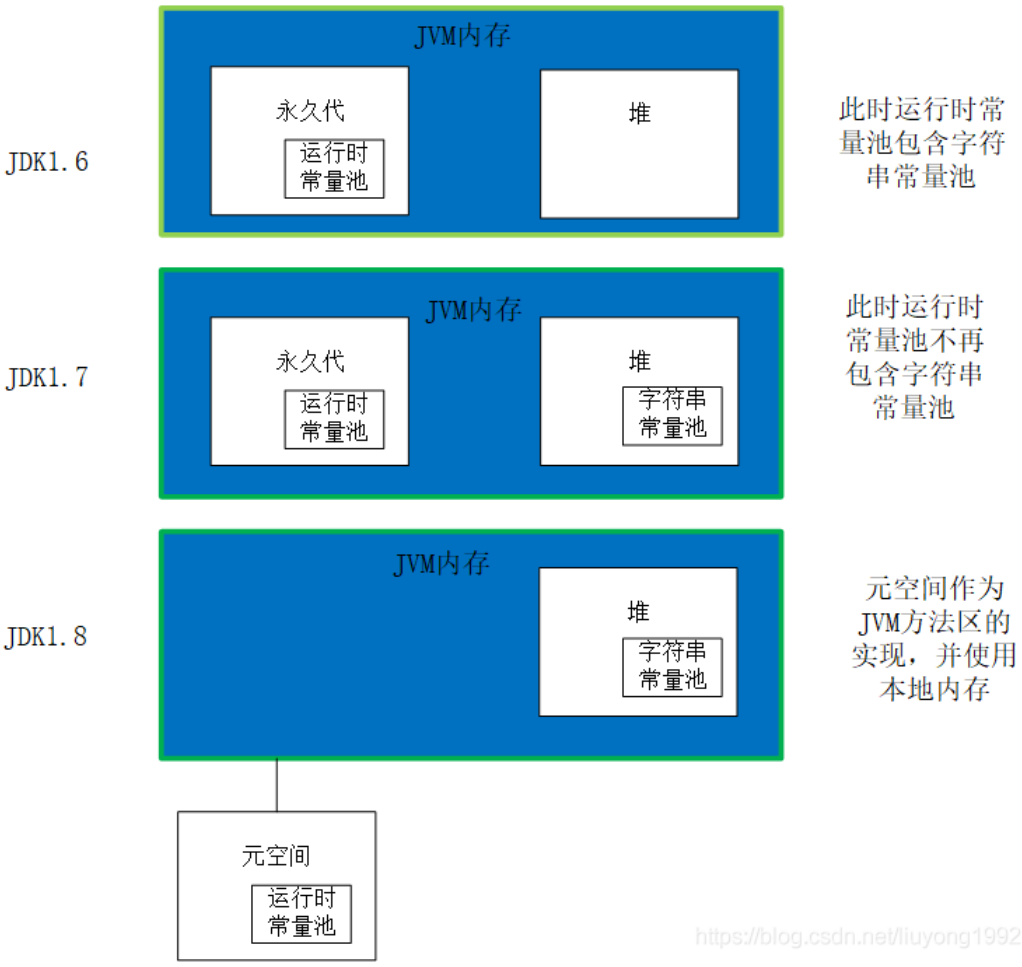
Jvm高级特性阅读笔记

Java内存模型

- 程序计数器：保存当前执行的字节码指令的地址
- Java虚拟机栈：保存局部变量表 操作数 对象连接 方法返回信息
- 本地方法栈：jvm规范并未强制规定，各个虚拟机自行实现 旨在为Java虚拟机提供本地方法服务
- 堆：保存对象
- 方法区：class文件中 类信息 静态变量 常量 JIT（即时编译器）生成的代码
- 运行时常量池： class文件 编译期生成的字面量 符号引用
- 直接内存：引入NIO后在 基于管道和缓冲区的IO 通过DirectByteBuffer 对象可以操作Java内存模型外的内存，但是由于物理内存限制 这部分内存的使用和Java内存模型中的内存使用 还有元空间之间是相互竞争的关系，部分jvm调优就是在限制这部分内存的各自分配。

> 在jdk1.7以前 运行时常量池是在方法区中，之后移动到堆中 因为之前的方法区被称为永久代 顾名思义 这种实现方式简单粗暴 也确实造成了严重的bug 永久代内存空间不足导致OOM 或内存泄漏，所以从1.7开始HotSpot 逐步改造永久代的实现，在1.7版本移动了运行时常量池，到1.8时将永久代彻底改造为在本地内存中实现的“元空间”可以使用-XX:MaxMetaspaceSize
> 控制原空间大小 默认是没有限制 但是在书中提到的运行时常量池理论上 还是属于方法区的一部分，只是在实现时，存入了堆中



对象的创建

首先要验证方法区的运行时常量池中是否存在这个类的符号引用，如果存在则检查这个符号引用的对象是否被加载 解析 初始化

第二步 为该对象分配堆内存空间 ， 有两种方式：指针碰撞和空闲列表
基于这两种方式对指针的使用都非常频繁，在并发条件下无法避免内存泄漏的可能，所以使用CAS+失败重试 或 TLAB的方式 的方式保证操作的原子性
其中TLAB在分配空间时 已将空间对象初始化为零值 提高了分配效率
接下来对对象进行必要的设置 对象的元数据信息，类的实例信息，hashcode，对象的GC分代年龄，是否启用偏向锁等都在对象头中设置
此时 new的动作才刚执行到构造函数 ClassFile中的 <init>方法还没有开始执行
init执行后对象才算生成。
对象的内存结构包括 对象头 实例数据 Padding
其中Padding无实际意义，只用于占位 保证对象长度为8字节的整倍数。

对象的访问定位

栈中的连接信息即对象的引用
指针定位：引用中包含对象在堆中的指针，对象中又包含对象类型数据的指针 指向方法区中对象类型数据
句柄定位：引用中包含句柄指针，堆中的句柄池中包含对象实例数据的指针 对象类型数据的指针 分别指向堆中的对象实例 方法区中对象类型数据

使用句柄定位的好处是 在对象进行移动时 只需要改动句柄池中的指针即可

使用指针定位的好处是访问速度要比句柄定位快 由于对象的频繁访问这种特性 java虚拟机主流使用指针定位

垃圾收集器和内存分配策略

Java 1.2后 进行引用的概念扩充
强、软、弱、虚 分别对应 不回收，溢出前回收，下次GC回收，回收提醒

对于垃圾回收 判断哪种对象需要收集判断出现两种算法
引用计数器算法和可达性分析算法
对象在被标记为可回收时 也不一定会被回收
原因出在finalize()方法，这是个优先级很低的方法，在使用System.gc()方法后执行。
如果出现
finalize方法被重写 或者 重写的finalize方法已经被执行过 则正常执行GC
否则 该对象进入F-queue 在F-queue中GC会不阻塞地进行二次标记，如果挂上了引用 就可以移出F-queue 重新进入堆中，否则正常执行GC

方法区的回收

方法区中类型信息对应的堆中的实例已被回收
类加载器已被回收
如果是反射实例没有引用指针的情况下
满足这三个条件后 方法区的类型信息被允许回收，但不类似堆中 满足条件后必定会被回收，具体可通过VM arg配置 例如 -Xnoclassgc

分代收集理论

建立在两个分代假说下

> 弱分代假说 大部分对象朝生夕灭 和强分代假说 逃过越多次GC的对象越难被GC

对应这新生代 老年代
分代收集实现中发现出现跨代引用问题
随后提出了跨代引用假说
即

> 跨代引用相对同代引用是极少数

当出现跨代引用，新生代由于有老年代的引用而无法被收集，而成功晋级老年代，此时跨代引用问题就消除了，所以需要在新生代创建一个全局数据结构 记忆集（Remembered Set）这个结构将存在跨代引用的老年代对象标记出来，下次进行minor GC 时 直接扫描标记的老年代对象。
前面的 引用计数算法和可达性分析算法旨在进行垃圾标记
接下来要说的是垃圾收集算法 主要分4种

1. 标记-清除：标记后直接删除
2. 标记-复制：直接删除会又内存碎片化出现，所以将内存1：1分开，将非标记对象移动过去后统一删除整个内存空间，又因为这种方式浪费了一半的内存空间，所以出现了Appel式回收，8：1：1，其中8是Eden，两个1是Survivor，每次GC都将Survivor复制到另一个空白空间后将Eden和Survivor清除，但也有特殊情况，当MinorGC存货的对象过大 Survivor无法容纳，可以通过内存分配担保的方式进入老年代。
3. 标记-整理（压缩）：将所有存货的对象移动到内存空间的一端，然后直接清除边界以外的内存

4. 1，3混用 空闲时用1，碎片化影响内存分配时用3