目录

一、JVM 内存分哪几个区，每个区的作用是什么?.....2
二、 如和判断一个对象是否存活?(或者 GC 对象的判定方法)2
三、 简述 java 垃圾回收机制..... 3
四、 java 中垃圾回收的算法有哪些？ 3
五、 java 内存模型.....4
六、 java 类加载过程.....4
七、 简述 java 类加载机制?.....5
八、 什么是类加载器，类加载器有哪些?.....5
九、简述 java 内存分配与回收策略以及 Minor GC 和 Major GC（Full GC）..... 6
十、 什么是 Java 虚拟机？为什么 Java 被称作是“平台无关的编程语言”？ 6
十一、 JDK 和 JRE 的区别是什么？ 7
十二、 常用的内存调试工具有哪些？ 7
十三、静态分派与动态分派是什么？（看源码）..... 7
十四、 对象创建方法，对象的内存布局，对象的访问定位。7
（1）对象创建..... 7
（2）对象内存布局..... 9
（3）对象访问定位..... 10
十五、GC 收集器有哪些？ 11
源码 1..... 12
源码 2..... 13

一、JVM 内存分哪几个区，每个区的作用是什么?

方法区（公有）：

1. 有时候也成为永久代，在该区内很少发生垃圾回收，但是并不代表不发生 GC，在这里进行的 GC 主要是对方法区里的常量池和对类型的卸载

2. **方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后的代码等数据。**

3. 该区域是被线程共享的。

4. 方法区里有一个**运行时常量池**，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

堆（公有）

java 堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这里创建，因此该区域经常发生垃圾回收操作。

虚拟机栈:（私有）

1. 虚拟机栈也就是我们平常所称的栈内存,它为 java 方法服务，每个方法在执行的时候都会创建一个栈帧，**用于存储局部变量表、操作数栈、动态链接和方法出口等信息**。

2. 虚拟机栈是线程私有的，它的生命周期与线程相同。

3. **局部变量表里存储的是基本数据类型、returnAddress 类型（指向一条字节码指令的地址）和对象引用**，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表对象的句柄或者与对象相关联的位置。局部变量所需的内存空间在编译器间确定

4.操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，而是压栈和出栈的方式

5.每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接.动态链接就是将常量池中的符号引用在运行期转化为直接引用。

本地方法栈（私有）

本地方法栈和虚拟机栈类似，只不过本地方法栈为 Native 方法服务。

程序计数器（私有）

内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条**需要执行的字节码指令**，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个 java 虚拟机规范**没有规定任何 OOM 情况的区域**。

二、如和判断一个对象是否存活?(或者 GC 对象的判定方法)

1. 引用计数法

所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说明此对象没有被引用，也就是“死对象”,将会被垃圾回收.

引用计数法有一个缺陷就是**无法解决循环引用**问题，也就是说当对象 A 引用对象 B，对象 B 又引用者对象 A，那么此时 A,B 对象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流的虚拟机都没有采用这种算法。

2.可达性算法(引用链法)

该算法的思想是：从一个被称为 GC Roots 的对象开始向下搜索，**如果一个对象到 GC Roots 没有任何引用链相连时，则说明此对象不可用**。
在 java 中可以作为 GC Roots 的对象有以下几种:

虚拟机栈中引用的对象
方法区类静态属性引用的对象
方法区常量池引用的对象
本地方法栈 JNI 引用的对象

虽然这些算法可以判定一个对象是否能被回收，但是当满足上述条件时，一个对象比不一定会被回收。当一个对象不可达 GC Root 时，这个对象并 不会立马被回收，而是出于一个死缓的阶段，**若要被真正的回收需要经历两次标记**

标记 1：如果对象在可达性分析中没有与 GC Root 的引用链，那么此时就会被第一次标记并且进行一次筛选，筛选的条件是是否有必要执行 finalize()方法。当对象没有覆盖 finalize()方法（只会被调用一次）或者已被虚拟机调用过，那么就认为是没必要的，该对象标记为垃圾。

标记 2：如果该对象有必要执行 finalize()方法，那么这个对象将会放在一个称为 F-Queue 的对队列中，虚拟机会触发一个 Finalize()线程去执行，此线程是低优先级的，并且虚拟机不会承诺一直等待它运行完，这是因为如果 finalize()执行缓慢或者发生了死锁，那么就会造成 F-Queue 队列一直等待，造成了内存回收系统的崩溃。GC 对处于 F-Queue 中的对象进行第二次被标记，如果执行 finalize 过程中该对象与其他对象仍然没有建立关联，这时，该对象将被移除”即将回收”集合，等待回收，否则放回堆里。

三、简述 java 垃圾回收机制

在 java 中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。在 JVM 中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

四、java 中垃圾回收的算法有哪些？

1 标记-清除:

这是垃圾收集算法中最基础的，根据名字就可以知道，它的思想就是标记哪些要被回收的对象，然后统一回收。这种方法很简单，但是会有两个主要问题：1.效率不高，标记和清除的效率都很低；2.会**产生大量不连续的内存碎片**，导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次 GC 动作。

2 复制算法:

为了解决效率问题，**复制算法将可用内存按容量划分为相等的两部分，然后每次只使用其中的一块，当一块内存用完时，就将还存活的对象复制到第二块内存上，然后一次性清楚完第一块内存，再将第二块上的对象复制到第一块**。但是这种方式，内存的代价太高，每次基本上都要浪费一半的内存。

于是将该算法进行了改进，内存区域不再是按照 1：1 去划分，而是将内存划分为 8:1:1

三部分，较大那份内存叫 Eden 区，其余是两块较小的内存区叫 Survior 区。每次都会优先使用 Eden 区，若 Eden 区满，就将对象复制到第二块内存区上，然后清除 Eden 区，如果此时存活的对象太多，以至于 Survivor 不够时，会将这些对象通过**空间分配担保机制**复制到老年代中。(java 堆又分为新生代和老年代)

3 标记-整理

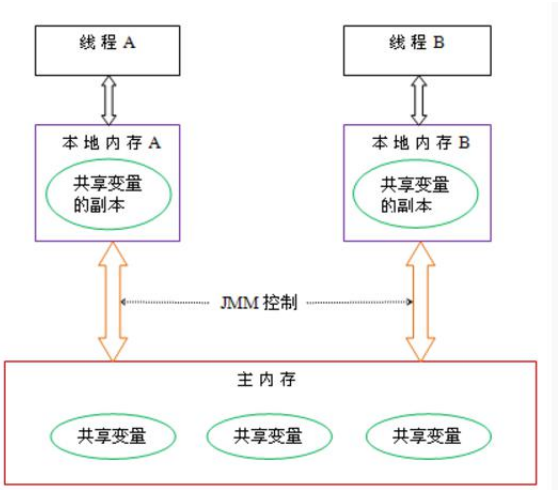
该算法主要是为了解决标记-清除，产生大量内存碎片的问题；当对象存活率较高时，也解决了复制算法的效率问题。它的不同之处就是在清除对象的时候先将存活对象移动到一端，然后清除掉端边界以外的对象，这样就不会产生内存碎片了。

4 分代收集

现在的虚拟机垃圾收集大多采用这种方式，它根据对象的生存周期，将堆分为新生代和老年代。在新生代中，由于对象生存期短，每次回收都会有大量对象死去，那么这时就采用复制算法。老年代里的对象存活率较高，没有额外的空间进行分配担保，所以可以使用标记-整理（G1 收集器） 或者 标记-清除(CMS 收集器)。

五、java 内存模型

简单说就是：主内存+本地内存（工作内存）



内存间交互有 8 中操作，每一种操作都是原子的不可再分的

1.lock 2.unlock **3.read** **4.load** 5.use 6.assign **7.store** **8.write**

六、java 类加载过程

java 类需要经历 7 个过程：加载、验证、解析、准备、初始化、使用、卸载

系统提供三种类加载器：

启动类加载器（<JAVA_HOME>\lib ）-----无法直接引用

扩展类加载器（<JAVA_HOME>\lib\ext>）----可以直接引用

应用程序类加载器（CLASSPATH）---负责加载用户指定路径上的类（默认使用这个加载器）。

加载

加载时类加载的第一个过程，在这个阶段，将完成一下三件事情：

1. 通过一个类的全限定名**获取该类的二进制流**。

2. 将该二进制流中的静态存储结构**转化**为方法区运行时数据结构。

3. 在内存中**生成该类的 Class 对象**，作为该类的数据访问入口。

验证

验证的目的是为了确保 Class 文件的字节流中的信息不会危害到虚拟机.在该阶段主要完成以下四种验证:

1. **文件格式验证**：验证字节流是否符合 Class 文件的规范，如主次版本号是否在当前虚拟机范围内，常量池中的常量是否有不被支持的类型.

2. **元数据验证**:对字节码描述的信息进行**语义分析**，如这个类是否有父类，是否集成了不被继承的类等。

3. **字节码验证**：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，**主要针对方法体的验证**。如：方法中的类型转换是否正确，跳转指令是否正确等。

4. 符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。

准备

准备阶段是为**类的静态变量分配内存并将其初始化为默认值**，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。

public static int value=123;//在准备阶段 value 初始值为 0 。在初始化阶段才会变为 123 。

解析

该阶段主要完成**符号引用到直接引用的转换动作**。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

初始化

执行类构造器<clinit>的过程。如：初始化类变量的初始值。

满足以下五个条件时类会被初始化（当然就会被加载）：

1 创建对象（new Object()）

2 通过反射加载类（Class clazz = Class.forName("org.com.Student") or = Student.class(不用放在 try 中)）

3 初始化子类时需要初始化父类（接口除外）

4 虚拟机启动时初始化 main 所在的类

5 利用 MethodHandle 生成方法句柄时，该方法对应的类需要初始化。

使用：略过

卸载：满足以下三个条件：**该类没有实例对象存在、该类加载器已 GC、该类的 class 对象没有任何引用**。

七、简述 java 类加载机制?

虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验，解析和初始化，最终形成可以被虚拟机直接使用的 java 类型,主要采用双亲委派模型机制进行加载：当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。

八、什么是类加载器，类加载器有哪些?

实现通过类的全限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

1. **启动类加载器**(Bootstrap ClassLoader)用来加载 java 核心类库，无法被 java 程序直接引用。

2. **扩展类加载器**(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。

3. **系统类加载器**（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。

4. **用户自定义类加载器**，通过继承 java.lang.ClassLoader 类的方式实现。

九、简述 java 内存分配与回收策略以及 Minor GC 和 Major GC(Full GC)

内存堆主要分为 **新生代和老年代**，新生代分为一个 **Eden 区和两个 Survivor 区**：8:1:1

（-Xmx10240m -Xms10240m -Xmn5120m -XXSurvivorRatio=3 问：最小内存值和 Survivor 区总大小分别是（请自己算一下）

注意：-Xmx10240m：代表最大堆

-Xms10240m：代表最小堆

-Xmn5120m：代表新生代

-XXSurvivorRatio=3：代表 Eden:Survivor = 3）

回收策略：对象优先在堆的 Eden 区分配。大对象（大数组和长字符串）直接进入老年代.长期存活的对象将直接进入老年代.当 Eden 区没有足够的空间进行分配时，虚拟机会执行一次 Minor GC.Minor Gc 通常发生在新生代的 Eden 区，在这个区的对象生存期短，往往发生 Gc 的频率较高，回收速度比较快;Minor GC 时会让老年代进行空间分配担保，如果担保失败，就会进行 full gc，Full Gc/Major GC 发生在老年代，一般情况下，触发老年代 GC 的时候不会触发 Minor GC,但是通过配置，可以在 Full GC 之前进行一次 Minor GC 这样可以加快老年代的回收速度。

十、什么是 Java 虚拟机？为什么 Java 被称作是“平台无关的编程语言”？

Java 虚拟机是一个可以执行 Java 字节码的虚拟机进程。Java 源文件被编译成能被 Java 虚拟机执行的字节码文件。

Java 是平台无关的语言是**指用 Java 写的应用程序不用修改就可在不同的软硬件平台上运行**。平台无关有两种：源代码级和目标代码级。C 和 C++具有一定程度的源代码级平台无关，表明用 C 或 C++写的应用程序不用修改只需重新编译就可以在不同平台上运行。

Java 主要靠 Java 虚拟机（JVM）在目标码级实现平台无关性。JVM 是一种抽象机器，它附着在具体操作系统之上，本身具有一套虚机器指令，并有自己的栈、寄存器组等。但 JVM 通常是在软件上而不是在硬件上实现。（目前，SUN 系统公司已经设计实现了 Java 芯片，主要使用在网络计算机 NC 上。另外，Java 芯片的出现也会使 Java 更容易嵌入到家用电器中。）JVM 是 Java 平台无关的基础，在 JVM 上，有一个 Java 解释器用来解释 Java 编译器编译后的程序。Java 编程人员在编写完软件后，通过 Java 编译器将 Java 源程序编译为 JVM 的字节代

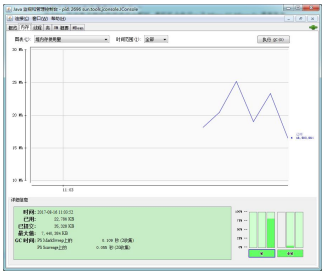
码。任何一台机器只要配备了 Java 解释器，就可以运行这个程序，而不管这种字节码是在何种平台上生成的。另外，Java 采用的是基于 IEEE 标准的数据类型。通过 JVM 保证数据类型的一致性，也确保了 Java 的平台无关性。

十一、JDK 和 JRE 的区别是什么？

JDK 是 Java 的开发工具，它不仅提供了 Java 程序运行所需的 JRE，还提供了一系列的编译，运行等工具，如 javac，java，javaw 等。JRE 只是 Java 程序的运行环境，它最核心的内容就是 JVM（Java 虚拟机）及核心类库。

十二、常用的内存调试工具有哪些？

jmap、jstack、**jconsole**，一下是 Jconsole，可以监控堆中各个区的内存占用情况



十三、静态分派与动态分派是什么？（看源码）

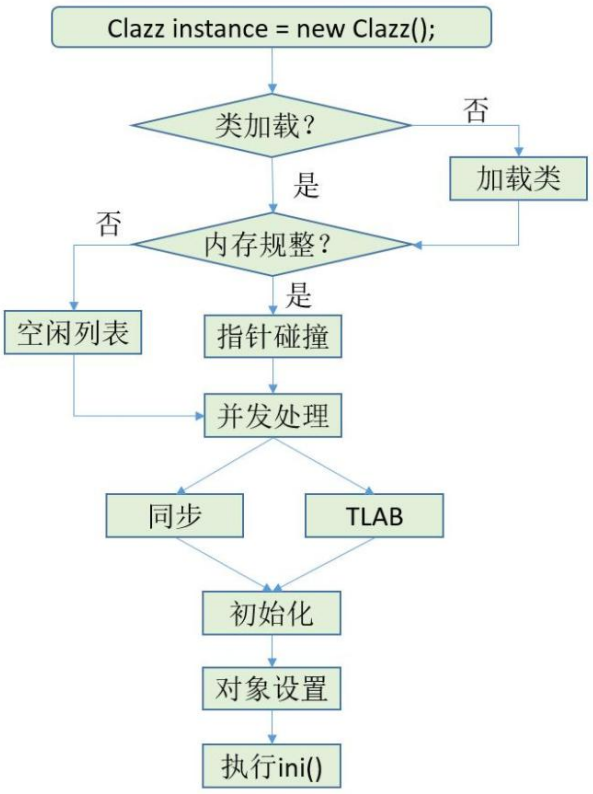
静态分派只会涉及重载，而重载是在编译期间确定的，那么静态分派自然是一个静态的过程（因为还没有涉及到 Java 虚拟机）。静态分派的最直接的解释是在重载的时候是通过参数的静态类型而不是实际类型作为判断依据的

动态分派的一个最直接的例子是重写。对于重写，我们已经很熟悉了，那么 Java 虚拟机是如何在程序运行期间确定方法的执行版本的呢？

十四、对象创建方法，对象的内存布局，对象的访问定位。

（1）对象创建

一个简单的创建对象语句 Clazz instance = new Clazz();包含的主要过程包括了**类加载检查、对象分配内存、并发处理、内存空间初始化、对象设置、执行 init 方法**等。



1. 类加载检查

JVM 遇到一条 new 指令时，首先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的加载过程。

2. 对象分配内存 (有两种方法：指针碰撞和空闲列表)

对象所需内存的大小在类加载完成后便完全确定（对象内存布局），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。

根据 Java 堆中是否规整有两种内存的分配方式：（Java 堆是否规整由所采用的垃圾收集器是否带有压缩整理功能决定）

指针碰撞(Bump the pointer)

Java 堆中的内存是规整的，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，分配内存也就是把指针向空闲空间那边移动一段与内存大小相等的距离。例如：Serial、ParNew 等收集器。

空闲列表(Free List)

Java 堆中的内存不是规整的，已使用的内存和空闲的内存相互交错，就没有办法简单的进行指针碰撞了。虚拟机必须维护一张列表，记录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。例如：CMS 这种基于 Mark-Sweep 算法的收集器。

3. 并发处理

对象创建在虚拟机中时非常频繁的行为，即使是仅仅修改一个指针指向的位置，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。

同步

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

本地线程分配缓冲（Thread Local Allocation Buffer, TLAB）

把内存分配的动作按照线程划分为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（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。

4. 内存空间初始化

虚拟机将分配到的内存空间都初始化为零值（不包括对象头）,如果使用了 TLAB，这一工作过程也可以提前至 TLAB 分配时进行。

内存空间初始化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**。

注意：类的成员变量可以不显示地初始化（Java 虚拟机都会先自动给它初始化为默认值）。方法中的局部变量如果只负责接收一个表达式的值，可以不初始化，但是参与运算和直接输出等其它情况的局部变量需要初始化。

5. 对象设置

虚拟机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之中。

6. 执行 init()

在上面的工作都完成之后，从虚拟机的角度看，一个新的对象已经产生了。但是从 Java 程序的角度看，对象的创建才刚刚开始。**init()方法还没有执行，所有的字段都还是零**。

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产生出来。

（2）对象内存布局

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：**运行时数据和类型指针**。

运行时数据

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。

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，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无法确定数组的大小。

(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，查找对象的元数据并不一定要经过对象本身，可参考对象的访问定位)

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。无论是从父类中继承下来的，还是在子类中定义的，都需要记录下来。HotSpot 虚拟机默认的分配策略为 longs/doubles、ints、shorts/chars、bytes/booleans、oop，从分配策略中可以看出，相同宽度的字段总是分配到一起。

对齐填充

HotSpot 虚拟机要求对象的起始地址必须是 8 字节的整数倍，也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的时候，就需要通过对齐填充来补全。

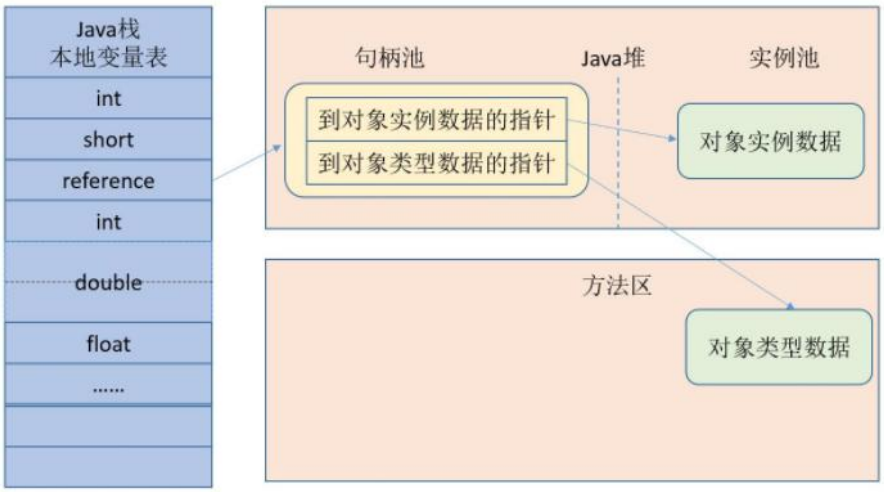
（3）对象访问定位

Java 程序需要通过栈上的引用数据来操作堆上的具体对象。对象的访问方式取决于虚拟机实现，目前主流的访问方式有使用**句柄和直接指针**两种。

句柄，可以理解为指向指针的指针，维护指向对象的指针变化，而对象的句柄本身不发生变化；指针，指向对象，代表对象的内存地址。

句柄

Java 堆中划分出一块内存来作为句柄池，引用中存储对象的句柄地址，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。

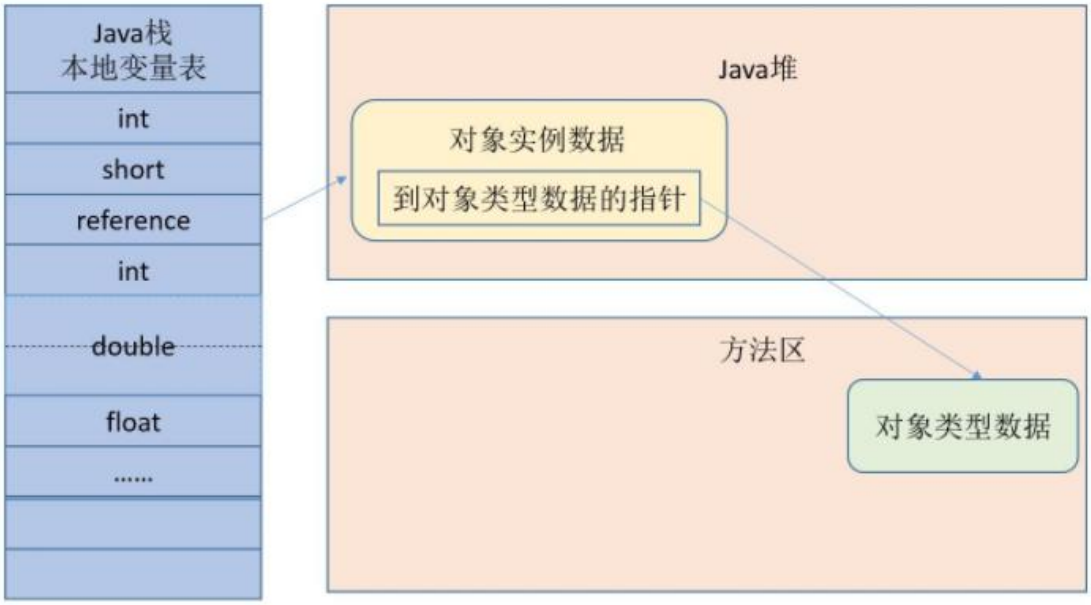


句柄访问对象

优势：**引用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，而引用本身不需要修改**。

直接指针

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，而引用中存储的直接就是对象地址。



直接内存访问对象

优势：**速度更快，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非常可观的执行成本。**（例如 HotSpot）

十五、GC 收集器有哪些？

串行垃圾回收器（Serial Garbage Collector）
并行垃圾回收器（Parallel Garbage Collector）
并发标记扫描垃圾回收器（CMS Garbage Collector）
G1 垃圾回收器（G1 Garbage Collector）

CMS 收集器（以获取最短回收停顿时间为目标），老年代使用

四个阶段：

初始标记：单线程，只标记与 GC Roots 相关联的对象，时间很快，但是要暂停所有线程

并发标记：不用暂停用户线程，进行 GC Roots Tracing，时间比较长，不过是和用户线程并发的。

重新标记：多线程标记，标记在并发标记阶段发生变化的记录，时间比初始标记长，但是远远小于并发标记

并发清除：和用户线程并发

默认启动的回收线程数是（CPU 数量+3）/4

缺点：降低吞吐量；不能处理浮动垃圾；会产生碎片。

G1 收集器（可预测的停顿）（可单独使用，标记-整理）能建立可预测的停顿时间模型

将 Java 堆分成若干个 Region，根据回收价值进行回收，每个 Region 有一个 Rememberd set,记录在不同 Region 的引用信息。

也是四个阶段：

初始标记，并发标记，最终标记（可以并行），筛选回收（根据各个 Region 的回收价值和成本排序，根据用户期望的 GC 停顿时间来制定回收计划）

源码 1

```
1.  import java.lang.invoke.MethodHandle;
2.  import java.lang.invoke.MethodHandles;
3.  import java.lang.invoke.MethodType;
4.  import java.util.Arrays;
5.  import java.util.List;
6.  public class MethodHandleTest {
7.      public static void main(String[] args) throws Throwable {
8.          MethodHandles.Lookup lookup = MethodHandles.lookup();
9.          MethodHandle mh = lookup.findStatic(MethodHandleTest.class, "doubleV
    al", MethodType.methodType(int.class, int.class));
10.         List<Integer> dataList = Arrays.asList(1, 2, 3, 4, 5);
11.         MethodHandleTest.transform(dataList, mh);// 方法做为参数
12.         for (Integer data : dataList) {
13.             System.out.println(data);//2,4,6,8,10
14.         }
15.     }
16.     public static List<Integer> transform(List<Integer> dataList, MethodHand
    le handle) throws Throwable {
17.         for (int i = 0; i < dataList.size(); i++) {
18.             dataList.set(i, (Integer) handle.invoke(dataList.get(i)));//invo
    ke
19.         }
20.         return dataList;
21.     }
22.     public static int doubleVal(int val) {
23.         return val * 2;
24.     }
```

```
25. }
```

源码 2

```
import java.util.*;
class A
{
	public static void say()
	{
		System.out.println("A say");
	}
	public void read()
	{
		System.out.println("A read");
	}
}
class B extends A
{
	public static void say()
	{
		System.out.println("B say");
	}
	public void read()
	{
		System.out.println("B read");
	}
}

public class Demo
{
	public static void sort(int[] array)
	{
		if(array == null || array.length ==1) return;



	}
	public void fun(A a)
	{
		a.say();
		a.read();
		System.out.println("fun A");
	}
	public void fun(B b)
	{
```

```
            b.say();
            b.read();
            System.out.println("fun B");
        }

        public static void main(String[] args)
        {
            A a = new B();
            new Demo().fun(a);
        }
}
```