类加载过程：

1 类变量初始化 ：准备阶段赋默认值，初始化阶段执行clinit方法（两种赋值方式：第一显示赋值，第二静态代码块赋值，这两种方法最终会变成clinit方法）

2 成员变量初始化:(new 执行init方法 栈指针指向，会触发DCL问题)（三种赋值方式：第一显示赋值，第二代码块赋值，第三构造器赋值，这三种方法最终会变成init方法）

1 指针碰撞（CMS外的垃圾收集器使用）

2 空闲列表（CMS）

1 jvm分为四个部分：运行时数据区，类加载系统和字节码文件，垃圾回收，jvm调优

2 运行时数据区：pc寄存器、虚拟机栈、本地方法栈、堆、方法区

2.1 pc寄存器（线程私有、不会oom，不会gc）存储的是下一个要执行的指令的位置

2.2虚拟机栈（线程私有、会oom，不会gc）

局部变量表：底层是数组结构，用来存储局部变量值（基本变量、对象引用等），在编译期间就已经确定下来大小。非静态方法存储的第一个位置是this变量。

操作数栈：底层是数据结构，但是当成栈来使用，只有入栈和出栈两种操作。存储的也是需要操作的变量值（基本变量、对象引用），在编译期间就已经确定下来大小。

动态链接：指向运行时常量池中方法的位置（表明当前方法是那个对象的什么方法）

符号引用：常量池中的字符串的字面量，java/lang/Object

直接引用：这个Object对象在内存中的位置0X00000001

非虚方法（虚方法）：编译期间就确定下来的方法就是非虚方法，如静态方法、final方法、private方法、构造器方法、父类方法

invokeStatic:执行静态方法

invokeSpecial：执行非虚方法（final方法除外）

invokeVirtual:执行虚方法(包含final方法)

invokeInterface:执行接口方法

虚方法表：存在方法区中（不用再循环父类找虚方法）

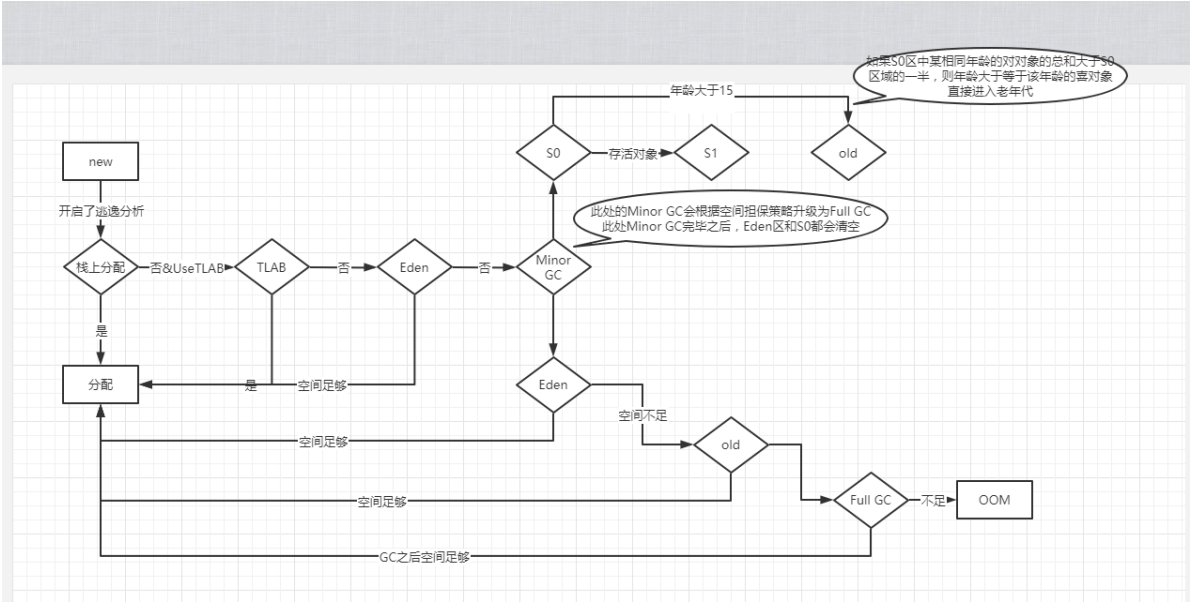
方法返回地址：保存的是方法调用的下一个指令地址，为了使方法正常退出时，可以知道返回到那个位置。

正常返回：需要恢复上层方法的局部变量表，操作数栈，吧返回值（如果有的话）压入调用者的操作数栈中。

异常返回：通过异常表来确定如何处理

2.3 本地方法栈（线程私有、会oom，不会gc）

2.4 堆（线程共享、会oom，会gc）



堆分为：新生代和老年代，新生代包含1个Eden和2个Survivor

Minor GC:eden区满了之后&&满足空间担保，S0或者S1满并不会触发，Minor GC之后会清空Eden区和一个Survivor

Full GC:老年代满了、方法区满了、Eden区满了之后，不满足空间担保。

逃逸分析技术和标量替换：有了逃逸分析技术才有了栈上分配、锁消除（锁粗化）、标量替换。hotspot并没有支持逃逸分析技术。

2.5 方法区（线程共享、会oom，会gc）

方法区：类信息（域信息、方法信息）、常量、静态变量、jit及时编译的代码

类信息：类的全名称、修饰符、父类、接口列表

域信息：域名称、修饰符、类型

方法信息：方法名、返回类型、参数列表、修饰符、字节码、异常列表(init\clinit)

常量：数值类型的常量池（字符串类型的常量池1.6位于方法区，1.7迁移到堆中）

常量池包含：字面量和符号引用（类和接口的全限定名、字段的名称和描述赋、方法的名称和描述赋），运行时常量池：字面量和直接引用，1.7之后字符串常量池和类变量都在堆中

每个类都对应一个运行时常量池

静态变量：也位于方法区（final修饰的static变量（非引用类型）是常量，在编译期间就直接赋值）,这里说的静态变量是=号左边的值。（思考下，静态变量、实例变量、局部变量存储的位置-这三个均指=左边的值，=右边的值均是在堆中）静态变量在1.7之后是跟class对象在一起的

2.5 字符串常量池：

对象初始化的方式：

1 new(xxxBuilder\xxxFactory)

2 clone(深copy和浅copy)

3 class.newInstance(只能调用空参构造方法、public权限)

4 Constructor.newInstance（无限制）

5 反序列化（文件、网络）

创建对象的步骤

1 判断对象的类信息是否加载：加载、链接（验证、准备、解析）、初始化

2 为对象分配内存：内存规整-指针碰撞(非cms)、内存不规整-空闲列表（cms）

3 处理并发安全问题：一个是cas+失败重试，一个是TLAB

4 初始化分配到的空间-给实例变量赋默认值

5 设置对象头

6 执行init方法进行初始化

对象的内存布局

对象头：Mark Word（8个字节，hashCode\GC信息\锁信息）和Klass pointer(开启指针压缩4个字节)，如果是数组，还会记录数组的长度

实例数据：（Interger 4个字节，Long、Double8个字节、引用类型4个字节）

对其填充：至少是8的倍数

对象访问的方式：

句柄访问：句柄池

直接指针：

垃圾回收：

对象标记阶段算法：引用计数和根搜索算法，jvm采用根搜索算法，那些可以作为gc-roots,在标记阶段需要stw，并且需要safepoint，对象是垃圾是在对象头中标示，在对象被标示为垃圾之后，还可以调用finalize()（只会调用一次）方法复活（放入finalize-queue队列中，被低优先级的finalizer线程处理）。

对象被回收经历两个阶段：第一个是gc-roots找不到该对象，第二个是finalize()方法

gc和对象头的关系（标记成11？）

垃圾标记算法：

1 引用计数算法：对象内部存储一个计数的属性，优点：效率高，缺点：无法解决循环引用问题

2 根搜索算法：（jvm采用的）

那些可以作为gc-roots:1 jvm栈局部变量表中指针指向的对象 2 类的静态变量指向的对象 3 本地方法栈中指针指向的对象 4 sync的对象

5 系统类：rt.jar中的对象 6 没有结束的线程 7 常量指向的对象(如果是新生代垃圾回收，老年代的对象也可以是gc-roots？此处老年代有大量对象，该如何处理？)

根搜索算法：在多线程环境下会存在问题，需要stw，但是stw需要safePoint，gc-roots太多，需要oopmap

引入 SafePoint 的原因不是因为 HotSpot 采用了可达性分析，而是因为使用了准确式垃圾收集算法和 OopMap 结构，准确式GC要求jvm必须清楚的知道内存中哪些位置存放了对象引用，因此 HotSpot 采用了一种策略，那就是在外部用一数据结构来记录对象引用的位置，OopMap 就是这样的一个数据结构。但是，程序运行期间，很多指令都是有可能修改引用关系的，即要修改OopMap。如果碰到就修改，那代价也太大了，故而引入了 SafePoint，只在 SafePoint 才会对 OopMap 做一个统一的跟新。这也使得，只有 SafePoint 处 OopMap 是一定准确的，因此只能在 SafePoint 处进行 GC 行为。

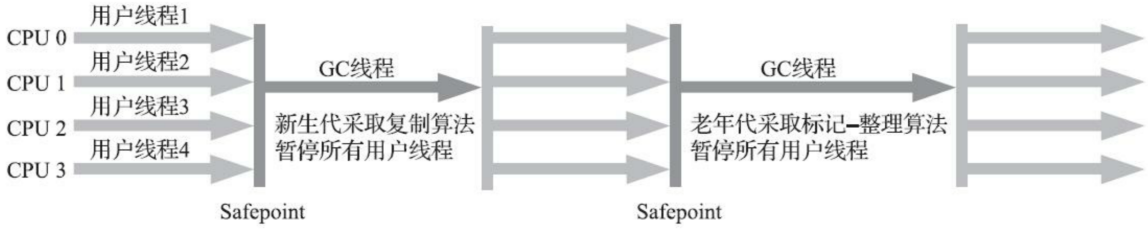
至于 SafeRegion，理解起来很简单。如果一段代码中，没有任何指令会去修改引用关系，那么这段代码运行期间任何时刻进行 GC 行为都是安全地。之所以要引入 SafeRegion，如果线程长期处于 Sleep 状态，而无法到 SafePoint，这使得 GC 无法回收该线程在堆中产生的垃圾。而有了 SafeRegion ，这种情况就迎刃而解了。

安全点的选择不能太少，否则GC等待时间太长；也不能太多，否则会增大运行负荷

根搜索算法：oopmap\safePoint\card table\Rset

对象被垃圾收集器回收之前，还可以调用finalize()方法，进行资源的释放

1 serial&serial old垃圾收集器



2 parnew-serial

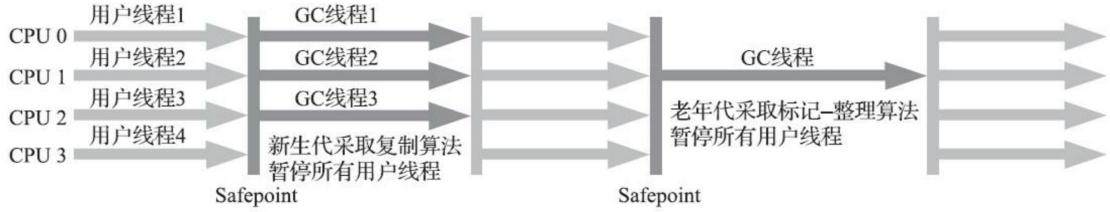


图3-8　ParNew/Serial Old收集器运行示意图

3 parallel scavenge-parallel old

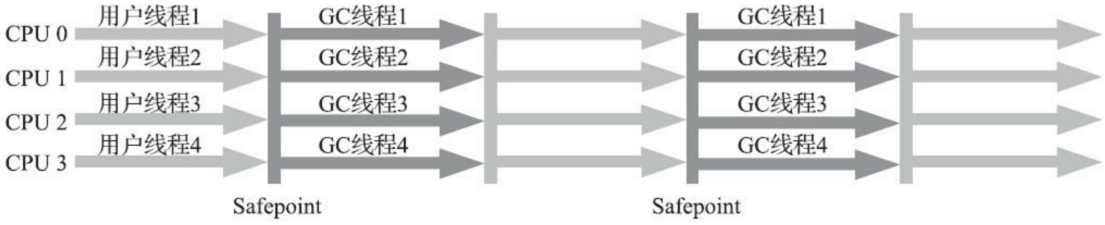


图3-10　Parallel Scavenge/Parallel Old收集器运行示意图

4 cms

图3-11　Concurrent Mark Sweep收集器运行示意图

5 g1

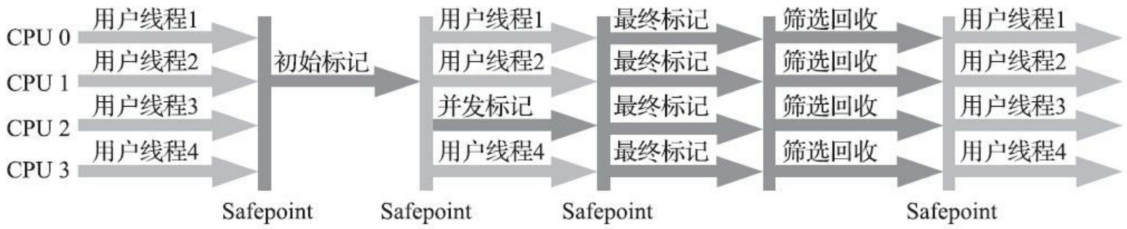


图3-13　G1收集器运行示意图

3 Object obj = new Object();底层的字节码指令

new #11 <java/lang/Object> (在内存中给对象分配空间，并把对象的地址入栈)

dup （把对象的地址copy一份入栈）

invokeSpecial #12 <java/lang/Object.> （执行object的构造方法）

astore_1 （把操作数栈中的数据存储到局部变量表的第二个位置）

return （方法返回）

执行引擎：

1 解释器：直接运行，速度慢

2 jit编译器：需要编译时间，速度快

高级语言->汇编语言->机器指令（机器码）

3 垃圾收集器

String相关问题：

两种创建方式

String s = "a";

String s = new String("s");

String是final的，1.8底层是final的char[], 1.9底层是final的byte[]. (思考为什么)

字符串常量池中不能存储相同的数据，底层是一个hashTable,1.6中大小是1009,1.7之后60013，1009是最小值，可以使用-XX:StringTableSize设置大小

8中基本类型和String类型都有常量池

String常量池为什么要移动：1 永久代大小比较小 2 垃圾回收频率低

字符串拼接操作：

1 常量（final字符串的变量是常量）与常量的拼接-在编译期优化。只要一个其中一个是变量，结果就在堆中

2 常量池中不会存在相同内容的常量

3 变量拼接的原理是StringBuilder

4 调用intern()方法，则将字符串加入常量池，并返回对象的地址

变量字符串+操作，底层的原理是StringBuilder.appand,StringBuilder.toString()->new String();

new String("abc")有几个对象：看字节码，一个是new关键字，一个是ldc

new String("a") + new String("b")有几个对象：一个stringbuilder,一个string,一个a，一个string,一个b，一个toString方法返回new String()，又创建了一个String对象。

String.intern()方法在1.7之后，随着字符串常量池的改变而改变。

Cms gc调优

https://tech.meituan.com/2019/12/05/aqs-theory-and-apply.html

java分析问题命令行工具

https://www.cnblogs.com/duanxz/p/4515437.html

jvm常用指令

arthas(阿里巴巴开源软件，可用于生产)

jps

jstat

jstack

jmap

javap

jinfo

jconsole (图形工具-不能用于生产)

jvisualvm(图形工具-不能用于生产)

-verbose:gc

-verbose:class

-XX:+TraceClassLoading

-XX:+TraceClassUnLoading

-XX:+PrintStringTableStatistics

-Xms

-Xmx

-Xmn

-Xss

-XX:StringTableSize

-XX:NewRatio

-XX:SurvivorRatio

-XX:PrintGCDetails

-XX:PrintFlagsInitial

-XX:PrintFlagsFinal

-XX:useTLAB

-XX:useG1(别的垃圾收集器)

-XX:MaxTenuringThreshold

-XX:PermSize

-XX:MaxPermSize

-XX:MetaspaceSize

-XX:MaxMetaspaceSize

Parallel常用参数：

-XX:SurvivorRatio

-XX:PreTenureSizeThreshold

-XX:MaxTenuringThreshold

-XX:ParallelGCThreads(并行收集线程数一般和CPU核数一致)

-XX:+UseAdaptiveSizePolicy

CMS常用参数:

-XX:+UseConcMarkSweepGC

-XX:ParallelsGCThreads

-XX:CMSInitiatingOccpancyFranction(使用多少比例的老年代之后开始CMS收集，默认是68%，如果频繁发生Serialold应该调小这个值)

-XX:UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction -XX:ConcGCThreads

-XX:MaxGCPauseMillis -XX:GCTimeRatio

G1常用参数:

-XX:+UseG1GC

-XX:MaxGCPauseMillis

-XX:G1HeapRegionSize(1M-32M,2的N次幂),Region数量在2048左右

-XX:G1NewSizePercent(默认是5%)

-XX:G1MaxNewSizePercent(新生代最大比例60%)

-XX:GCTimeRatio

-XX:ConcGCThreads

-XX:InitiatingHeapOccupancyPercent

JVM调优

1 从需要出发开始规划（比如访问量是多少，需要多大内存空间，多少CPU，使用什么垃圾回收器）

2 优化JVM运行环境（慢、卡顿现象）

3 解决OOM问题

如果别人要问怎么选择最合适的垃圾收集策略：具体问题，具体分析。压力测试，调试参数

一般是运维人员通知CPU或者内存使用过高

jps定位java进程

jstack pid |more(查看死锁，查看那个线程CUP占用比较高)

jstack定位具体线程，重点关注Waiting Bolcked

Waiting on <0X111111> (a java.lang.Object)

假如有一个进程中有100个线程，很多线程都在waiting on xxx,一定找到那个线程持有这个锁

怎么找？搜索jstack dump信息，找xxx,看那个线程持有这个锁Running

面试官问线上如何定位问题：不能说使用图形界面（公司自己研发的可以说），最好说命令行或者arthas

图形化界面可以说是在测试环境或者压测环境使用

jstat -gc

jmap -histo 4566|head -20(查看JVM中使用最多的20个对象)--生产不能随便执行

jmap -dump:format=b,file=xx pid:线程系统不能执行

1 设定了heapdump，不专业，因为会有运维预警

2 很多服务器，高可用，停了这个不影响

3 压力测试环境可以说

1 系统CPU经常100%，如何调优？

CPU100%一定是有线程占用系统资源

1 找出进程CPU使用最高的top

2 该进程中那个线程使用CPU最高 top -Hp

3 到处该线程的堆栈信息jstack

4 查找那个线程方法消耗时间jstack

5 工作线程占比高|垃圾回收线程占比高

2 系统内存飙高，如何查找问题

1 到处堆内存jmap

2 分析ivisualvm\mat\jprofiler

3 如果生产服务器频繁FGC如何定位解决

jps -mvl

jinfo pid查看JVM信息（可以查看启动参数）

jinfo -flag xx pid

arthas:

1 dashboard(可以查看线程占用CPU的情况，分析是业务线程或者是垃圾回收线程占用比例较高（可以使用top+jps+jstack定位），具体处理，可以查看堆的使用详情)

2 jvm

3 thread(查看所有的线程)

4 thread xxx(threadid)

5 thread -b查看死锁

6 heapdump--生产不能随便执行

问题一：FASTJSON使用不当导致内存溢出问题

场景：接口对外输出数据时候，输出JSON格式数据，并且要求时间格式是YYYY-MM-DD，但是默认时间的格式返回的是毫秒数

错误写法：

SerializeConfig config = new SerializeConfig();

config.put(Date.class, new SimpleDateFormatSerializer("yyyy-MM-dd HH:mm:ss"));

String res = JSON.toJSONString(srcRes, config);

导致的问题:运维发警报说是机器内存占用异常，直接把服务KILL掉

分析思路：jstate监控各个区域的内存变化情况，发现元空间一致增加，原因是1.7升级到1.8并未设置元空间大小，基本可以锁定是加载了过多的类，但是项目已经运行了一段时间，应该是某个功能动态生成了很多类，所以在启动命令里面加了-verbose:class(此处不能使用jinfo动态增加-XX:+TraceClassLoading，这个值不能动态改变)，程序启动之后在catalina.out文件中发现生成了大量的com.alibaba.fastjson.serializer.ASMSerializer,该类是利用asm技术生成的，每次new就生成一个，最终导致内存溢出。解决办法：单例一个对象

问题二：接口访问超时问题

同事反馈接口访问超时，要到具体信息后发现，不是一个接口超时，很多接口超时，基本判断是应用程序问题（服务器问题运维会报警），通过uav查看接口请求时间的服务器运行情况，发现基本与full gc重合，full gc发生6s左右，接口超时时间一般为3s左右，查看gc情况发现,minor gc频繁发生，一分钟25次，cms gc 1个小时一次（别的服务10多个小时一次），gc前老年代内存占用1.8G左右（服务器配置，2c4g，内存设置3g），gc后300M左右，gc日志中有new threshold1(max 6)（思考为什么）,基本判定是过早晋升导致的问题，解决方案，调整新生代大小，规则，老年代大小为gc后活跃对象的3倍，剩余的给新生代，所以新生代2g，老年代1g，调整后，minor gc一分钟10次，cms gc 10多个小时一次，超时情况发生频率减少。