# 类加载器

# GC

<https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

## java内存划分：

线程共享的：堆区，方法区。

***1.堆区：***存放对象实例，是理解Java GC机制最重要的区域，没有之一。1.8之后删除永久代。

***2.方法区：***用于存储虚拟机加载的：静态变量+常量+类信息+运行时常量池 （类信息：类的版本、字段、方法、接口、构造函数等描述信息 ），默认最小值为16MB，最大值为64MB，可以通过-XX:PermSize 和 -XX:MaxPermSize 参数限制方法区的大小。

线程私有的：虚拟机栈，本地方法栈，程序计数器。

***1.虚拟机栈：***jvm执行的java方法。

***2.本地方法栈：***jvm执行的native方法。HotSpot将虚拟机栈和本地方法栈合并。

***3.程序计数器：***每个线程都有一个程序计算器，就是一个指针，指向方法区中的方法字节码（下一个将要执行的指令代码），由执行引擎读取下一条指令，是一个非常小的内存空间，几乎可以忽略不记。由于java多线程是通过线程轮流切换并分配执行时间来实现的，任何时刻只会执行一条线程中的指令。因此为了切换线程后依旧能恢复到正确的位置，需要一个独立的计数器。

## 执行对象

自动释放不再被引用的对象所占的堆内存，整理碎片空间。

可达性分析：从GC Roots开始向下搜索，没有任何引用链相连时，则证明此对象是不可用的。不可达对象，需要被回收。

GC Roots包括：

a．虚拟机栈中引用的对象。

b．方法区中类静态属性实体引用的对象。

c．方法区中常量引用的对象。

d．本地方法栈中JNI（本地接口）引用的对象。

## 优缺点

优点：可以提高生产效率（减少释放空间的编程时间）和增加程序的健壮性（不再因为错误的释放空间导致程序崩溃）。

缺点：影响程序的性能，因为要动态的监视和释放。在明确的内存不再使用内存比起来，这个活动占用了更多的cpu时间。程序员缺乏安排cpu对释放对象的控制。

## 执行时机

程序调用通知System.gc()。

Minor GC：

当Eden区满后。

Major GC：

调用System.GC；老年代空间不足；方法区空间不足；调用Minor GC后，/移至年老区的对象大小大于可用空间大小；

## 流程

绝大多数刚创建的对象被分配在Eden区，其中的大多数对象很快就会消亡。Eden区是连续的内存空间，因此在其上分配内存极快；

Eden区满，执行Minor GC，清理消亡对象，将剩余的对象复制到一个存活区Survivor0（此时，Survivor1是空白的，两个Survivor总有一个是空白的）；

此后，每次Eden区满了，就执行一次Minor GC，并将剩余的对象都添加到Survivor0；

当Survivor0也满的时候，将其中仍然活着的对象直接复制到Survivor1，以后Eden区执行Minor GC后，就将剩余的对象添加Survivor1（此时，Survivor0是空白的）。

当两个存活区切换了几次（HotSpot虚拟机默认15次，用-XX:MaxTenuringThreshold控制，大于该值进入老年代）之后，仍然存活的对象（其实只有一小部分，比如，我们自己定义的对象），将被复制到老年代。

## Minor GC

使用停止-复制清理法，用于清理年轻代。

JVM会安全的暂停所有正在执行的线程（Stop The World），来回收内存空间，在这个时间内，所有除了回收垃圾的线程外，其他有关JAVA的程序，代码都会静止，反映到系统上，就会出现系统响应大幅度变慢，卡机等状态。

指针碰撞（bump-the-pointer）用于追踪最后创建的一个对象，在创建对象的时候，只需要检查最后一个对象后边是否有足够的内存即可。

TLAB（thread-local-allocation-buffers）对于多线程而言，将eden区分为若干段，每个线程使用独立的一段。

## Major GC

使用标记整理清理法，用于清理年老代。将存活对象全部移动至一端。

## Full GC

清理整个堆空间—包括年轻代和老年代。

## 空配分配担保

在发生minor gc之前，虚拟机会检测 : 老年代最大可用的连续空间>新生代all对象总空间？

1、满足，minor gc是安全的，可以进行minor gc。

2、不满足，虚拟机查看HandlePromotionFailure参数：

（1）为true，允许担保失败，会继续检测老年代最大可用的连续空间>历次晋升到老年代对象的平均大小。若大于，将尝试进行一次minor gc，若失败（应该时gc之后再上述的检测），则重新进行一次full gc。

（2）为false，则不允许冒险，要进行full gc（对老年代进行gc）。

## GC垃圾回收器

Serial（串行）

Parallel / Throughput（并行）---------HotSpot默认使用

CMS（并发）

and the new kid on the block G1（G1）

## OutOfMemoryError

**堆：**

堆达到最大的容量限制，需要分析是内存泄漏还

是内存溢出。（GC Roots的可达性分析）

**方法区：**

分配空间不足，常量池分配不足等就会

java.lang.OutOfMemoryError:PermGen space

**虚拟机栈和本地方法栈：**

如果线程请求的栈深度大于虚拟机所允许的深度，将

抛出java.lang.StackOverflowError。

不断创建线程的方法，在这种情况下，为每个线程分配的内存越大，就会越容易产生OOM异常。

**程序计数器：**唯一不会引发OOM问题。

***OOM:GC overhead limit exceeded***

如果系统大量的时间都在GC（98%）而回收的效果不明显（2% heap空间），就会抛出这个异常。

## 各项参数

<https://www.iteye.com/blog/leichenlei-2097797>

|  |  |
| --- | --- |
| -Xms | 堆初始值（建议：操作系统可用的1/64） |
| -Xmx | 堆最大值（建议：操作系统的1/4） |
| -XX:PermSize | 方法区初始值 |
| -XX:MaxPermSize | 方法区最大值 |
| -XX:newSize | 新生代初始值 |
| -XX:MaxnewSize | 新生代最大值 |
| -Xmn | 同时设置新生代初始值和最大值 |
| -XX:PretenureSizeThreshold | 直接就进入老年代的对象大小 |
| -XX:MaxTenuringThreshold | 新生代进入老年代的“岁数” |
| -XX:SurvivorRatio | eden比率 |
| -XX:+PrintGCDetails | 输出GC时信息 |

# 线程及锁原理

# utp，tcp和websocket

1：UDP是面向报文的，发送方的UDP对应用层交下来的报文，不合并，不拆分，只是在其上面加上首部后就交给了下面的网络层，也就是说无论应用层交给UDP多长的报文，它制统统发送，一次发送一个。而对接收方，接到后直接去除首部，交给上面的应用层就完成任务了。因此，它需要应用层控制报文的大小

2：TCP是面向字节流的，它把上面应用层交下来的数据看成无结构的字节流来发送，知可以想象成流水形式的，发送方TCP会将数据放入“蓄水池”（缓存区），等到可以发送的时候就发送，不能发送就等着，TCP会根据当前网络的拥塞状态来确定每个报文段的道大小。

3：每一条TCP连接只能是点到点的，UDP支持一对一，一对多和多对多的交互通信。

4：TCP是流模式，UDP是数据报模式。

# netty

# mysql树

# hashmap的扩容和红黑树

# redis原理