JVM性能调优

JVM内存结构

类加载机制和JDK的调优工具命令

GC日志格式GC调优实战

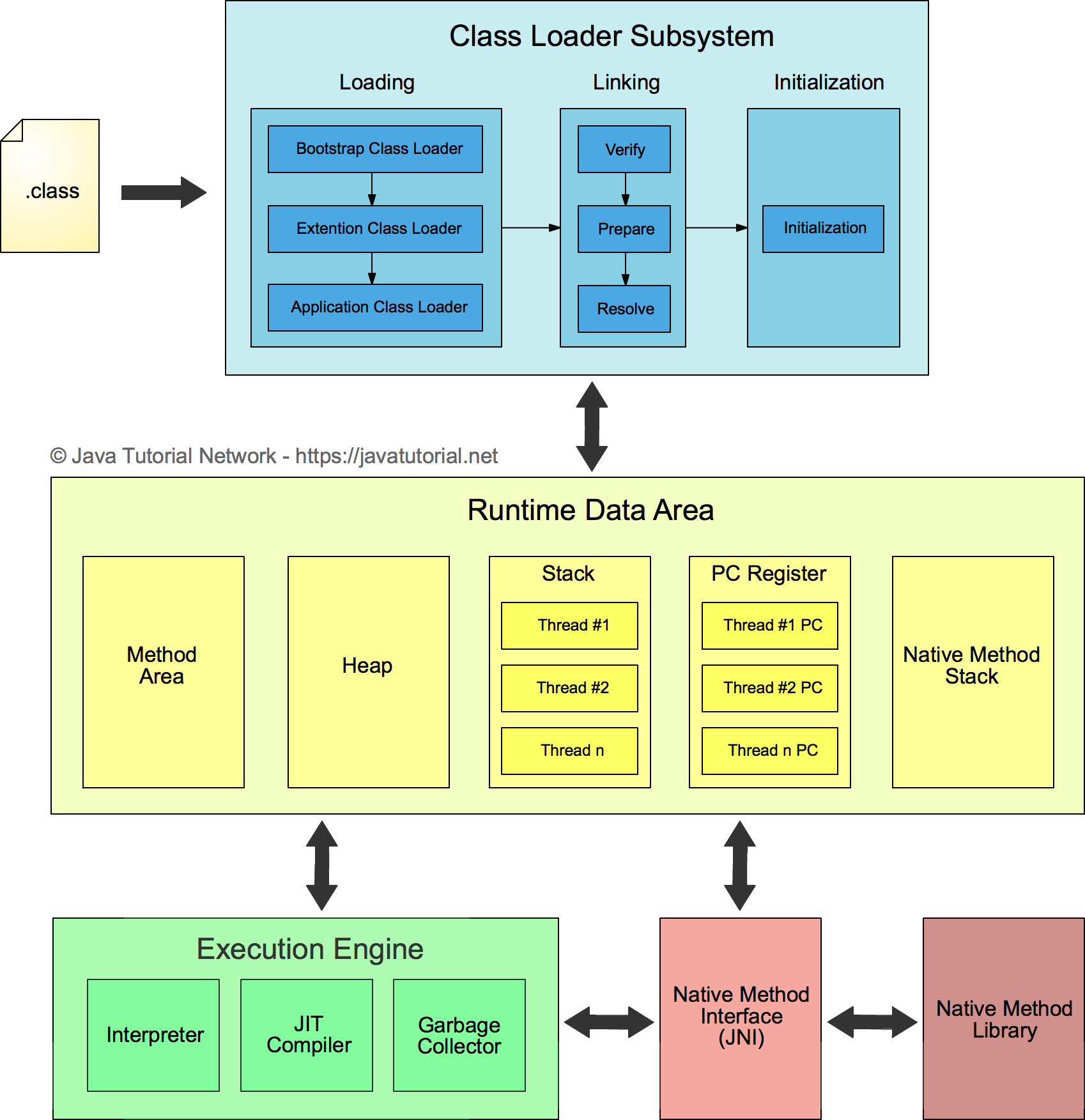
java虚拟机(java virtual machine，JVM)，一种能够运行java字节码的虚拟机。作为一种编程语言的虚拟机，实际上不只是专用于Java语言，只要生成的编译文件匹配JVM对加载编译文件格式要求，任何语言都可以由JVM编译运行。 比如kotlin、scala等。

jvm有很多，不只是Hotspot，还有JRockit、J9等等

**JVM的基本结构**

JVM由三个主要的子系统构成类加载子系统

运行时数据区（内存结构） 执行引擎



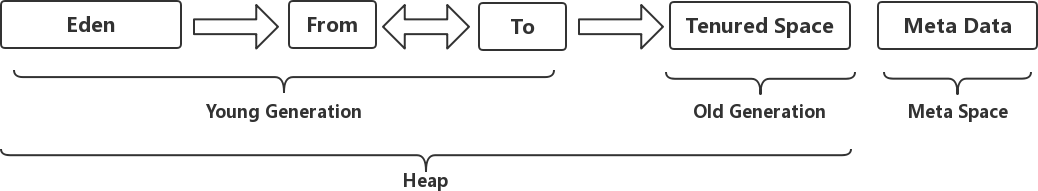
**运行时数据区（内存结构）**

# 方法区（Method Area）

类的所有字段和方法字节码，以及一些特殊方法如构造函数，接口代码也在这里定义。简单来说，所有定义的方法的 信息都保存在该区域，静态变量+常量+类信息（构造方法/接口定义）+运行时常量池都存在方法区中，虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做Non-Heap（非堆），目的应该是为了和Java的堆区分开，JDK1.7之前叫永久代，之后叫元空间

# 堆（Heap）

虚拟机启动时自动分配创建，用于存放对象的实例，几乎所有对象（包括常量池）都在堆上分配内存，当对象无法在 该空间申请到内存是将抛出OutOfMemoryError异常。同时也是垃圾收集器管理的主要区域。



## 新生代（Young Generation）

类出生、成长、消亡的区域，一个类在这里产生，应用，最后被垃圾回收器收集， 结束生命，默认占1/3。

新生代分为两部分：伊甸区（Eden space）和幸存者区（Survivor space），所有的类都是在伊甸区被new出来的。幸存区又分为From和To区。当Eden区的空间用完是，程序又需要创建对象，JVM的垃圾回收器将Eden区进行垃圾回 收（Minor GC），将Eden区中的不再被其它对象引用的对象进行销毁。然后将Eden区中剩余的对象移到From Survivor区。若From Survivor区也满了，再对该区进行垃圾回收，然后移动到To Survivor区。

伊甸区与From区To区占比默认 ：8：1:1

## 老年代（Old Generation）

新生代经过多次GC仍然存货的对象移动到老年区。若老年代也满了，这时候将发生Major GC（也可以叫Full GC）， 进行老年区的内存清理。若老年区执行了Full GC之后发现依然无法进行对象的保存，就会抛出OOM（OutOfMemoryError）异常，默认占2/3。

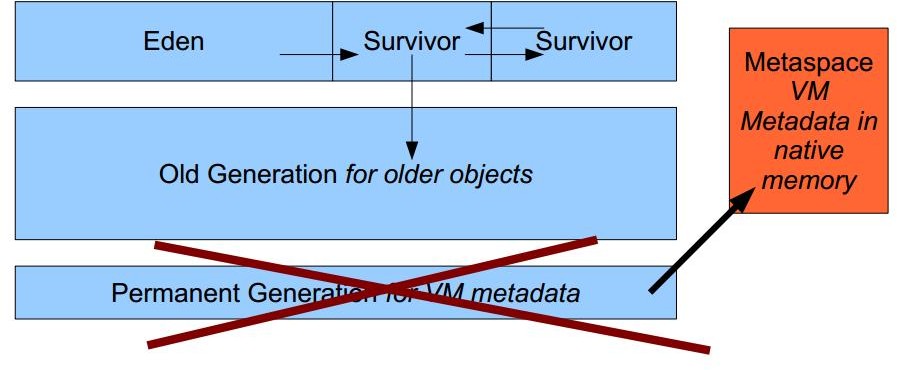
## 元空间（Meta Space）

在JDK1.8之后，元空间替代了永久代，它是对JVM规范中方法区的实现，区别在于元数据区不在虚拟机当中，而是用 的本地内存，永久代在虚拟机当中，永久代逻辑结构上也属于堆，但是物理上不属于、堆外内存。

## 为什么移除了永久代？

参考官方解释<http://openjdk.java.net/jeps/122>

大概意思是移除永久代是为融合HotSpot与 JRockit而做出的努力，因为JRockit没有永久代，不需要配置永久代。



# 栈(Stack)

Java线程执行方法的内存模型，一个线程对应一个栈，每个方法在执行的同时都会创建一个栈帧（用于存储局部变量表，操作数栈，动态链接，方法出口等信息）不存在垃圾回收问题，只要线程一结束该栈就释放，生命周期和线程一 致

# 本地方法栈(Native Method Stack)

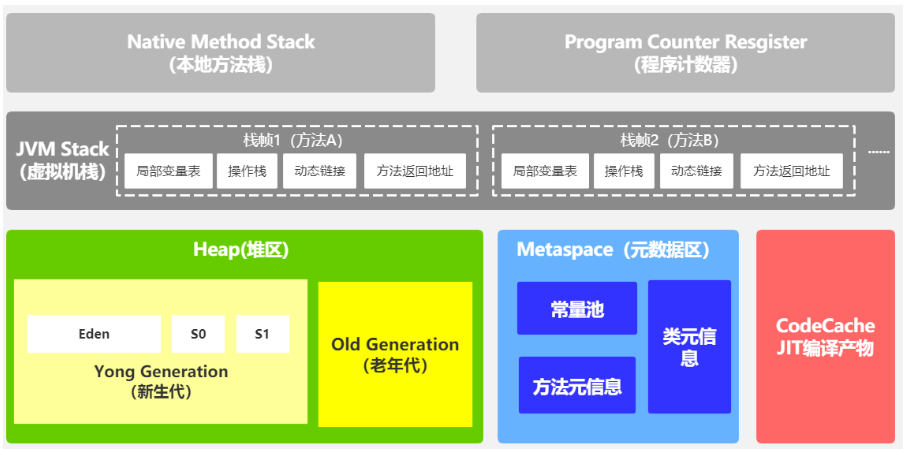
和栈作用很相似，区别不过是Java栈为JVM执行Java方法服务，而本地方法栈为JVM执行native方法服务。登记native 方法，在Execution Engine执行时加载本地方法库

# 程序计数器(Program Counter Register)

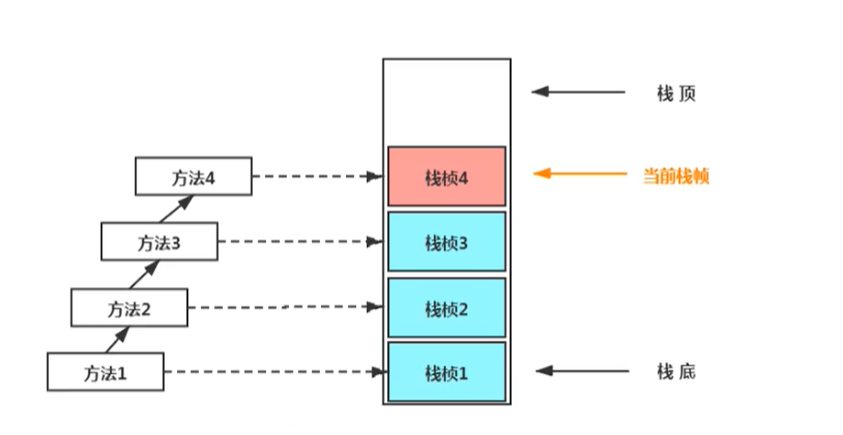
就是一个指针，指向方法区中的方法字节码（用来存储指向吓一跳指令的地址，也即将要执行的指令代码），由执行 引擎读取下一条指令，是一个非常小的内存空间，几乎可以忽略不计

# 参考示例

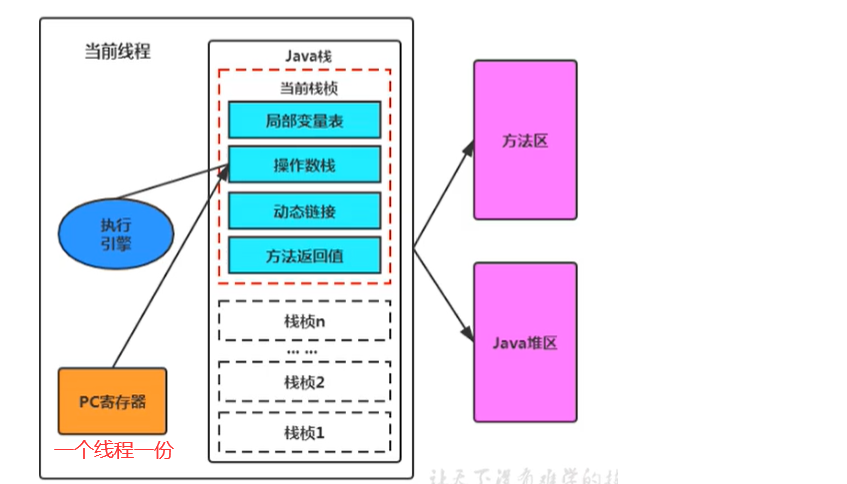
运行时数据区



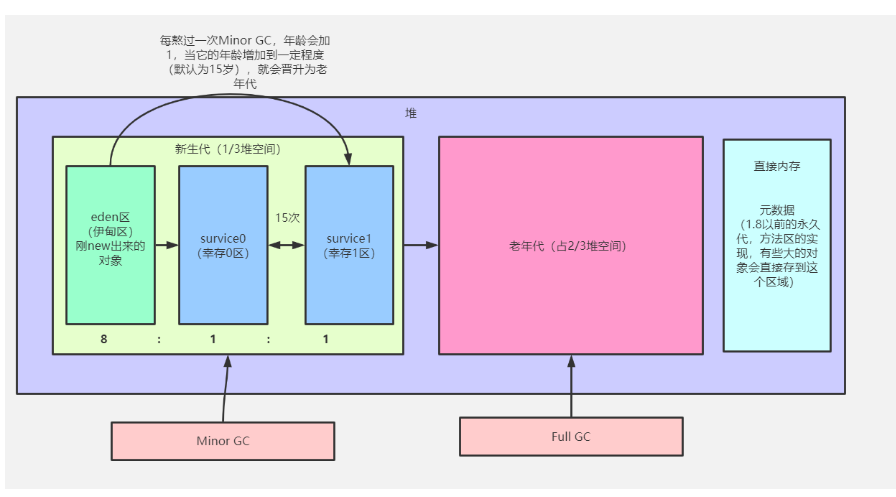
栈运行原理



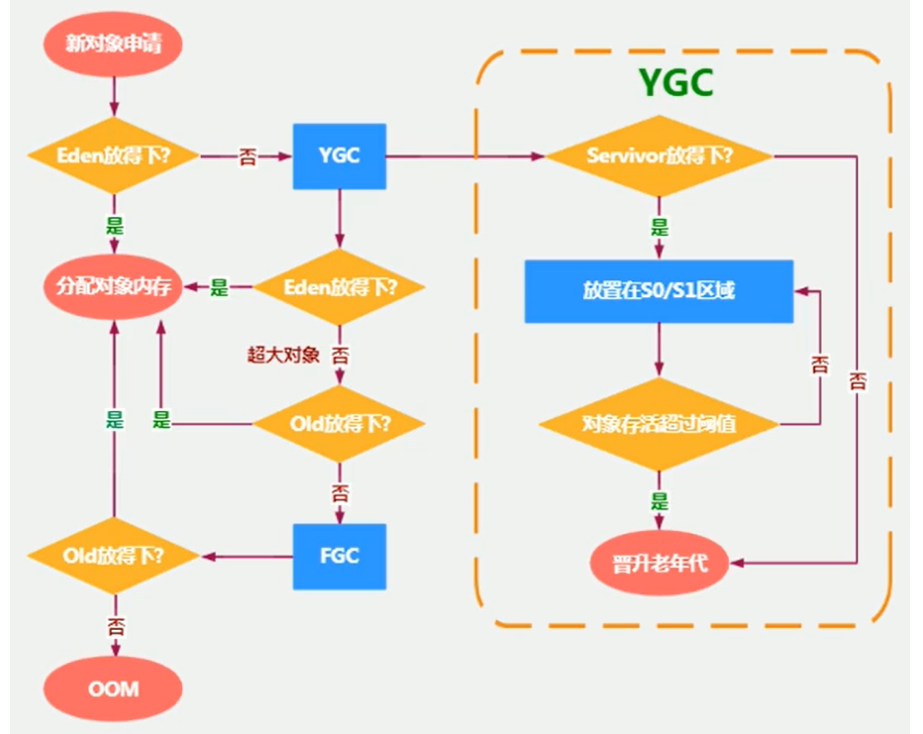
程序计数器



堆内存结构



对象分配规则



# TLAB(Thread Local Allocation Buffer)

为什么有TLAB？

堆区是线程共享区域，任何线程都可以访问到堆区中的共享数据

由于对象实例的创建在JVM中非常频繁，因此在并发环境下从堆区中划分内存空间是线程不安全的

为避免多个线程操作同一地址，需要使用加锁等机制，进而影响分配速度。

什么是TLAB?

从内存模型而不是垃圾收集的角度，对Eden区域继续进行划分，JVM为每个线程分配了一个私有缓存区域，它包含在Eden空间内。

多线程同时分配内存时，使用TLAB可以避免一系列的非线程安全问题，同时还能够提升内存分配的吞吐量，因此我们可以将这种内存分配方式称之为**快速分配策略**。

尽管不是所有的对象实例都能够在TLAB中成功分配内存，但JVM确实是将TLAB作为内存分配的首选。

在程序中，开发人员可以通过选项“-XX:UseTLAB”设置是否开启TLAB空间。

默认情况下，TLAB空间的内存非常小，仅占有整个Eden空间的1%，当然我们可以通过选项“-XX:TLABWasteTargetPercent"设置TLAB空间所占用Eden空间的百分比大小。

一旦对象在TLAB空间分配内存失败时，JVM就会尝试着通过使用**加锁机制**确保数据操作的原子性，从而直接在Eden空间中分配内存。

