# Java知识图谱

# Java虚拟机

Java（java virtual machine）虚拟机是执行java字节码的平台，通俗的讲就是一个程序。

## 运行时数据区

Java虚拟机在执行java程序的时候会把他所管理的内存划分为若干个不同的数据区域。这些区域有不同的用处以及有不同的创建时机。运行时数据区包括，程序计数器、java虚拟机栈、本地方法栈、堆、方法区（内含运行时常量池）、直接内存（非虚拟机管理的区域，但是会被经常使用）

程序计数器

方法区

堆

Java虚拟机栈

本地方法栈

执行引擎

本地方法

### 程序计数器

程序计数器是java运行时的一块较小的内存区域，它属于线程私有，随着线程的创建而创建，随着线程的销毁而销毁。它代表当前字节码执行的行号指示器，字节码解释器的工作就是通过这个行号指示器来选举下一条需要执行的指令、分支、跳转、异常以及线程切换等。如果执行的是java方法，此时代表的是字节码的行号，如果执行的是Native方法则程序计数器的值为空。此区域也是java虚拟机唯一标识没有OutOfMemoryError的区域。

### 虚拟机栈

与程序计数器一样，虚拟机栈也是线程私有的区域，它和线程的生命周期相同。虚拟机栈描述的是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用来存储局部变量表，操作数栈，方法的出入空信息。每个方法的调用就对应着栈帧的入栈和出栈的过程。

#### 局部变量表

局部变量表存放了编译时可知的数据类型,包含基本类型（byte、char、int、long、double、float、boolean）和对象引用（reference 可能是对象的引用指针或者是句柄）returnAdress（指向一条字节码指令的地址）。其中64位长度的long/double 占用两个局部变量表的空间其余的占用一个空间。局部变量表所需的空间在编译的时候就已经确定了。当进入一个方法的时候在栈帧中分配多大的局部变量表示确定的，不会在运行时期改变。

在java虚拟机规范中对此区域定义了两种异常情况，当线程请求的栈的深度大于虚拟机栈所允许的栈的深度的时候会出现StackOverflowError。当线程请求所需内存大于虚拟机栈所分配的内存的时候会出现OutOfMemoryError异常。

### 本地方法栈

本地方法栈和虚拟机栈一样也是线程私有的区域，只不过本地方法栈运行的是Native方法，与虚拟机栈一样本地方法栈也会抛出StackOverflowError和OutOfMemoryError。

### 堆

堆是java虚拟机管理的最大的一块区域，它是被线程共享的一块区域，在虚拟机创建的时候创建，此区域唯一的目的是存放对象的实例，数组对象等。java堆是垃圾收集器管理的主要区域，基于垃圾收集算法的不同，堆划分为新生代和老年代。java虚拟机规范了此区域可以处于物理上不连续的空间，只要是逻辑上连续即可。在此区域中可能发生OutOfMemoryError异常，堆的大小是可控的使用-XX:max20m -XX:mas20m 来调整堆的大小。

### 方法区

方法区和堆一样也是属于线程共享的一块区域，他用于存储已被虚拟机加载的类信息、常量、静态变量以及及时编译后的代码数据等信息。虽然java虚拟机把方法区描述为堆的一部分，但是它确是“非堆”对于再SunHotspot虚拟机上开发的人员来说此区域被称作永久代。

### 运行时常量池

运行时常量池属于方法区的一部分，用于存放编译期生产的字面量和符号引用。这部分内容将在运行时进入运行时常量池中。运行时常量池相对于Class常量池具备了动态性，也就是并非预置在Class文件中的常量池才能进入方法区的运行时常量池，运行期间也能将新的常量放入运行时常量池中，典型的是String.intern()方法。运行时常量池收方法区的约束，当无法再申请到足够的内存后会抛出OutOfMemoryError异常。

### 直接内存

直接内存并不是java虚拟机所管理的一块内存区域，有与jdk1.4之后引入了NIO（基于通道的IO）它可以使Native函数直接分配堆外内存。此区域收物理内存的影响，当内存不足的时候同样会出现OutOfMemoryError异常。

## 什么是对象

对象是存储再java堆中的Class实例。

### 对象的创建

java是一门面向对象的语言，在程序运行的过称重无时无刻不有对象创建出来，可以通过 new 关键字，反序列化，反射等。那在虚拟机底层是如何创建对象的呢？

虚拟机遇到一个new 指令时会去检查这个指令的参数是否在常量池中定位到一个符号引用并且检查这个符号引用的类是否已经加载解析初始化过，如果没有那就先进行类的加载。在类加载验证解析通过后会为新创建的对象分配内存空间，对象所需的内存空间在类加载完成的时候已经完全确定的了。堆空间的划分主要有两种方式 指针碰撞和空闲列表。指针碰撞将指针移向空闲的一部分。空闲列表是 维护一个列表记录那些内存是可以使用的。选择哪种方式是由java堆是否完整决定的，而java堆是否完整是由采用何种垃圾收集算法决定的。

内存分配完成之后会初始化内存，接下来虚拟机对对象进行必要的设置，例如对象属于哪个实例，如何找到对象的元数据，对象的hash码对象的GC分代信息等。这些信息都存储在对象头之中。根据虚拟机运行的状态是否启动偏向锁，对象头会有不同的设置方法。

### 对象的内存布局

在hotSpot虚拟机中对象被分为三部分，对象头，实例数据和对其填充。

Hotspot 虚拟机的对象头被分为两部分，第一部分存储对象自身数据包括hash码GC分代、锁状态、线程持有的锁、偏向线程id、偏向时间戳。官方称作Mark word.

表-Hotspot虚拟机对象头。

|  |  |  |
| --- | --- | --- |
| 存储内容 | 标志位 | 存储状态 |
| 对象的hash码对象的分代信息 | 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标志 |
| 偏向线程id 偏向时间戳 分代信息 | 01 | 可偏向 |

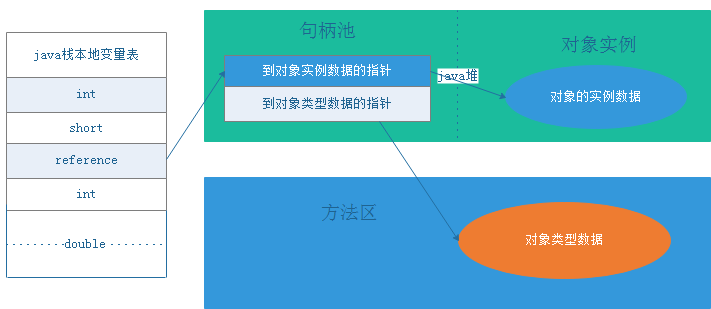
对象头的另一部分是类型指针，即对象指向它类元数据的指针，通过此指针访问对象的实例数据。如果对象是java数组，则对象头中还应包含记录数组长度的信息。普通java对象的元数据可以确定对象的大小 数组不可以。

### 对象的定位

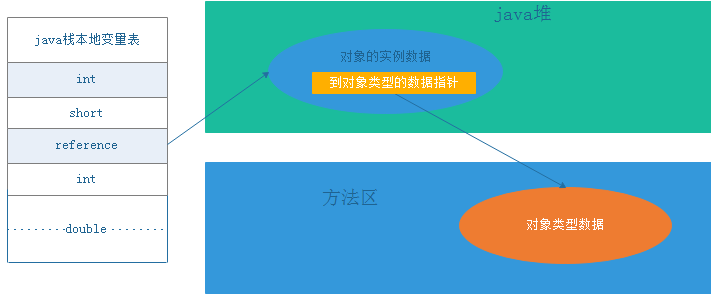
建立对象是为了使用对象，我们java程序需要通过栈上的reference引用来操作堆上的对象。由于reference只是持有引用并没有具体说明怎么找到独享的实例的，目前主流的有两种一种是对象指针另外一种是对象句柄。

如果使用的是对象的句柄 那reference中存储的就是对象句柄的地址，如果采用的是对象指针那reference存储的就是对象的指针。

对象句柄的访问方式



使用对象指针的访问方式



这两种对象的访问方式各有优势，句柄访问方式主要是稳定，在对象被移动时只会改变句柄中对象指针。对象指针的访问方式主要是提高效率减少了一次对象的寻址。

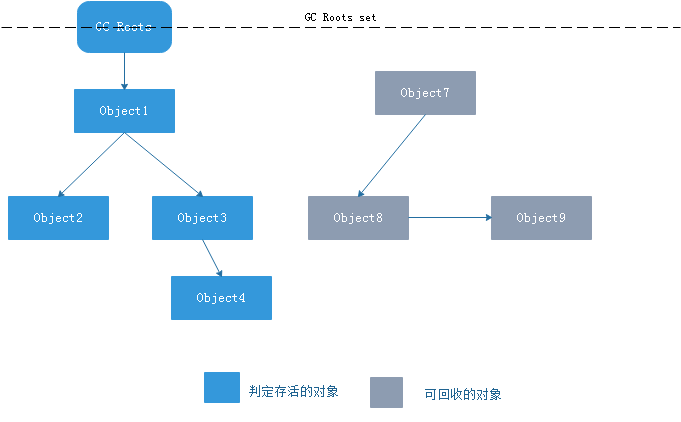
# 垃圾收集器与内存分配

java垃圾收集主要发生在堆和方法区中。这里要思考有三个问题，回收什么内存，什么时候开始回收，怎么回收。

### 回收堆区

首先来看对象的生存状态，对象从创建开始直至死亡发生了怎样的流程，如何判断一个对象是否或者是进行垃圾收集的前提条件。目前主流的判定分析法有，可达性分析法和引用计数法。

引用计数法的实现简单，判定效率也很高，在大部分情况下都是一个不错的算法，但是他不能解决对象之间的相互依赖问题。可达性分析法的主要思路是通过一系列成为GC Roots的对象座位起始点，从这个节点开始向下搜索，搜索所走过的路径成为引用链，当一个对象到GC Roots没有任何引用链项链时则证明此对象是不可用的。



在java语言中可做为GC Roots的对象包括以下几种

虚拟机栈（栈帧中的局部变量表）中持有的引用

方法区中静态属性引用的对象

方法区中常量引用的对象

本地方法栈中JNI（一般说的native方法）引用的对象

在jdk1.2之后将引用划分为 强引用、弱引用、软引用、虚引用。

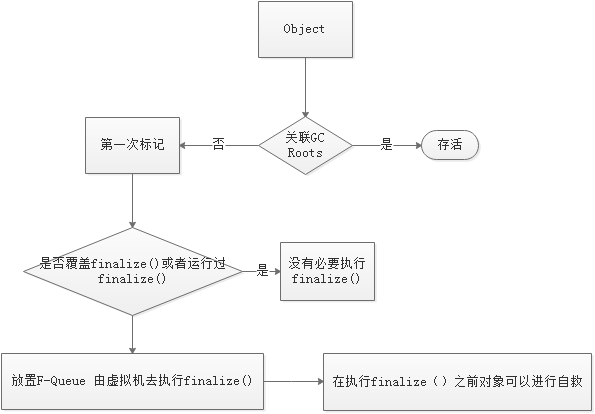
强引用在程序中是普遍存在的例如Object obj = new Object();只要强引用还在垃圾回收器就不会回收他们。

软引用是用来藐视一些还有用但并非必要的对象，对于软引用关联着的对象，在系统发生内存溢出前将会把这些对象列为回收范围之中进行第二次回收。如果这次回收还没有足够的内存才会抛出内存溢出的异常。在1.2之后提供了SoftReferrnce类来实现软引用。

弱引用也是用来描述非必须对象的额，但是他的强度比软引用还要弱一些，被弱引用关联的对象只能生存到下次垃圾收集发生之前，当垃圾收集器工作时候，无论房钱内存是否足厚，都会回收掉只被弱引用关联的独享。在jdk1.2之后提供了WeakReference类来实现弱引用。

虚引用也成幽灵引用或者幻影引用，他是最弱的一种引用关系。一个对象是否有虚引用的存在完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象的实例。

即使是可达性分析法判定是不可达的对象也不一定是非死不可的，他们暂时处于缓刑阶段



### 回收方法区

方法区回收效率远低于堆区的回收，此区域主要回收常量信息、类，接口等相关信息，此区域的判定方法也是比较复杂的

判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。

类需要同时满足下面3个条件才能算是“无用的类”：

该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例。

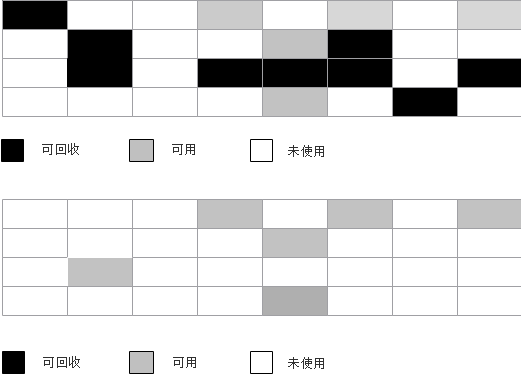
加载该类的ClassLoader已经被回收。

该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法

### 垃圾收集算法

#### 标记清除算法

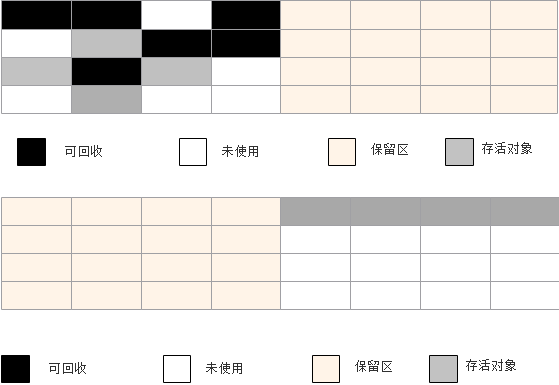
最基础的算法，分为标记和清除两部分，首先标记出所有要回收的对象，然后根基标记的信息进行统一的回收。此算法有两处不好一是效率问题，标记和清除两个阶段的效率都不高，二是清除过后会产生大量的内存碎片。



#### 复制算法

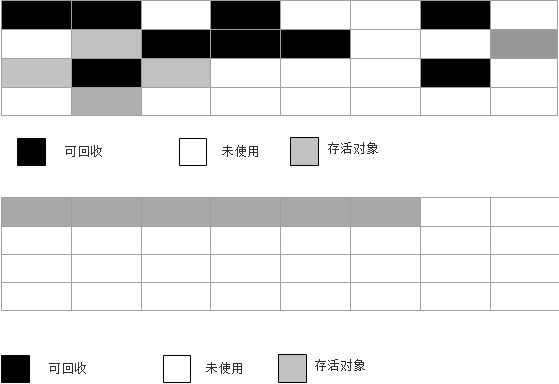
为了解决效率问题，一种称为复制的收集器算法出现了，他将可用内存按照容量划分为大小相等的两块，每次只使用其中的一块。当这一块内存用完了就将还存活的对象复制到另外一块上面，然后一次性的把使用过后的内存清理掉。这样每次得到的都是完整的内存空间。只是这种代价就是每次只能用一半的内存。

新生代中对象98%都是朝生夕死的所以内存划分没有必要按照1:1的的方式进行。而是将内存划分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块SURvivor当回回收时



#### 标记整理

复制收集算法在对象存活率较高时就要进行较多的复制操作，效率会降低，更关键的是如果不想浪费50%的内存就需要有额外的空间进行担保，以免应对被使用的内次中所有对象都100%的存活。所以在老年代一般不能采用这种算法。根据老年代的特点，提出了另外一种算法 标记整理算法，标记过程和标记清除算法一样，但是后期不是对可回收的对象进行清除，而是将所有存活的对象移向一段没然后直接清除掉便捷意外的内存。



#### 分代收集算法

当前商业虚拟机都采用分代收集，它将内存划分为几块，一般java中划分为新生代和老年代，这样根据分代信息合理采用不同的收集方式。新生代中的对象一般都是朝生夕死的所以采用复制算法，老年代中对象稳定采用标记整理算法。