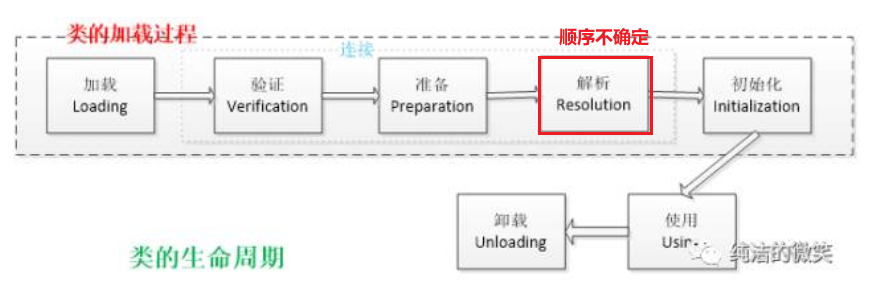
# 类的加载机制

## 什么是类的加载

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个 java.lang.Class对象，用来封装类在方法区内的数据结构；

加载的都是.class文件（来源系统，网络，文件提取，动态编译）

## 类的生命周期



* 加载
  + 获取二进制字节流
  + 静态存储结构转化成方法区的运行时数据结构
  + 堆中生成class对象
* 连接
  + 验证：文件格式，元数据，字节码，符号引用
  + 准备：静态变量分配内存，并将其初始化为默认值（final和static修饰，直接赋值）
  + 解析：把类中的符号引用转换为直接引用
* 初始化

为类的静态变量赋予正确的初始值

* 卸载（结束生命周期：System.exit，正常运行结束，异常终止，虚拟机进程终止）

## 类加载器

* 启动类加载器：JDK\jre\lib中所有.java文件
* 扩展类加载器：JDK\jre\lib\ext中所有.javax文件
* 应用类加载器：加载用户类路径
* 自定义加载器：继承ClassLoader类，重写findClass 方法

## 类加载方式

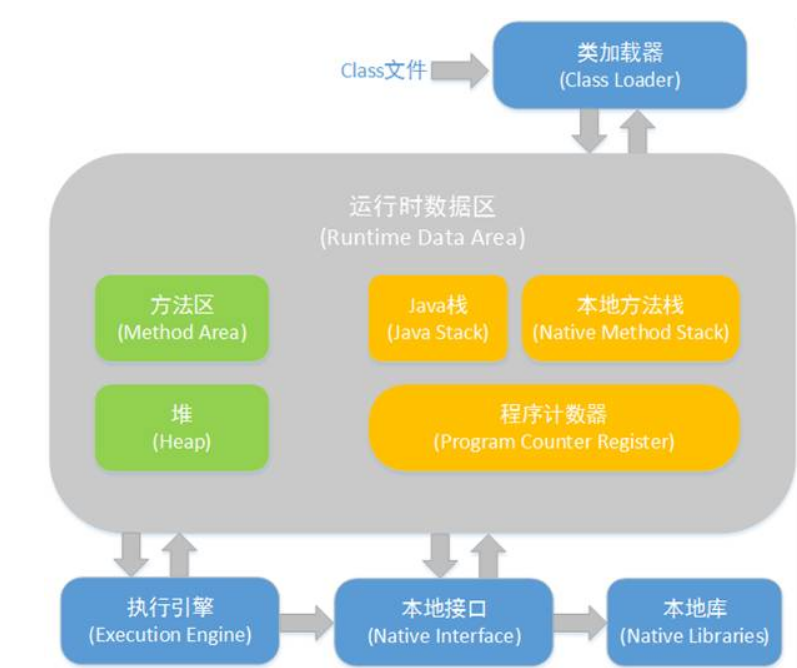
* 由JVM初始化加载
* Class.forName()方法动态加载(解释及执行static)
* ClassLoader.loadClass()动态加载（只将.class文件加载到jvm中，不执行static）

## 双亲委派模型

如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把请求委托给父加载器去完成，如果父类无法加载，则由子类加载

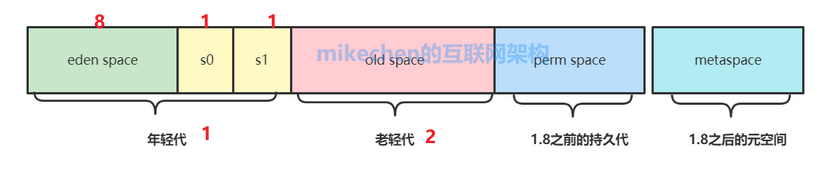
破坏双亲委派机制：自定义类加载，重写loadclass方法

# Jvm内存模型



## 五大内存结构

### Java堆（Heap）



内存中最大的一块内存区域，也是被各个线程共享的内存区域，该内存区域存放了对象实例及数组

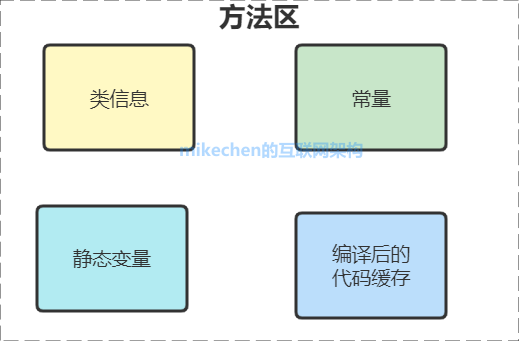
-Xms(最小值)：默认为操作系统物理内存的1/64

-Xmx(最大值)：默认为物理内存的1/4

堆内存空闲<40%,堆内存增大至-Xmx；堆内存空闲>70%,堆内存缩小至-Xms

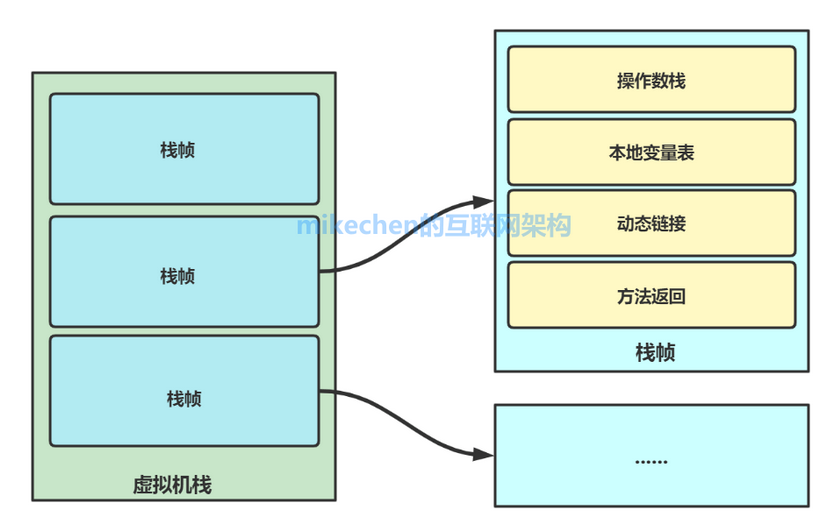
避免在运行时频繁调整Heap的大小，通常-Xms与-Xmx的值设成一样

### 方法区（Method Area）



各个线程共享的内存区域，它用于存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等数据。主要就是存放从class文件里加载进来的类的

### 虚拟机栈（JVM Stacks）



主管Java程序的运行，它保存方法的局部变量、部分结果，并参与方法的调用和返回

每个方法被执行的时候都会创建一个”栈帧”,用于存储局部变量表、操作栈、方法出口等信息

### 本地方法栈（Native Method Stacks）

本地方法栈则是为虚拟机使用到的Native 方法服务

与虚拟机栈一样，本地方法栈区域也会抛出StackOverflowError 和OutOfMemoryError异常

### 程序计数器（Program Counter Register）

记录下一条需要执行的字节码指令

# 垃圾及回收器

## 内存中的垃圾

* 引用计数算法

循环引用的内存对象就成了无法回收的内存。从而引起内存泄露

* 可达性分析算法

GC Roots的对象作为起始点，从这些节点开始向下搜索

GC ROOT

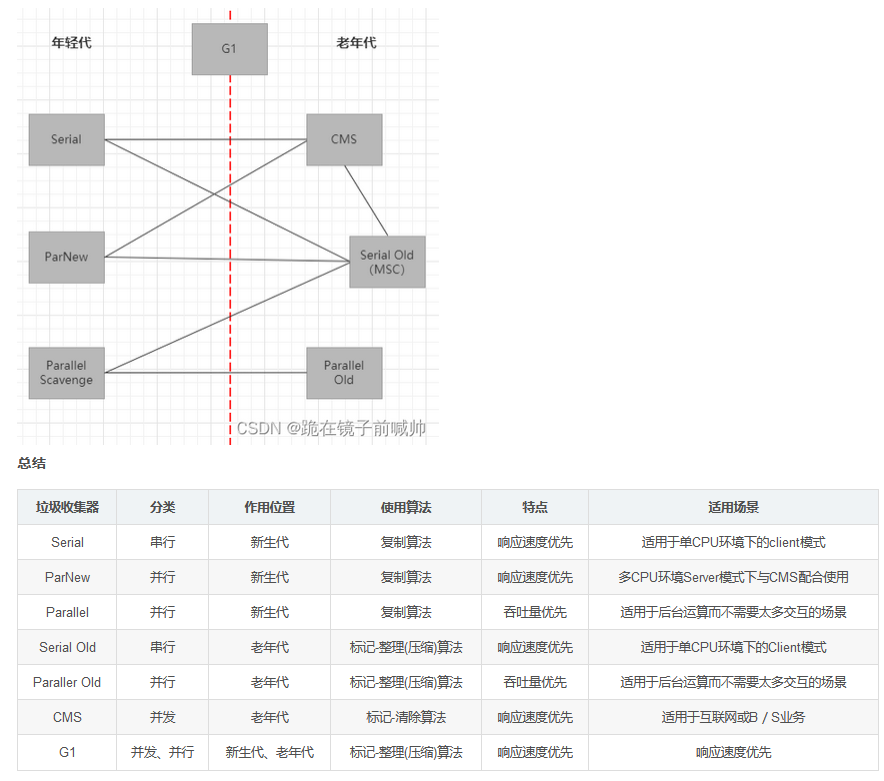
* + 虚拟机栈中引用的对象（栈帧中的本地变量表）
  + 方法区中常量，静态属性引用的对象
  + 本地方法栈中Navtive方法引用的对象

## 回收算法

* 标记 - 清除算法
* 标记 - 整理算法
* 复制算法
* 分代收集算法

## 垃圾回收器

-XX:+PrintCommandLineFlags查看垃圾回收器类型



### CMS垃圾回收过程

用于老年代的回收

* 过程：初始标记（GcRoot直接关联对象,STW）===>并发标记(GcRoot间接关联的对象)===>重新标记(再次判断,STW)===标记清除
* 缺陷：并发操作创建了额外的线程会抢占cpu资源；无法处理浮动垃圾；产生垃圾碎片

### G1垃圾回收过程

G1将Java堆划分为多个大小相等的独立区域,JVM最多可以有2048个Region  
G1有专门分配 大对象的Region叫Humongous区(超过了一个Region大小的50%)

过程：初始标记（GcRoot直接关联对象,STW）===>并发标记(GcRoot间接关联的对象)===>重新标记(再次判断,STW)===> 筛选回收（Cleanup，STW）

筛选回收过程详解

筛选回收阶段首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间(-XX:MaxGCPauseMillis)来制定回收计划

# JVM GC 调优

## 常见内存溢出场景

内存溢出一般分两种，峰值没有限流，瞬间创建大量对象导致内存溢出；程序bug导致内存泄漏，从而导致内存溢出

### Java heap space

* 内存泄漏

Java 应用程序一直持有 Java 对象的引用，导致对象无法被 GC 回收

* 配置问题

JVM 参数指定的堆大小不合适

* finalize 方法的过度使用

finalize，对象被GC之前执行的动作；会存放至引用队列中等待finalize线程执行；但是该线程与主线程相比，优先级较低，就会出现引用队列中的数据越来越多，最终oom

### GC overhead limit exceeded

垃圾收集器一直在运行，但是 GC 效率很低

### Requested array size exceeds VM limit

请求的数组大小超过 JVM 限制

### MetaSpace

JVM 的元空间用尽，则会抛出这个异常；加大 MaxMetaSpaceSize 参数的值

## CPU的问题

CPU的问题最重要的是要找到是哪些线程在消耗 CPU，通过线程栈定位到问题代码

* top
* top -H -p PID
* printf ‘%x/n’ tid
* jstack tid |grep tid 堆栈信息
* jstat -gcutil pid
* jmap -dump:format=b,file=name.dump pid

线程失控情况

* 使用线程池时，最大线程数设置不合理（newCachedThreadPool最大为Int max）
* 线程池的拒绝策略不合理，队列满了仍提交的任务交给提交任务线程处理

## GC 调优策略

### 降低 YGC 频率

时间开销：T1（扫描新生代）和 T2（复制存活对象）

复制对象的成本要远高于扫描成本，单次 YGC 时间更多取决于 GC 后存活对象的数量，而非Eden区的大小，GC 后存活对象的数量多的话，不应该增加年轻代空间（稍微降低对象年龄计数器）

### 降低 Full GC 的频率

由于堆内存空间不足或老年代对象太多，会触发 Full GC，频繁的 Full GC 会

带来上下文切换，增加系统的性能开销

* 减少创建大对象‘’
* 增大堆内存空间

## GC 性能衡量指标

* **吞吐量**

应用程序所花费的时间和系统总运行时间的比值

* **停顿时间**

垃圾收集器正在运行时，应用程序的暂停时间

* **垃圾回收频率**

## JVM调优

### 查看 JVM 堆内存分配

java -XX:+PrintFlagsFinal -version | grep HeapSize