

Java代码执行流程

## JVM与Java体系结构

### JVM生命周期

#### 虚拟机的启动

Java虚拟机的启动是通过引导类加载器 (bootstrap class loader) 创建一个初始类 (initial class) 来完成的，这个类是由虚拟机的具体实现指定的。

#### 虚拟机的执行

程序开始执行它运行，程序结束时它停止

#### 虚拟机的退出

程序正常执行退出

程序在执行过程中遇到了异常或错误而异常终止

由于操作系统出现错误而导致Java虚拟机进程终止

某线程调用Runtime类或system类的exit方法，或Runtime类的halt方法，并且Java安全管理器也允许这次exit或halt操作。

### Java的架构模型

基于栈的指令集架构，另一种指令集架构是基于寄存器的指令集架构

栈式架构设计简单，不需要硬件支持

由于跨平台性的设计，Java的指令都是根据栈来设计的。不同平台CPU架构不同，所以不能设计为基于寄存器的。优点是跨平台，指令集小，编译器容易实现，缺点是性能下降，实现同样的功能需要更多的指令。

字节码反编译      `javap -v xxx.class`

第一款商用Java虚拟机，JDK1.4时完全被淘汰。

Sun Classic VM

这款虚拟机内部只提供解释器。如果外挂JIT，解释器和编译器不能配合工作

Exact VM

目前Hotspot占有绝对的市场地位，称霸武林。

HotSpot VM

不管是现在仍在广泛使用的JDK6，还是使用比例较多的JDK8中，默认的虚拟机都是HotSpot

Sun/oracle JDK和openJDK的默认虚拟机

名称中的HotSpot指的就是它的热点代码探测技术。

从服务器、桌面到移动端、嵌入式都有应用。

不太关注程序启动速度，因此JRockit内部不包含解析器实现，全部代码都靠即时编译器编译后执行。

JRockit

JRockit JVM是世界上最快的JVM。

IBM J9

市场定位与HotSpot接近，服务器端、桌面应用、嵌入式等多用途VM广泛用于IBM的各种Java产品。

有影响力的三大商用虚拟机之一

KVM和CDC / CLDC Hotspot

智能控制器、传感器

老人手机、经济欠发达地区的功能手机

Azul VM

Azul VM是AzulSystems公司在HotSpot基础上进行大量改进，运行于Azul Systems公司的专有硬件Vega系统上的java虚拟机。

Liquid VM

Liquid VM不需要操作系统的支持，或者说它自己本身实现了一个专用操作系统的必要功能，如线程调度、文件系统、网络支持等。

随着JRockit虚拟机终止开发，Liquid VM项目也停止了。

Apache Harmony

它是IElf和Inte1联合开发的开源JVM

Micorsoft JVM

只能在window平台下运行。但确是当时Windows下性能最好的Java VM。

由AliJVM团队发布。阿里，国内使用Java最强大的公司，覆盖云计算、金融、物流、电商等众多领域，需要解决高并发、高可用、分布式的复合问题。有大量的开源产品。

Taobao JVM

基于openJDK开发了自己的定制版本AlibabaJDK，简称AJDK。是整个阿里Java体系的基石。

目前已经在线上，把oracle官方JvM版本全部替换了。

谷歌开发的，应用于Android系统

Dalvik VM

Dalvik VM只能称作虚拟机，而不能称作“Java虚拟机”，它没有遵循 Java虚拟机规范

不能直接执行Java的Class文件

基于寄存器架构，不是jvm的栈架构。

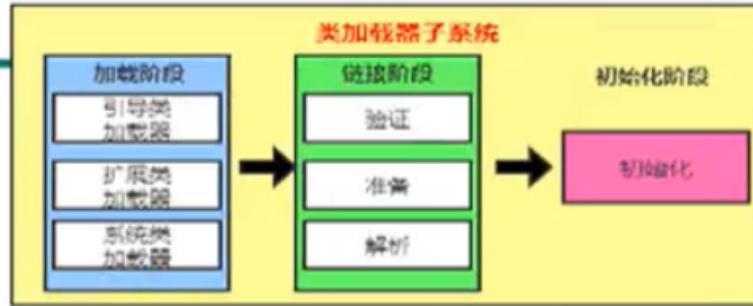
GraalVM在HotSpot VM基础上增强而成的跨语言全栈虚拟机，可以作为“任何语言”

的运行平台使用。语言包括：Java、Scala、Groovy、Kotlin；C、C++、Javascript、Ruby、Python、R等

Graal VM

支持不同语言中混用对方的接口和对象，支持这些语言使用已经编写好的本地库文件

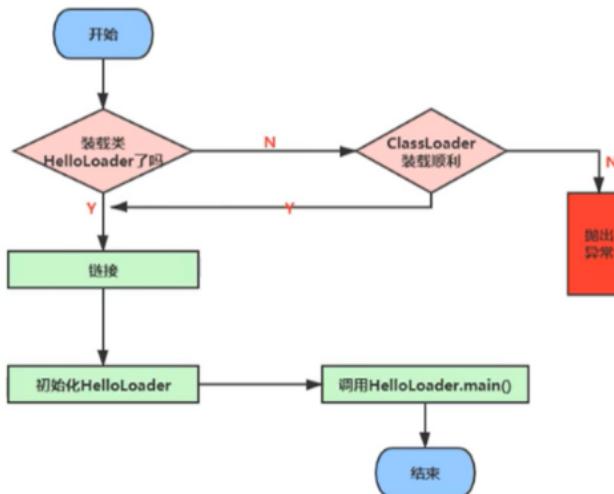
JVM的发展

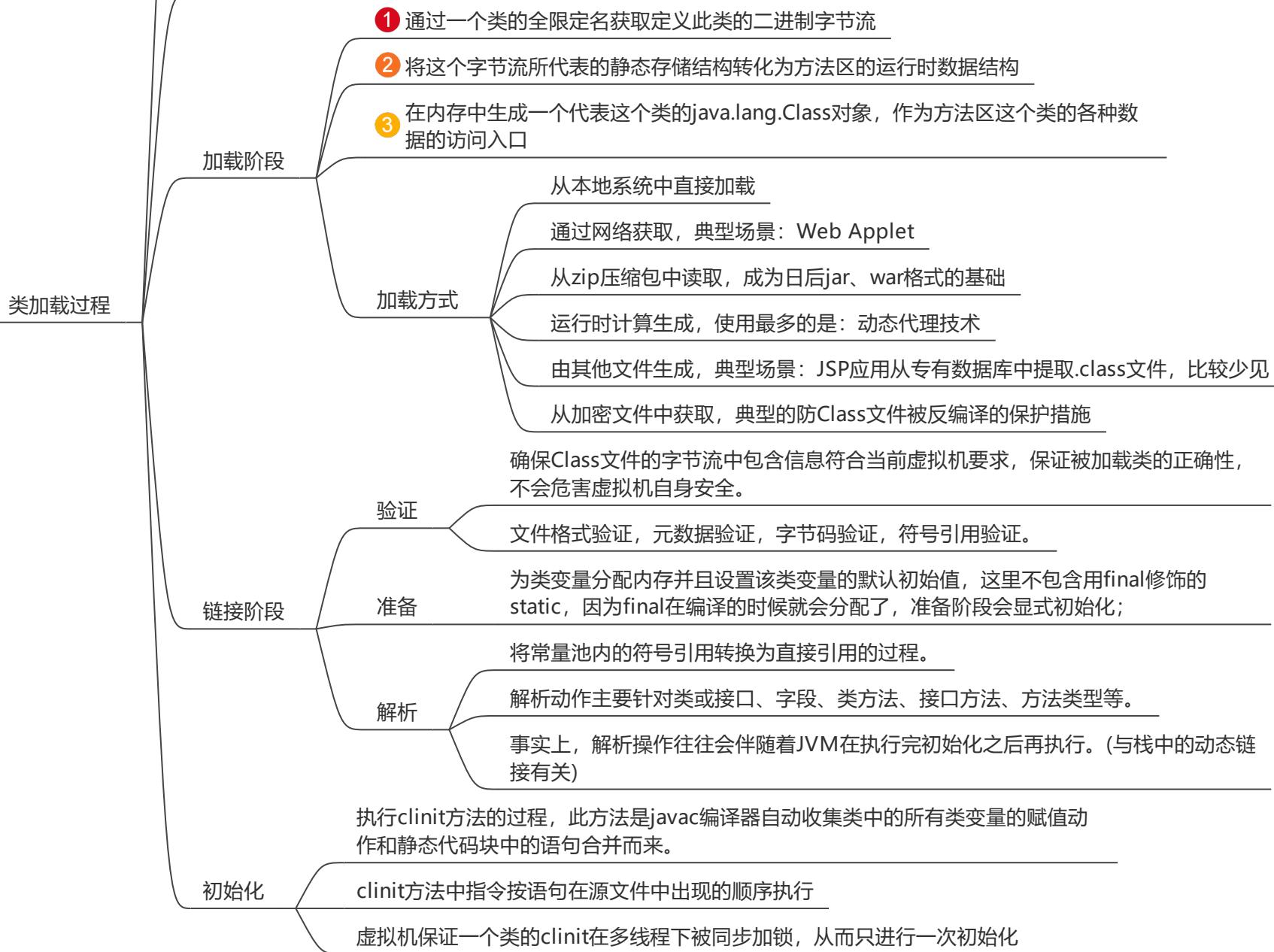


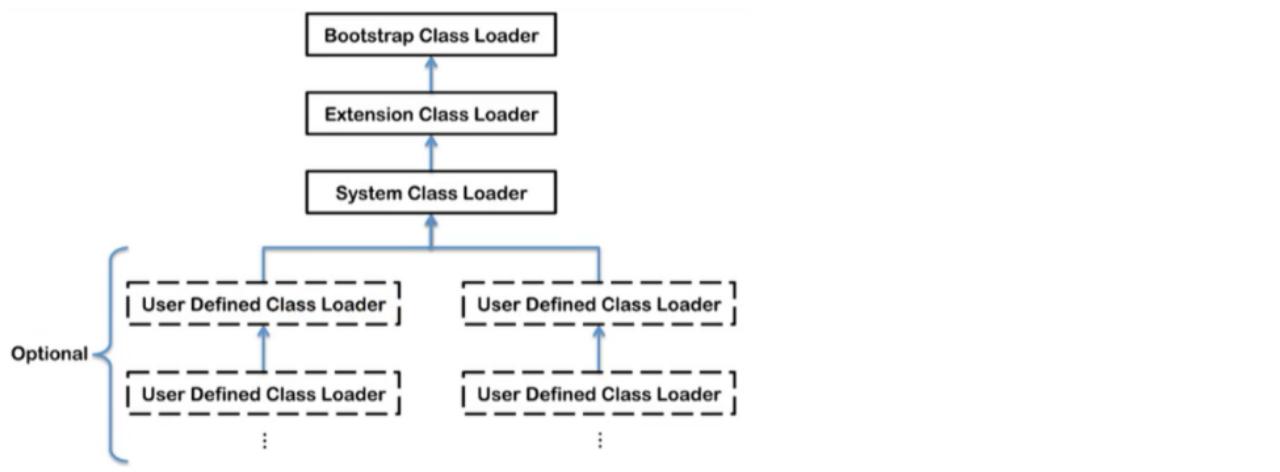
类加载器子系统负责从文件系统或者网络中加载Class文件，class文件在文件开头有特定的文件标识。

**作用**

加载的类信息存放于一块称为方法区的内存空间。除了类的信息外，方法区中还会存放运行时常量池信息，可能还包括字符串字面量和数字常量（这部分常量信息是Class文件中常量池部分的内存映射）







`ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();`

系统类加载器  
父加载器为扩展类加载器,派生于ClassLoader类  
它负责加载环境变量classpath或系统属性java.class.path指定路径下的类库  
该类加载是程序中默认的类加载器,一般来说, Java应用的类都是由它来完成加载

`ClassLoader extClassLoader = systemClassLoader.getParent();`

扩展类加载器  
父加载器为启动类加载器,派生于ClassLoader类  
从java.ext.dirs系统属性所指定的目录中加载类库,或从JDK的安装目录的jre/lib/ext子目录(扩展目录)下加载类库。如果用户创建的JAR放在此目录下,也会自动由扩展类加载器加载。

`ClassLoader bootstrapClassLoader = extClassLoader.getParent();`

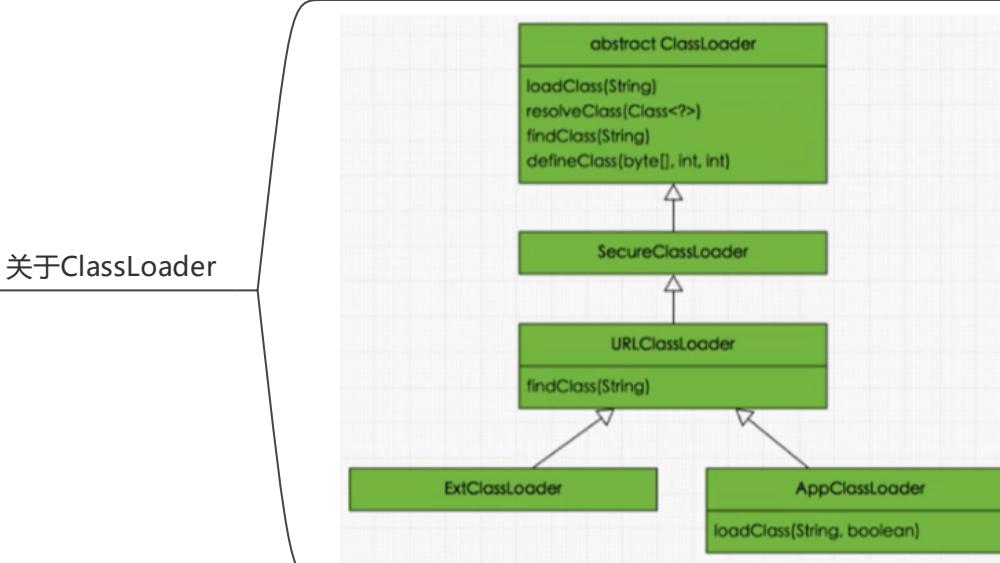
根加载器  
获取不到,无法直接通过代码获取  
使用C/C++实现,嵌套在JVM内部,用来加载Java核心库  
加载扩展类和应用程序类加载器,并指定为它们的父类加载器  
只加载包名为java/javax/sum等开头的类

`ClassLoader classLoader = ClassLoaderTest.class.getClassLoader();`

为什么要自定义类加载器?  
继承抽象类java.lang.ClassLoader  
实现步骤  
自定义的类加载逻辑写在findclass中  
如果没有太过复杂的需求,可以直接继承URLClassLoader,避免自己去编写findclass方法

String类加载器  
获取不到  
Java的核心类库都是使用根加载器进行加载的

除了根加载器,其他所有的类加载器都继承自ClassLoader



## 类加载子系统

### 类加载器分类

#### 自定义加载器

隔离加载类

修改类加载的方式

扩展加载源

防止源码泄露

继承抽象类java.lang.ClassLoader

自定义的类加载逻辑写在findclass中

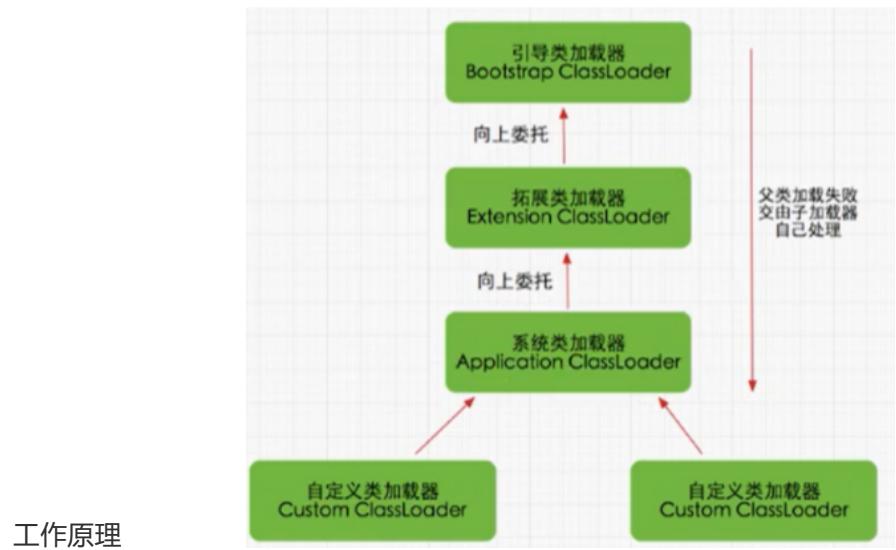
如果没有太过复杂的需求,可以直接继承URLClassLoader,避免自己去编写findclass方法

#### String类加载器

获取不到

Java的核心类库都是使用根加载器进行加载的

### 关于ClassLoader



### 双亲委派机制 (Parent Delegation Mechanism)

#### 实例举例 (Example)

加载jdbc.jar 用于实现数据库连接的时候，首先我们需要知道的是 jdbc.jar是基于 SPI接口进行实现的，所以在加载的时候，会进行双亲委派，最终从根加载器中加载 SPI核心类，然后在加载SPI接口类，接着在进行反向委派，通过线程上下文类加载器进行实现类 jdbc.jar的加载。

#### 沙箱安全机制 (Sandboxed Security Mechanism)

如果我们自定义String类，加载String类的时候率先使用根加载器加载，而引导类加载器加载的过程中没有main方法，此时报错信息会报错没有main方法，因为加载的不是自定义String，而是jdk的

#### 优势 (Advantages)

避免类的重复加载

沙箱安全机制保护程序安全

类的完整类名必须一致

两者的类加载器必须相同

即使两个类对象来源同一个Class文件，被同一个虚拟机加载，但只要它们的 ClassLoader实例对象不同，那么这两个类对象也是不相等的

#### 如何判断两个Class对象是否相等 (How to Determine if Two Class Objects are Equal)

#### 类的主动使用和被动使用 (Active and Passive Use of Classes)

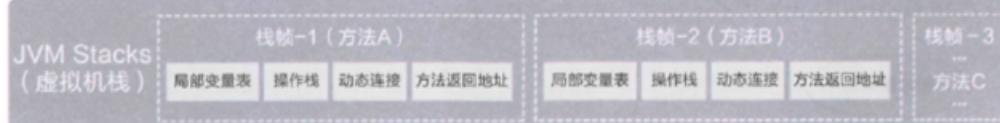
##### 主动使用 (Active Use)

- new
- 访问静态变量
- 调用静态方法
- 反射
- 初始化一个类的子类
- 启动类
- 动态语言支持

除了主动使用都是被动使用，不会导致类的初始化，注意到，如果是.class引用 Class对象，会进行类的加载和链接，不会进行初始化

## Native Method Stacks (本地方法栈)

## Program Counter Register (程序计数器)



JVM内存布局规定了Java在运行过程中内存申请、分配、管理的策略，保证了JVM的高效稳定运行。不同的JVM对于内存的划分方式和管理机制存在着部分差异。Java虚拟机定义了若干种程序运行期间会使用到的运行时数据区，其中有一些会随着虚拟机启动而创建，随着虚拟机退出而销毁。另外一些则是与线程一一对应的，这些与线程对应的数据区域会随着线程开始和结束而创建和销毁。

在Hotspot JVM里，每个线程都与操作系统的本地线程直接映射。当一个Java线程准备好执行以后，此时一个操作系统的本地线程也同时创建。Java线程执行终止后，本地线程也会回收。

JVM线程  
系统线程

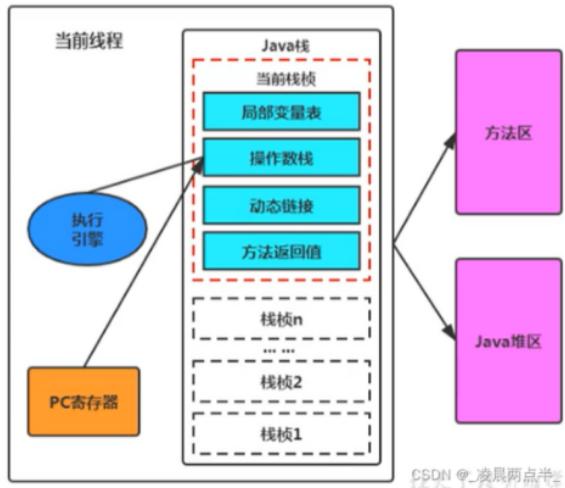
虚拟机线程，JVM到达安全点才会出现，这种线程的执行类型包括STW的垃圾收集，线程栈收集、线程挂起，以及偏向锁撤销

周期任务线程：时间周期事件的体现（比如中断），执行周期性操作的调度

GC线程，垃圾收集

编译线程：字节码编译成本地代码

信号调度线程，接受信号发送给JVM



PC寄存器用来存储指向下一条指令的地址，也即将要执行的指令代码。由执行引擎读取下一条指令。

每个线程都有它自己的程序计数器，是线程私有的，生命周期与线程的生命周期保持一致。

任何时间一个线程都只有一个方法在执行，也就是所谓的当前方法。程序计数器会存储当前线程正在执行的Java方法的JVM指令地址；或者，如果是在执行native方法，则是未指定值 (undefined)。

程序控制流的指示器，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。

唯一一个没有OOM的区域

## 程序计数器 (线程私有)

每个线程在创建时都会创建一个虚拟机栈，其内部保存一个个的栈帧（Stack Frame），对应着一次次的Java方法调用。

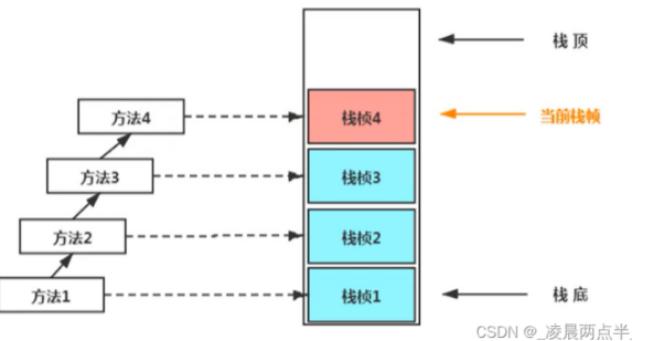
生命周期和线程一致，也就是线程结束了，该虚拟机栈也销毁了

保存方法的局部变量、部分结果，并参与方法的调用和返回。

如果采用固定大小的Java虚拟机栈，那每一个线程的Java虚拟机栈容量可以在线程创建的时候独立选定。如果线程请求分配的栈容量超过Java虚拟机栈允许的最大容量，Java虚拟机将会抛出一个StackOverflowError 异常。

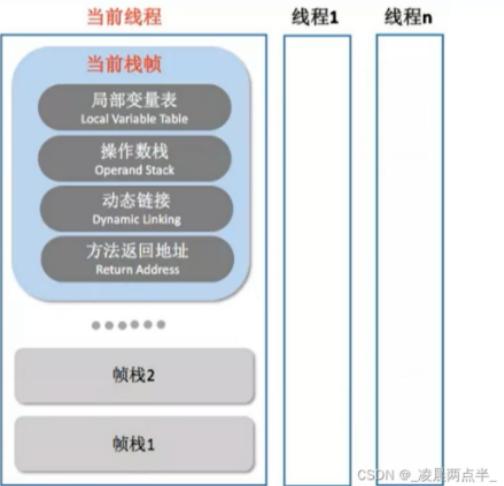
如果Java虚拟机栈可以动态扩展，并且在尝试扩展或者在创建新的线程时没有足够的内存去创建对应的虚拟机栈，那Java虚拟机将会抛出一个 outofMemoryError 异常。

-Xss设置线程最大栈空间



在一条活动线程上，一个时间点上只有一个活动的栈帧，即只有当前正在执行的方法的栈帧有效，这个栈帧被称为Current Frame，与当前栈帧对应的方法就是Current Method，定义这个方法的类是Current Class。方法调用入栈，方法返回出栈

虚拟机栈（线程私有）



主要用于存储方法参数和定义在方法体内的局部变量这些数据类型包括各类基本数据类型、对象引用（reference），以及returnAddress类型。

局部变量表所需的容量大小是在编译期确定下来的。在方法运行期间是不会改变局部变量表的大小的。

最基本的存储单元Slot。32位以内的类型只占用一个slot，64位占用两个slot

如果当前栈是构造方法或者实例方法创建的，那么该对象引用this将会存放在index为0的slot处，其余的参数按照参数表顺序继续排列。

栈帧

局部变量表

局部变量表中的slot可重用，局部变量超过其作用域，那么在其作用域之后声明的心局部变量就可能复用该槽位

局部变量表中的变量也是重要的垃圾回收根节点，只要被局部变量表中直接或间接引用的对象都不会被回收。

索引	类型	参数
0	int	int k
1	long	long m
3	float	float p
4	double	double q
6	reference	Object t

主要用于保存计算过程的中间结果，同时作为计算过程中变量临时的存储空间。

32bit的类型占用一个栈单位深度.64bit的类型占用两个栈单位深度

如果被调用的方法带有返回值的话，其返回值将会被压入当前栈帧的操作数栈中，并更新PC寄存器中下一条需要执行的字节码指令。

Java虚拟机的解释引擎是基于栈的执行引擎，其中的栈指的就是操作数栈。

操作数栈

内部组成

```
public void testAddOperation() {  
    byte i = 15;  
    int j = 8;  
    int k = i + j;  
}
```

```
public void testAddOperation();  
Code:  
0: bipush      15  
2: istore_1  
3: bipush      8  
5: istore_2  
6: iload_1  
7: iload_2  
8: iadd  
9: istore_3  
10: return
```

bipush入操作数栈

istore是操作数栈临时存储数据存入局部变量表

iload是局部变量表中数据放在操作数栈中iadd相加操作

然后存储局部变量表

栈顶缓存技术

操作数存储在内存，频繁进行入栈出栈影响执行速度，因此Tos技术就是将栈顶元素全部缓存在物理CPU的寄存器中，降低读写次数

```
public void localVar2() {  
    int a = 0;  
    System.out.println(a);  
}  
//此时的b就会复用a的槽位  
int b = 0;
```

每一个栈帧内部都包含一个指向运行时常量池中该栈帧所属方法的引用，包含这个引用的目的就是为了支持当前方法的代码能够实现动态链接（Dynamic Linking）

在Java源文件被编译到字节码文件中时，所有的变量和方法引用都作为符号引用（symbolic Reference）保存在class文件的常量池里。在进行类加载的时候加载链接初始化中的链接阶段，将符号引用转换为直接引用

### 动态链接

解析与分配（类加载中的解析阶段，事实上都是初始化之后进行）



### 运行时数据区

存放调用该方法的PC寄存器的值

### 方法返回地址

在方法退出后都返回到该方法被调用的位置。方法正常退出时，调用者的pc计数器的值作为返回地址，即调用该方法的指令的下一条指令的地址。而通过异常退出的，返回地址是要通过异常表来确定，栈帧中一般不会保存这部分信息。

在字节码指令中，返回指令包含ireturn（当返回值是boolean, byte, char, short和int类型时使用），lreturn（Long类型），freturn（Float类型），dreturn（Double类型），areturn。另外还有一个return指令声明为void的方法

在方法执行过程中遇到异常（Exception），并且这个异常没有在方法内进行处理，也就是只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出，简称异常完成出口。

方法执行过程中，抛出异常时的异常处理，存储在一个异常处理表，方便在发生异常的时候找到处理异常的代码

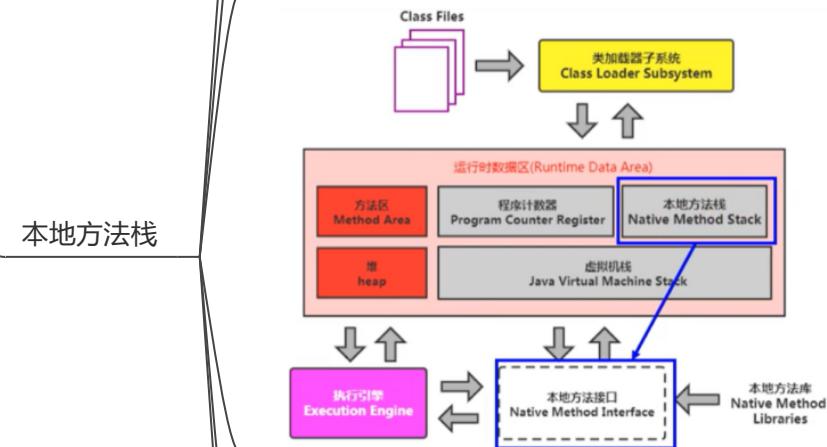
### 一些附加信息

# JVM

## 管理本地方法的调用

线程私有，和虚拟机栈一样允许被实现成固定或者是可动态扩展的内存大小，同样抛出SOF和OOM

具体做法是Native Method Stack中登记native方法，在Execution Engine执行时加载本地方法库



当某个线程调用一个本地方法时，它就进入了一个全新的并且不再受虚拟机限制的世界。它和虚拟机拥有同样的权限。可以通过本地方法接口来访问虚拟机内部的运行时数据区。可以直接使用本地处理器中的寄存器。直接从本地内存的堆中分配任意数量的内存。

注意：不是所有JVM都会实现此结构。在HotSpot的实现中，直接将本地方法栈和虚拟机栈合二为一

一个JVM实例只存在一个堆内存。Java堆区在JVM启动的时候即被创建，其空间大小也就确定了。是JVM管理的最大一块内存空间。《Java虚拟机规范》规定，堆可以处于物理上不连续的内存空间中，但在逻辑上它应该被视为连续的。

-Xms起始堆内存

-Xmx最大堆内存

通常会将-Xms和-Xmx两个参数配置相同的值，其目的是为了能够在Java垃圾回收机制清理完堆区后不需要重新分隔计算堆区的大小，从而提高性能。初始内存大小：物理电脑内存大小/64.最大内存大小：物理电脑内存大小/4

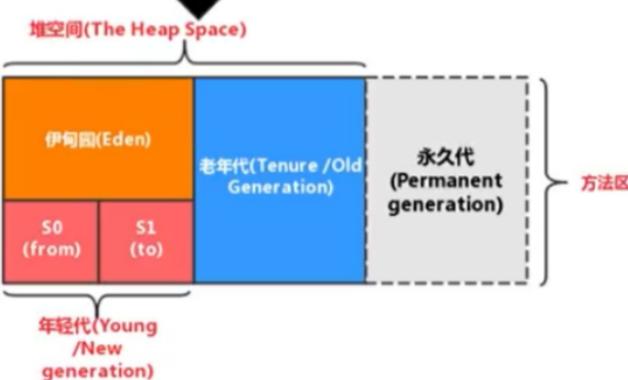
查看对内存的内存分配情况 jstat -gc 端口号

查看日志 -XX:+PrintGCDetails

一旦堆区中的内存大小超过 “-xmx” 所指定的最大内存时，OOM

Java VisualVM查看堆空间的内容，控制台使用jvisualvm命令可以打开

The heap is the run-time data area from which memory for all class instances and arrays is allocated



Java 7及之前堆内存逻辑上分为三部分：新生区+养老区+永久区

Java 8及之后堆内存逻辑上分为三部分：新生区+养老区+元空间

1.8从永久代替换成元空间



年轻代

Eden: from: to = 8 : 1 : 1      默认-xx:SurvivorRatio=8, 表示8:1:1

年轻代: 老年代 = 1 : 2      默认-XX:NewRatio=2, 表示1:2

如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，将被移动到survivor空间中，并将对象年龄设为1。对象在survivor区中每熬过一次MinorGC，年龄就增加1岁，当它的年龄增加到一定程度（默认为15岁，其实每个JVM、每个GC都有所不同）时，就会被晋升到老年代

内存细分

大对象直接分配到老年代（长度由-XX:PretenureSizeThreshold设置，只对Serial和ParNew有效）

长期存活的对象分配到老年代（默认15, -XX:MaxTenuringThreshold）

动态对象年龄判断：如果survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代，无须等到MaxTenuringThreshold 中要求的年龄。

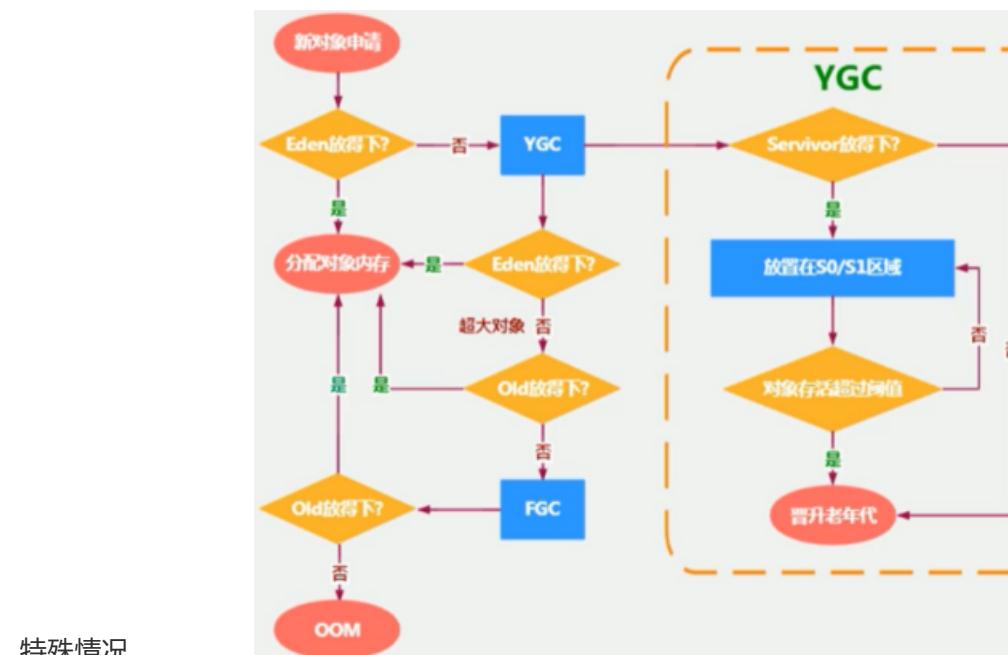
- 空间分配担保：-XX:HandlePromotionFailure

发生MinorGC之前，虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么这一次Minor GC就是安全的。如果不成立，虚拟机首先查看-XX:HandlePromotionFailure参数是否允许担保失败

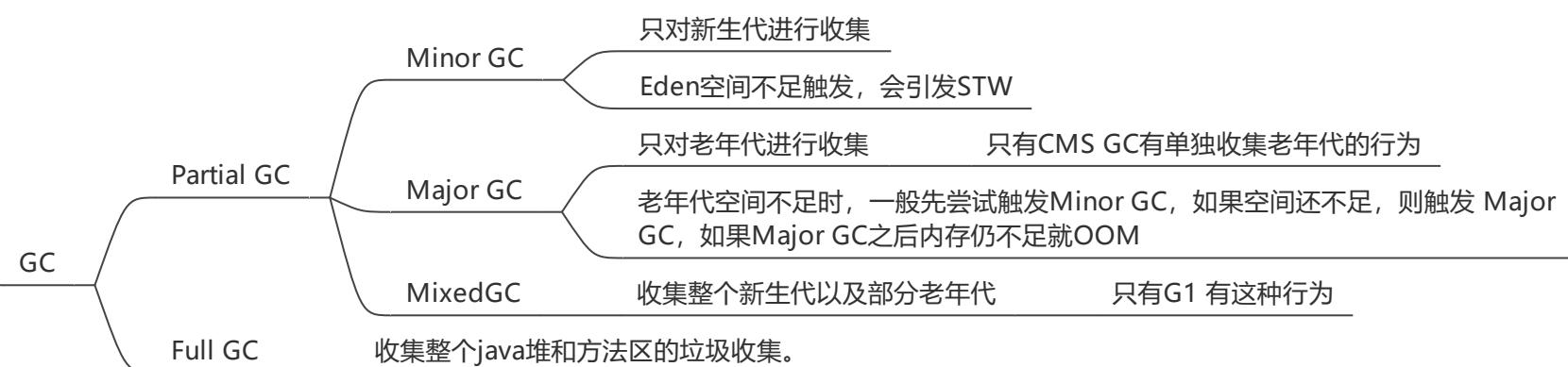
(Handle Promotion Failure) .如果允许，那会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，尝试进行一次Minor GC，尽管是有风险的。如果是小于，或者-XXHandlePromotion不允许冒险，那么就改成FullGC。通常情况下都会将此开关打开，避免FullGC太过频繁。

JDK Update24之后此参数失效，只要老年代连续空间大于新生代所有对象总大小，Minor GC，否则FullGC

对象分配过程



特殊情况

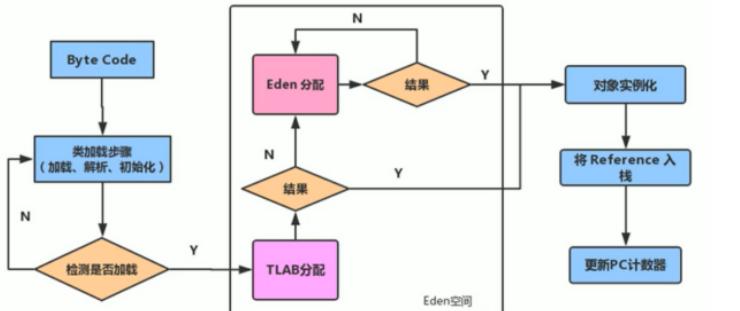


## Thread Local Allocation Buffer

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（因为在公共区域需要进行加锁），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。

-Xx:UseTLAB是否开启TLAB空间      -Xx:TLABWasteTargetPercent设置TLAB空间占用Eden空间百分比

TLAB



xms初始堆内存，默认物理内存1/64

xmx最大堆内存，默认物理内存的1/4

xmn新生代大小

NewRatio, 新生代老年代占比

Survivor Eden和s0.s1占比

MaxTenuringThreshold 垃圾最大年龄

printGCDetails输出详细GC处理日志

HandlePromotionFailure空间分配担保

堆空间的参数设置

一个对象并没有逃逸出方法的话，那么就可能被优化为栈上分配，这样就无需在堆上分配内存，无须进行垃圾回收了。

JIT编译器能够分析出一个新的对象的引用的适用范围从而决定是否要将这个对象分配到堆上

当一个对象在方法中被定义后，对象只在方法内部使用，则认为没有发生逃逸。

当一个对象在方法中被定义后，它被外部方法所引用，则认为发生逃逸。

逃逸分析

在jdk1.7之后，HotSpot中默认开启了逃逸分析

启发      开发中能使用局部变量的，就不要使用在方法外定义

栈上分配

将堆分配转化为栈分配。如果一个对象在子程序中被分配，要使指向该对象的指针永远不会发生逃逸，对象可能是栈上分配的候选，而不是堆上分配

同步省略

JIT编译器可以借助逃逸分析来判断同步块所使用的锁对象是否只能被一个线程访问而没有被发布到其他线程。如果没有，那么JIT编译器在编译这个同步块的时候就会取消对这部分代码的同步。

堆是分配对象的唯一选择吗？

标量（scalar）是指一个无法再分解成更小的数据的数据。Java中的原始数据类型就是标量。

分离对象和标量替换

经过逃逸分析发现一个对象不会逃逸，就可以将这个对象拆解为成员变量来代替

-server参数，启动server模式，server模式下才能启用逃逸分析

代码优化之标量替换

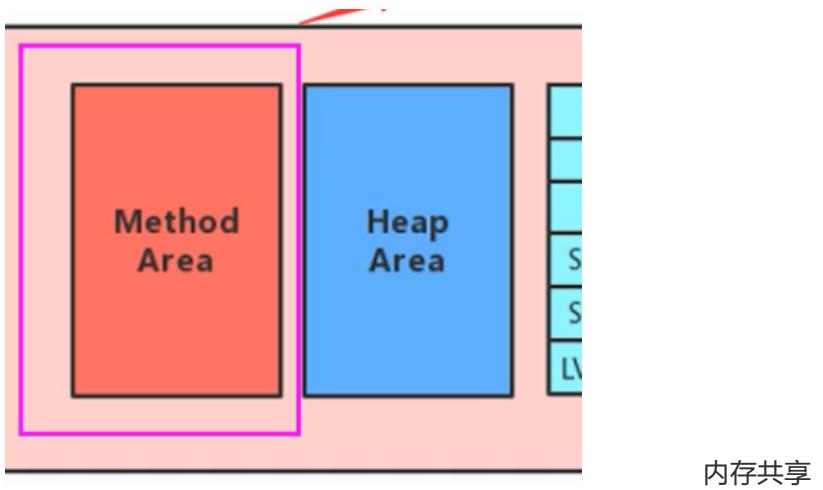
-XX:+DoEscapeAnalysis启用逃逸分析

-xx:+EliminateAllocations开启标量替换（默认打开）

逃逸分析的不足

根本原因是无法保证逃逸分析的性能消耗一定高于它的消耗，逃逸分析自身也需要进行复杂分析，相对耗时，Oracle HotSpot上并没有在栈上分配那些不会逃逸的对象

堆



#### 对象的访问定位

方法区主要存放的是 Class，而堆中主要存放的是 实例化的对象。方法区大小跟对空间一样，可以选择固定大小或者可扩展

或者java.lang.OutOfMemoryError:Metaspace

方法区是一种统称，jdk及以前习惯把方法区的实现称为永久代（通过-xx:PermSize来设置永久代初始分配空间。-XX:MaxPermSize来设定永久代最大可分配空间。）当加载类信息超过该值就会报OutOfMemoryError:PermGen space

类型信息、\*\*字符串常量池、静态变量\*\*等都放在永久代

JDK7,将字符串常量池、静态变量等移出放入堆

#### HotSpot中方法区的演进

jdk8开始元空间取代永久代，元空间存放在堆外内存中，元空间不在虚拟机设置的内存中，而是使用本地内存。元数据区大小可以使用参数 -XX:MetaspaceSize 和 -XX:MaxMetaspaceSize指定。如果不指定大小，默认情况下，会耗尽所有可用的系统内存。元数据区溢出，报OutOfMemoryError:Metaspace

触及初始高水位线，FullGC，根据GC释放了多少元空间重置这个水位线，为了避免频繁FullGC，应该设置一个较高的值

完全废弃永久代概念，和JRocket J9一样使用元空间。将类型信息全部移动到元空间中。字符串常量池、静态变量仍在堆中。

#### 方法区

##### 内存泄露和内存溢出

内存泄露是大量引用指向某些对象，但是这些对象以后不会使用了

内存溢出就是单纯的内存不够了

##### 如何解决OOM

通过dump出来的存储快照进行分析，分清楚发生Memory Leak还是Memory Overflow

内存泄露，进一步通过工具查看GC Roots引用链，定位泄露代码位置

内存溢出，检查堆参数xms和xmx，与物理机器内存对比看是否可以进行调整，从代码上检查是否存在某些对象生命周期过长，持有状态时间过长的情况

使用Jprofiler,将生成dump文件打开，点击Biggest Objects就能看到超大对象了，通过线程还能够定位到哪里出现OOM

不是虚拟机运行时数据区的一部分，也不是《Java虚拟机规范》中定义的内存区域。直接内存是在Java堆外的、直接向系统申请的内存区间。

BIO使用非直接缓存区，需要从用户态切换成内核态，需要存储两次缓存，Java的NIO库允许Java程序使用直接内存(allocateDirect())，用于数据缓冲区

#### 直接内存 (Direct Memory)



直接内存大小可以通过MaxDirectMemorySize设置  
如果不指定，默认与堆的最大值-xmx参数值一致

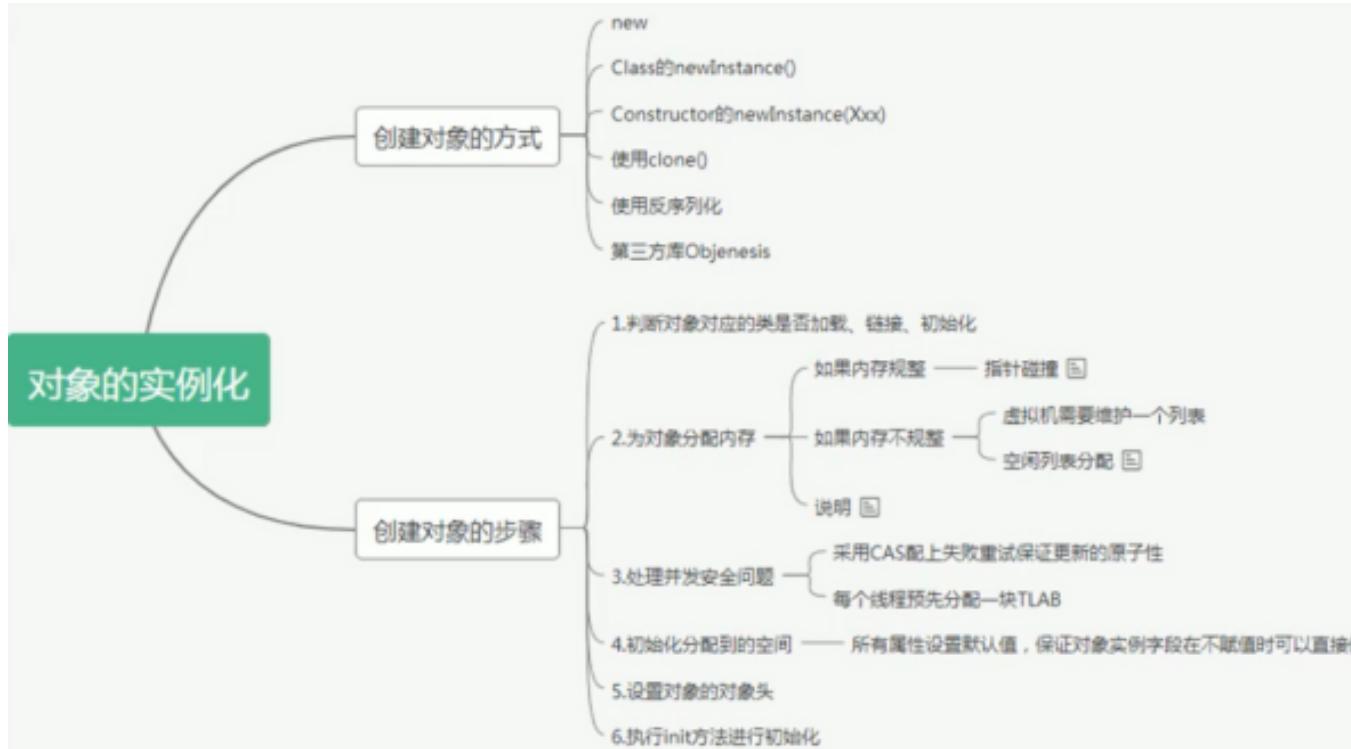
A native method is a Java method whose implementation is provided by non-java code

## 本地方法接口

### 为什么使用Native Method

然而有些层次的任务用Java实现起来不容易  
有时Java应用需要与Java外面的环境交互  
JVM依赖底层操作系统，通过本地方法实现与操作系统交互

目前本地方法接口使用的越来越少了，除非是硬件相关的应用，比如Java驱动打印机或者管理生产设备



① 到常量池中寻找引用，如果未被类加载，使用双亲委派机制查找对应的.class文件，没有找到抛出ClassNotFoundException异常，找到进行类加载生成Class对象

② 计算对象占用空间大小，划分内存，如果对象中有引用变量，划分引用变量空间即可4个字节

### 创建对象步骤

③ 初始化分配到的内存

属性默认初始化，显示初始化，代码块初始化，构造器初始化

④ 设置对象头

对象类的元数据信息，对象的HashCode，锁信息、分代信息

⑤ 执行init方法进行初始化

是否规整由垃圾收集器是否有Compact功能决定

用过内存放一边，空闲内存放一边，中间一个指针作为分界点，Serial ParNew基于这种压缩算法，虚拟机采用这种分配方式，一般使用有Compact过程的收集器使用指针碰撞。

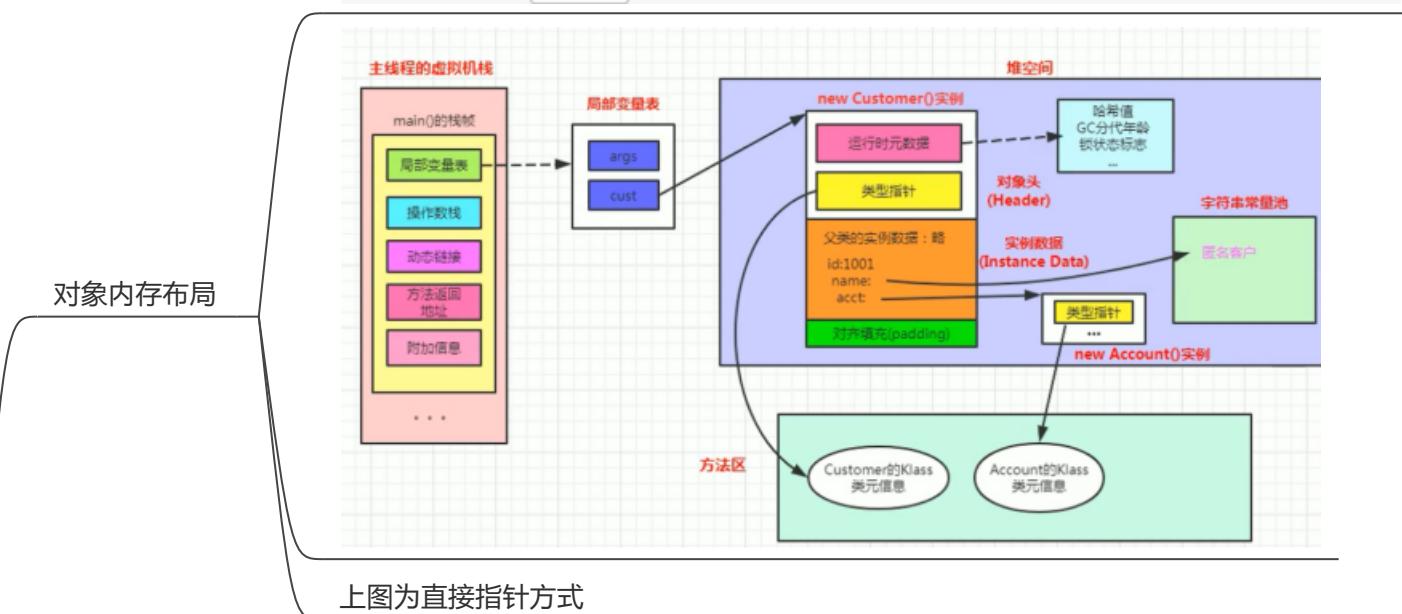
如果内存规整，指针碰撞

CMS使用空闲列表

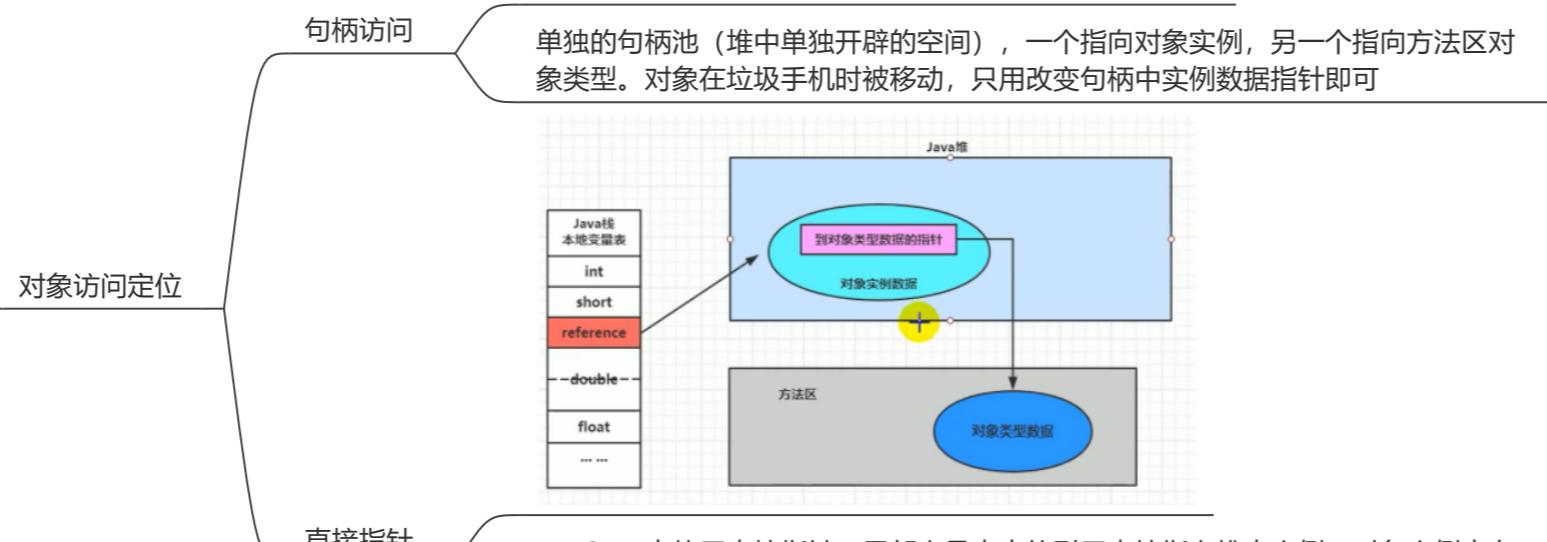
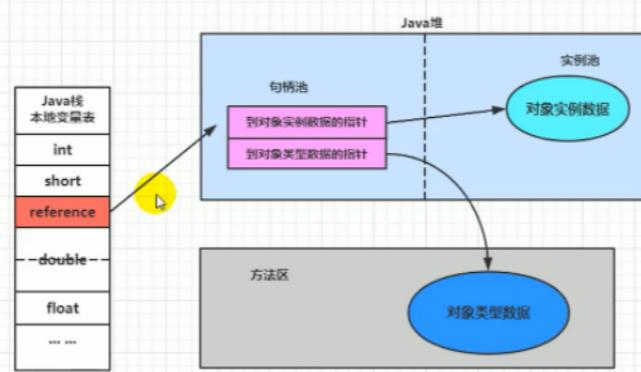
不规整，维护一个空闲列表进行分配

CAS分配保证更新原子性  
每个线程预先分配TLAB，通过设置 -XX:+UseTLAB参数来设置

处理并发问题



# 对象内存布局与访问定位



直接指针：HotSpot中使用直接指针，局部变量表中的引用直接指向堆中实例，对象实例中有类型指针指向方法区的类型数据

基本特性  
final的，不可继承  
实现了Serializable接口和Comparable接口

jdk8及以前内部为字符数组实现，JDK9改为byte数组

为什么JDK9改变结构？

每个字符使用两个字节，大多数字符串对象是拉丁字符，这些字符只需要一个字节  
粗糙农户，使用byte数组存储一个字节或两个字节的字符

永久代的默认比较小

永久代垃圾回收频率低

通过字面量的方式（区别于new）给一个字符串赋值，此时的字符串值声明在字符串常量池中。常量池中不存储相同内容字符串

jdk6之前，存放在永久代

jdk7放在堆中

jdk8也在堆中

内存分配

通过new作为对象分配在堆区

常量+常量 = 常量池

字符串拼接

只要其中有一个是变量，存放在堆中，变量拼接的原理是StringBuilder

JDK5之后使用StringBuilder，之前使用StringBuffer，线程安全但效率低，已经不推荐使用。StringBuilder的空参构造默认大小16，后面需要扩容，可以开始指定大小

特殊的如果是使用final+字面量的字符串引用，不会构建对象，而是放入常量池

String

字符串常量池一个

堆一个

new String("ab")几个对象？

StringBuilder

new String("a")

常量池a

new String("b")

常量池b

调用toString new String("ab"), 不会放到常量池

面试题

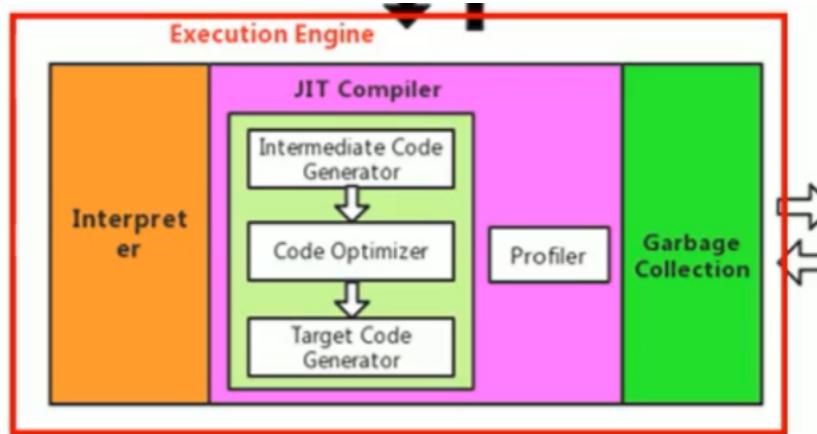
new String( "a" ) + new String( "b" ) 会创建几个对象？

jdk6是将对象复制一份放入串池

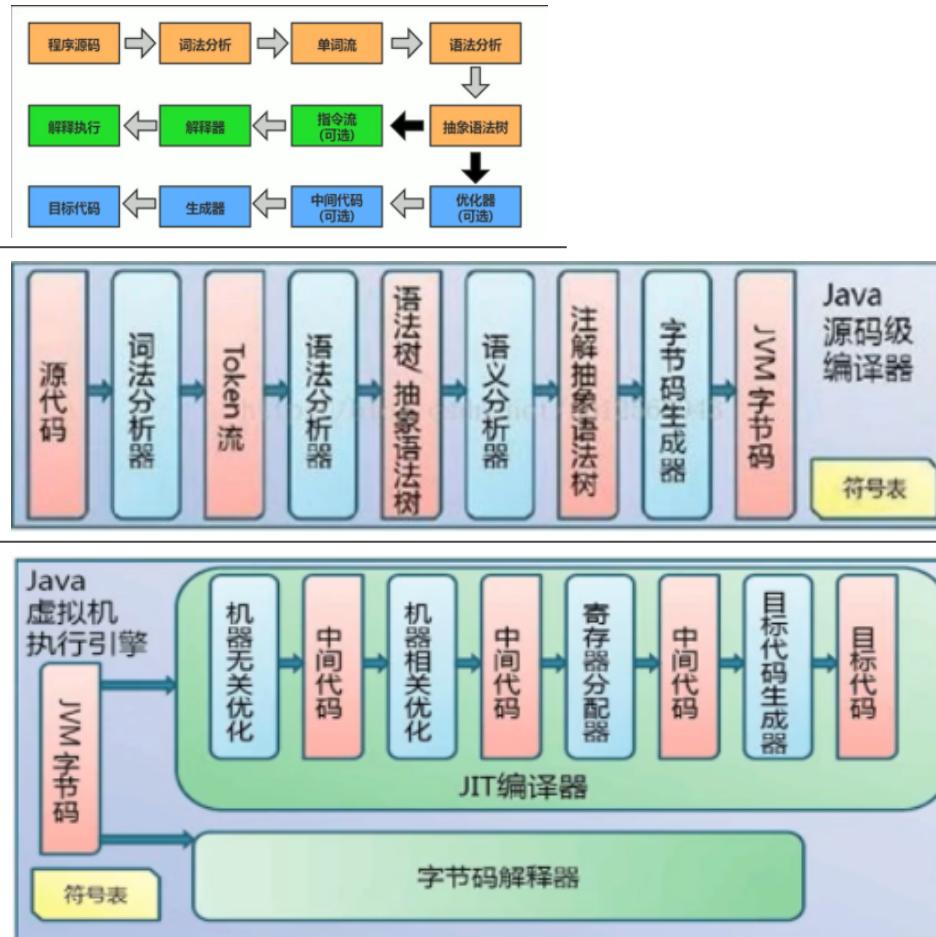
jdk7是将对象的引用地址复制到串池

G1中的字符串去重

将对中的后续String去重对象放到一个队列中，使用一个hashtable记录所有不重复的char数组，去重时，查这个hashtable，如果有相同的，调整引用，释放原引用，被垃圾回收，查找失败，char数组插入到hashtable



要想让一个Java程序运行起来，Execution Engine的任务就是将字节码指令解释/编译为对应平台上的本地机器指令才可以。



Java代码编译和执行过程

编译, 词法语法分析, 转换为汇编代码

C/C++ 编译过程 汇编, 进行其他目标代码, 库文件的链接, 汇编语言转机器指令

逐行解释, 将每条字节码文件中的内容翻译为对应平台的本地机器指令执行, 当一条字节码指令被解释执行完成后, 接着再根据PC寄存器中记录的下一条需要被执行的字节码指令解释操作

解释器

解释器分类

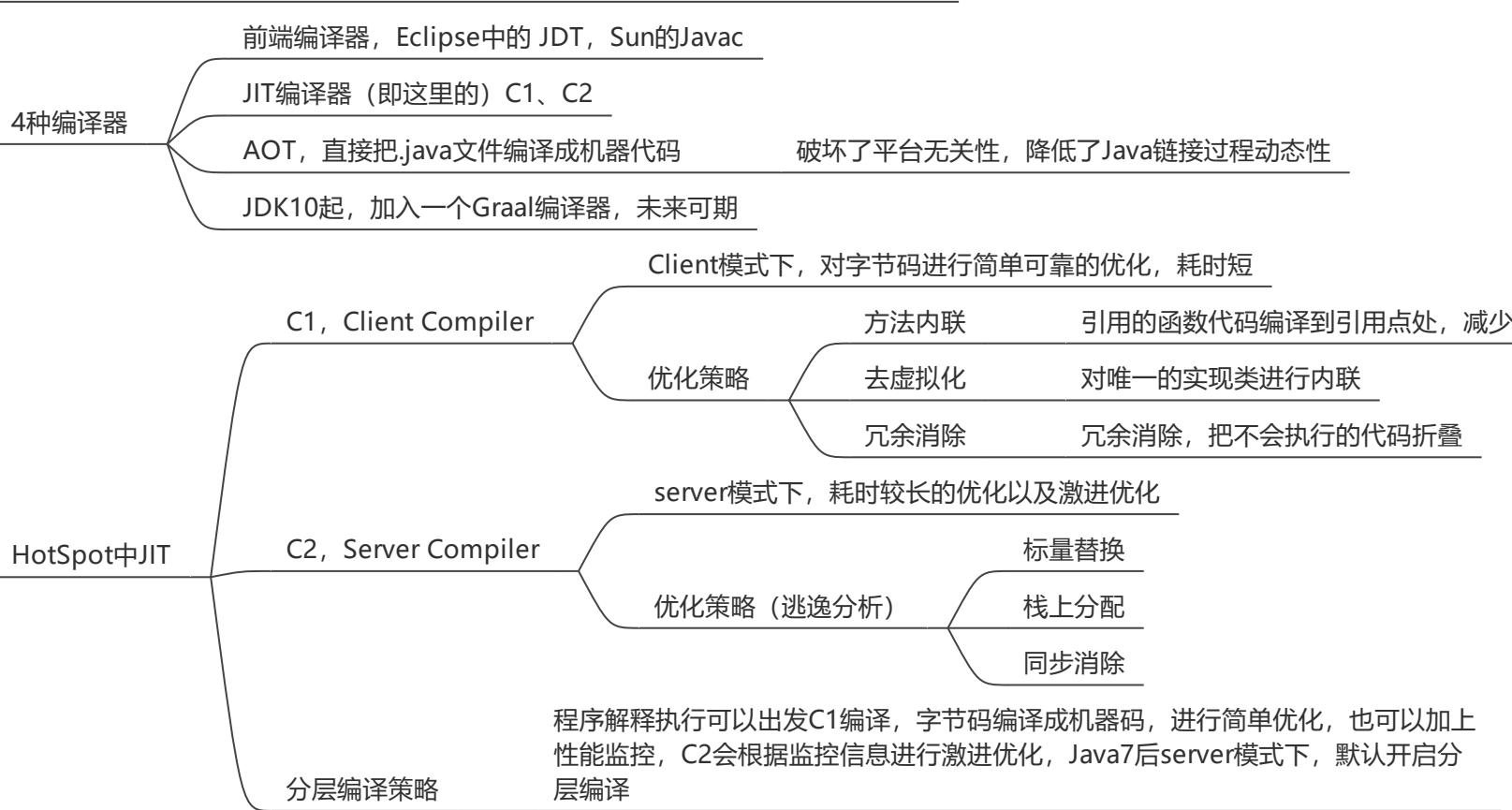
古老的字节码解释器, 通过纯软件模拟字节码的执行

模板解释器, 将每一条字节码和一个模板函数关联, 模板函数直接产生这条字节码执行时的机器码, 很大程度提高了解释器的性能

现状

基于解释器执行已经沦落为低效的代名词, JIT要比解释器高效很多

将方法编译成机器码之后再执行，HotSpot是解释器和JIT并存的架构，在Java虚拟机运行时，两者能够取长补短



程序启动后, 解释器可以马上发挥作用, 省去编译的时间, 立即执。随着时间的推移, 编译器发挥作用, 根据热点探测功能, 有价值的字节码代码编译成本地代码, 获得更高的执行效率, 同时, 解释器在编译器激进优化不成立的时候, 作为“逃生门”

两者如何协作?

HotSpot设置程序执行方法

-Xint: 完全采用解释器模式执行程序;

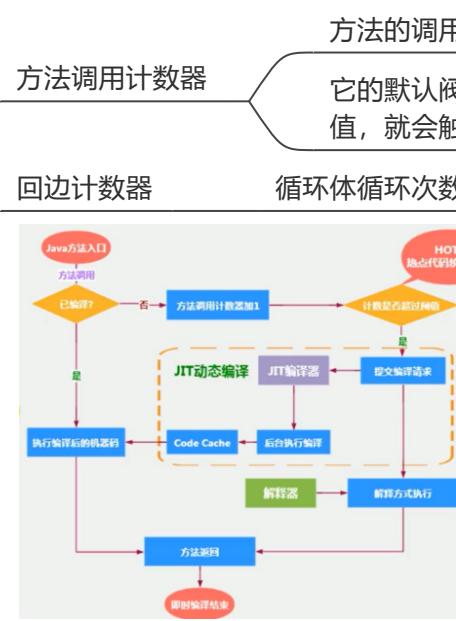
-Xcomp: 完全采用即时编译器模式执行程序。如果即时编译出现问题, 解释器会介入执行

-Xmixed: 采用解释器+即时编译器的混合模式共同执行程序。

一个被多次调用的方法, 或者是一个方法体内部循环次数较多的循环体都可以称之为“热点代码”, 因此都可以通过JIT编译器编译为本地机器指令。由于这种编译方式发生在方法的执行过程中, 因此称之为栈上替换

热点探测技术

目前HotSpot VM所采用的热点探测方式是基于计数器的热点探测。



方法调用计数器+循环计数器如果大于方法调用计数器的阈值就进行编译

方法调用计数器统计的并不是方法被调用的绝对次数, 而是一个相对的执行频率, 当超过一定的时间限度, 那这个方法的调用计数器就会被减少一半。这段时间被称为此方法的半衰周期。-XX:-UseCounterDecay关闭热度衰减。 -XX:CounterHalfLifeTime设置半衰周期

XX:CounterHalfLifeTime设置半衰周期

热点衰减

## 执行引擎

### 什么是垃圾?

垃圾是指在运行程序中没有任何指针指向的对象，这个对象就是需要被回收的垃圾。

### System.gc()

在默认情况下，通过system.gc()或者Runtime.getRuntime().gc()的调用，会显式触发FullGC，同时对老年代和新生代进行回收，尝试释放被丢弃对象占用的内存。system.gc()调用附带一个免责声明，无法保证对垃圾收集器的调用。(不能确保立即生效)。不一定会触发销毁的方法，调用System.runFinalization()会强制调用失去引用对象的finalize()

### STW

stop the world,指的是GC发生过程中，整个应用程序都会被短时间暂停，可达性分析算法中枚举根节点会导致停顿，因为保证一致性快照中进行。System.gc导致STW，所以不要使用

### 相关概念

#### 垃圾回收并行并发

多条垃圾收集线程并行工作，用户线程等待  
并行  
ParNew、Parallel Scavenge、Parallel Old;

单条线程执行  
串行  
Serial/Serial Old

垃圾收集线程和用户线程之间  
并发式垃圾回收器  
用户线程和垃圾收集线程并发交替执行

独占式垃圾回收器  
一旦运行，停止应用程序中的所有用户线程，直到垃圾回收过程完全结束

#### 安全点与安全区域

程序执行时并非在所有地方都能停顿下来开始GC，只有在特定的位置才能停顿下来开始GC，这些位置称为“安全点 (Safepoint) ”

程序在阻塞状态时，线程无法响应中断请求，走到安全点处，也不会被JVM唤醒，需要通过Safe Region来解决

#### 抢先式中断

目前没有采用，中断所有线程，如果有线程不在安全点，就恢复线程，让线程跑到安全点

#### 主动式中断

设置中断位置，各个线程运行到Safe Point的时候主动轮询这个标志，如果中断标志位为真，自己进行中断(轮询)

当线程运行到Safe Region的代码，标识已经进入Safe Region，这段时间内发生GC，忽略标识为Safe Region状态的线程

当线程即将离开Safe Region，检查JVM是否已经完成GC，如果完成了，继续运行，否则线程必须等待直到收到可以安全离开Safe Region的信号为止

### 面试：强引用、软引用、弱引用、虚引用区别？具体使用场景？

### 再谈引用

#### 强引用

object obj=new Object ()”这种引用关系。无论任何情况下，只要强引用关系还存在，垃圾收集器就永远不会回收掉被引用的对象。

#### 软引用 (softReference)

系统将要发生内存溢出以前，把这些对象进行GC

描述一些还有用，但是非必须的对象，当内存不足时，将这些对象回收，通常实现内存敏感的缓存，如果有闲置内存就暂时保存缓存，不足就删除缓存

#### 弱引用 (weakReference)

只能生存到下一次垃圾收集之前

更容易、更快被回收。发现即回收

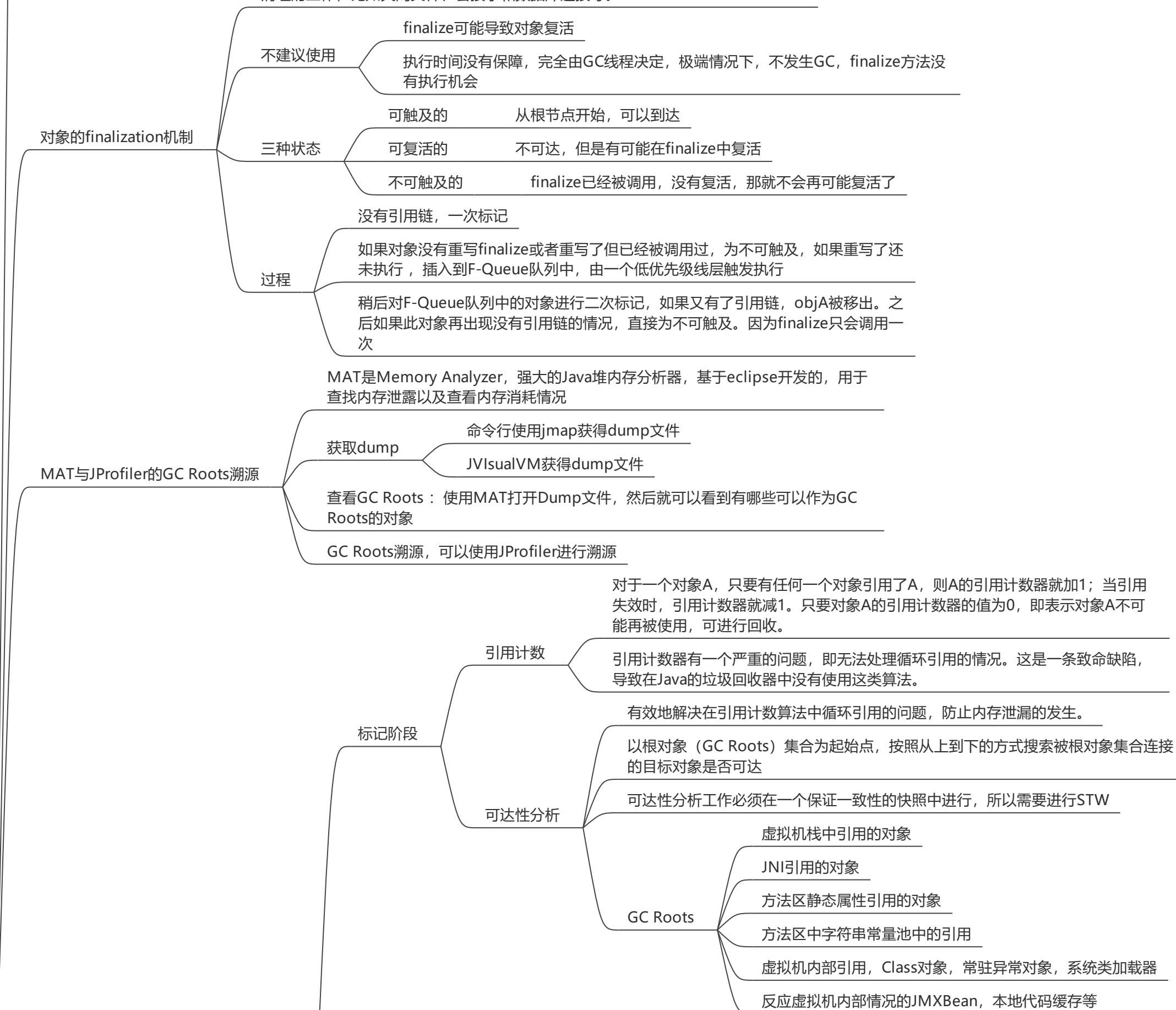
面试：你开发中使用过WeakHashMap吗？

用来存储图片信息，可以在内存不足的时候及时回收避免了OOM

#### 虚引用 (PhantomReference)

为一个对象设置虚引用关联的唯一目的在于跟踪垃圾回收过程。虚引用在创建时必须提供一个引用队列作为参数。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象后，将这个虚引用加入引用队列，以通知应用程序对象的回收情况。

允许开发人员提供对象被销毁之前的自定义处理逻辑。垃圾回收一个对象之前，先调用这个对象的finalize方法，允许在子类中重写该方法，用于进行一些资源释放和清理的工作，比如关闭文件、套接字和数据库连接等。



## 垃圾回收器

### 垃圾回收算法(垃圾收集器的理论基础)

#### 清除阶段

##### 标记清除算法 (Mark Sweep)

标记

从引用根节点开始遍历，标记所有被引用的对象

清除

对堆内存从头到尾进行线性遍历，如果发现某个对象在其Header中没有标记为可达对象，将其回收

并不是真的置空，而是把需要清除对象地址保存在空闲列表中（因为清除后内存不规整了），下次有新对象时可以进行覆盖。

STW，效率不算高，内碎片问题

##### 复制算法 (copying)

两块内存空间，回收后或者的对象复制到另一个内存块中，然后清除此内存，新生代的s0和s1即是此种方式

实现简单，运行高校，没有碎片问题，唯一缺点是两倍内存空间

##### 标记压缩/整理算法 (Mark-Compact)

从根节点开始标记所有存活对象

整理压缩清除边界空间

效率低于复制算法，但不用很高的内存代价，区域不分散，不用维护空闲列表开销

#### 分代收集算法

#### 增量收集算法

不一次性处理所有垃圾，每次只收集一小片区域的内存空间然后切换应用程序线程，反复

线程上下文转换消耗，吞吐量下降

#### 分区算法

将一个大的内存区域分割成多个小块，每次合理回收若干个小区间，而不是整个堆空间，减少每次STW的停顿

#### 并行并发分类

##### 碎片处理方式分类

压缩式

非压缩式

##### 内存区间分类

年轻代

老年代

#### 前三个不可能三角

内存占用忍，主要是暂停时间和吞吐量的矛盾，尽量在最大吞吐量优先的情况下，降低停顿时间

运行用户代码的时间占总运行时间的比例

单位时间STW最短

内存占用

暂停时间 每次STW时间最短

### GC分类和性能指标

#### 性能指标

垃圾收集开销

收集频率

快速

串行回收器

Serial、Serial old

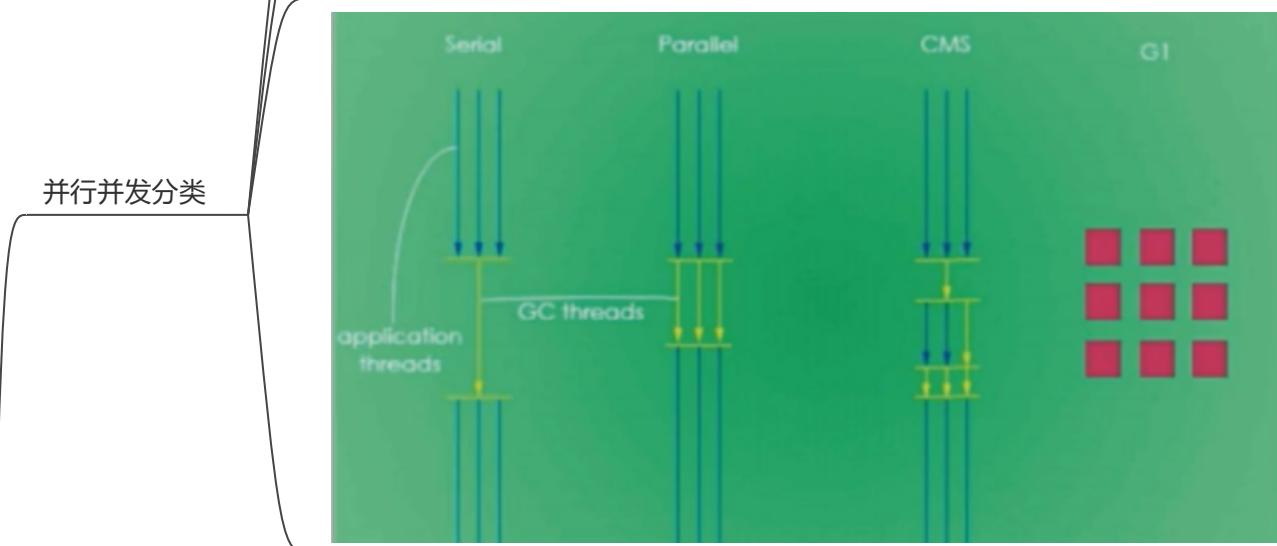
并行回收器

ParNew、Parallel Scavenge、Parallel old

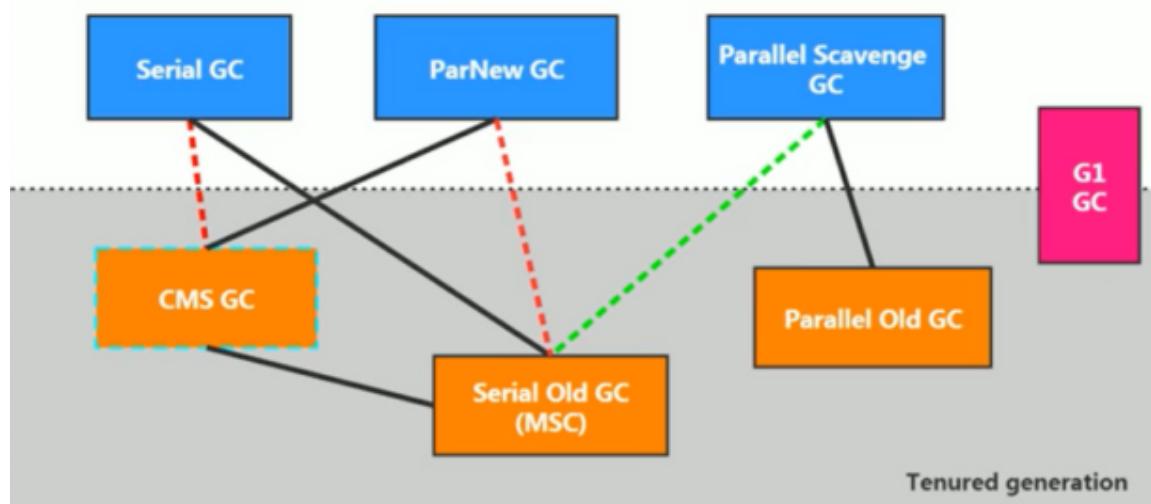
并发回收器

CMS、G1

并行并发分类



Young generation



黑色连线代表可以搭配使用，其中Serial Old作为CMS失败的后备预案

红色虚线表示兼容性测试版本，JDK9完全取消

绿色虚线，JDK14弃用

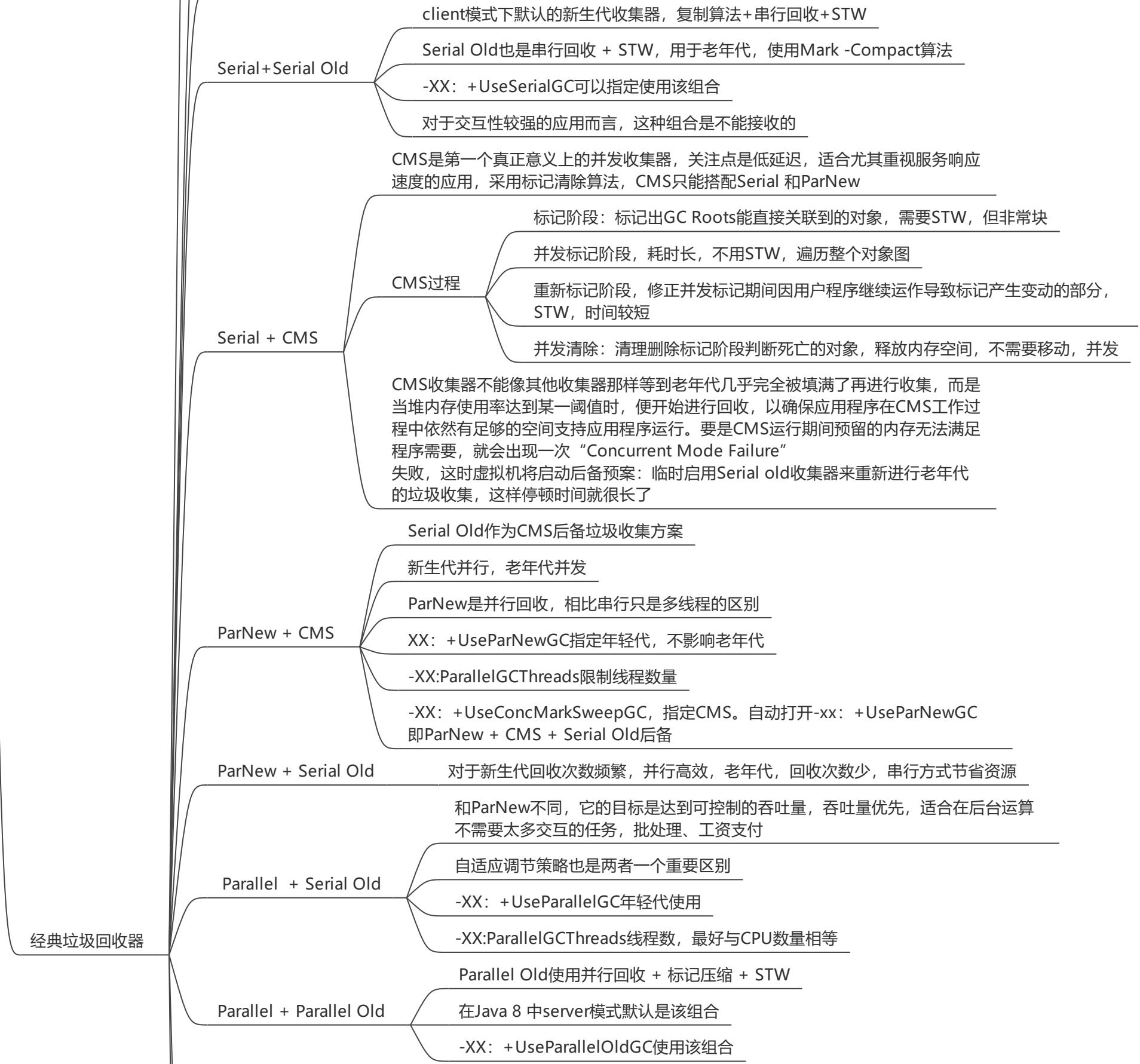
青色虚线，JDK14删除CMS

垃圾回收器分类

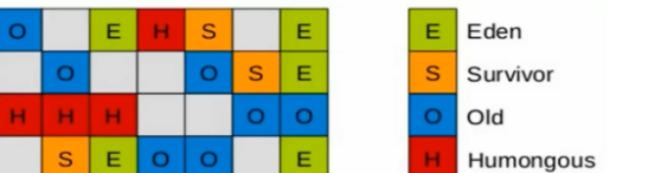
组合关系

查看默认垃圾收集器

-XX:+PrintCommandLineFlags:



收集器发展最前沿成果之一，JDK9之后的默认垃圾回收器，JDK8中可以使用-xx：  
+UseG1GC来启用。-XX:G1HeapRegionSize设置每个Region的大小。  
XX:MaxGCPauseMillis期望的最大GC停顿时间，默认200ms



G1将内存划分为约2048个region。内存的回收是以region作为基本单位的。在延时可控的情况下获得尽可能高的吞吐量，跟踪各个Region中的垃圾堆积的价值大小（回收获得空间和回收所用时间），所以取名为Garbage-First

面向服务端应用的垃圾收集器，主要针对配备多核CPU及大容量内存的机器，以极高概率满足GC停顿时间的同时，还兼具高吞吐量的性能特征

并行性与并发性

多个GC线程  
G1线程与应用线程交替同时执行

依然是分代性垃圾回收器，区分年轻代和老年代，只是区域不是连续空间，也不再坚持固定大小和固定数量。由于分区的原因，使得对于全局停顿情况可控制，维护一个各个region垃圾价值大小的优先列表，每次根据允许的收集时间优先回收价值最大的Region，未必做到CMS那么低的停顿，但是最差情况要好很多。

G1

为了垃圾收集产生的内存占用（Footprint）和程序运行时的额外执行负载（overload）都要比CMS要高。G1更适合于堆内存比较大的情况

Region之间是复制算法，整体上实际上可以看做是Mark-Compact算法

增加了一种新的内存区域Humongous存放大对象，超过1.5个region放到H

G1大多数行为把H区作为老年区看待  
每个Region通过指针碰撞分配空间

YoungGC MixedGC FullGC

回收模式

回收过程

① 当年轻代的Eden区用尽时开始年轻代回收过程；G1的年轻代收集阶段是一个并行的独占式收集器。然后从年轻代区间移动存活对象到Survivor区间或者老年区间，也有可能是两个区间都会涉及。

② 当堆内存使用达到一定值（默认45%）时，开始老年代并发标记过程。（类似CMS）

③ 标记完成马上开始混合回收过程

如果上述过程不能正常工作，比如G1复制存活对象没有空闲内存分段可用，回退到FullGC。

记忆集Remembered Set

一个Region中的对象可能被其他Region中的对象引用，双向卡表记录对其他region对象的引用，同时记录其他region中对象对本region中对象的引用

解决的是GC Roots不全的问题

如果内存小于100M或者单核、单机程序，没有停顿时间要求，使用串行收集器Serial

如果多CPU，高吞吐量，允许停顿时间超过1s，选择并行，Parallel追求高吞吐

如果是多CPU，追求低停顿时间，需要快速响应，使用并发收集器CMS G1

怎么选择垃圾回收器？

- ① YGC先进行STW，G1创建Collection Set (CSet回收集)。
- ② 第一阶段，扫描根
- ③ 第二阶段，更新RSet 准确反应老年代对所在内存分段中对象的引用
- ④ 第三阶段，处理RSet 识别被老年代对象指向的Eden中的对象，这些视为存活对象
- ⑤ 第四阶段，复制对象 遍历对象树，存活对象被复制到survivor，未达到阈值年龄+1，达到进入老年代。如果survivor空间不够，直接到老年代
- ⑥ 第五阶段，处理引用 处理软、弱、虚等引用
- ① 标记直接可达对象，触发youngGC
- ② 根区域扫描
- ③ 并发标记，整个堆中进行并发标记（并发）
- ⑤ 再次标记，由于并发执行，需要修改上一次的标记结果。G1采用比CMS更快的SATB
- ⑥ 独占清理，计算各个区域存活对象、回收比例，进行排序识别可以混合回收的区域
- ⑦ 并发清理阶段，识别并清理完全空闲的区域

老年代的G1回收器和其他GC不同，G1的老年代回收器不需要整个老年代被回收，一次只需要扫描/回收一小部分老年代的Region就可以了。年轻代一起被回收

回收整个YoungGC和部分Old Region

老年代的回收分8次回收（不一定进行8次，如果发现可以回收的垃圾占堆内存的比例低于10%，不再进行混合回收），G1优先回收垃圾多的内存分段，垃圾占内存分段比例高的，先被回收，有一个阈值决定内存分段是否被回收，默认65%。