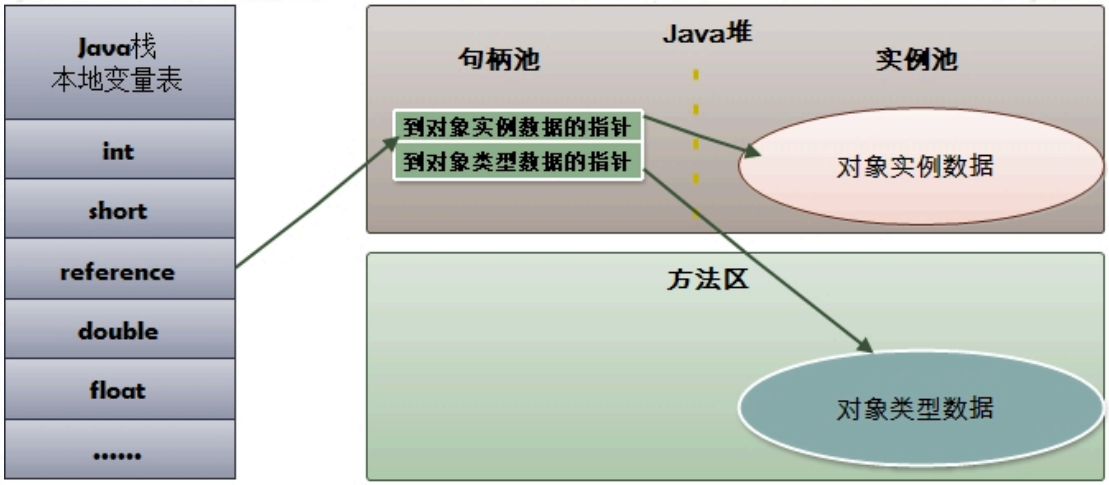
由于动态链接的存在,通过晚期绑定(Late Binding)使用的其他类的方法和变量在发生 变化时,将不会对调用它们的方法构成影响。

#二、对象访问#

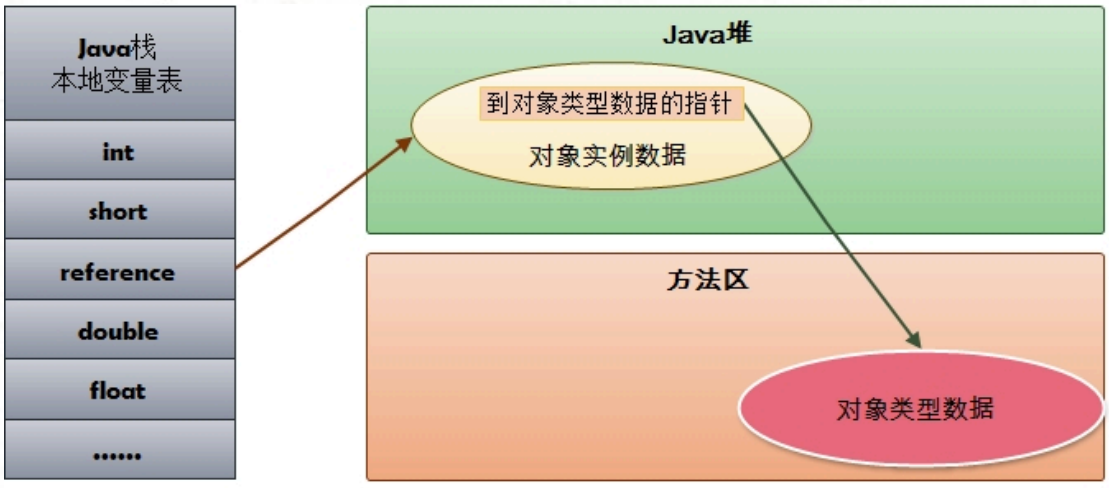
```
Object o=new Object();
```

```
//Object o 这部分语义将会反映到Java栈的本地变量表中，作为一个对象引用存在。
//new Object();这部分语义将会反映到Java堆中
```

对象访问方式主要有两种：句柄或者直接指针(速度更快) 句柄方式访问对象的内存分布如下



句柄方式访问对象



直接指针方式访问对象

问题

```
String a="123";
String b="123";
String c=new String("123");
String d=new String("123");
```

这几段代码在堆栈中的表现形式

面试的时候很多面试官都说里面是有一个字符串常量池的，其实这个字符

串常量池就是方法区中的常量池。

```
String a="123";//在虚拟机栈(本地变量表)中创建一个a的对象引用，指向常量池中的"123"
String b="123";//在虚拟机栈(本地变量表)中创建一个b的对象引用，指向常量池中的"123"
String c=new String("123");//在虚拟机栈(本地变量表)中创建一个c的对象引用，new String("123");这部分语义会在堆中创建一个String对象，c指向的是堆中的对象
String d=new String("123");//同上。
```

接下来会问这几个对象是否相等的问题

```
String a="a";
String b=a+"b";
String bb="a"+"b";
String c="ab";
String d=new String(b);
System.out.println(b==c);//false
System.out.println(bb==c);//true
System.out.println(c==d);//false
System.out.println(c==d.intern());//true
System.out.println(b.intern()==d.intern());//true
```