

# GC Algorithm回收算法

- Mark-Sweep (标记清除)
- Coping (拷贝)
- Mark-Compact (标记压缩)

1. 垃圾回收器的发展路线：是随着内存越来越大的过程而演化

从分代算法演化到不分代算法

Serial算法 几十兆

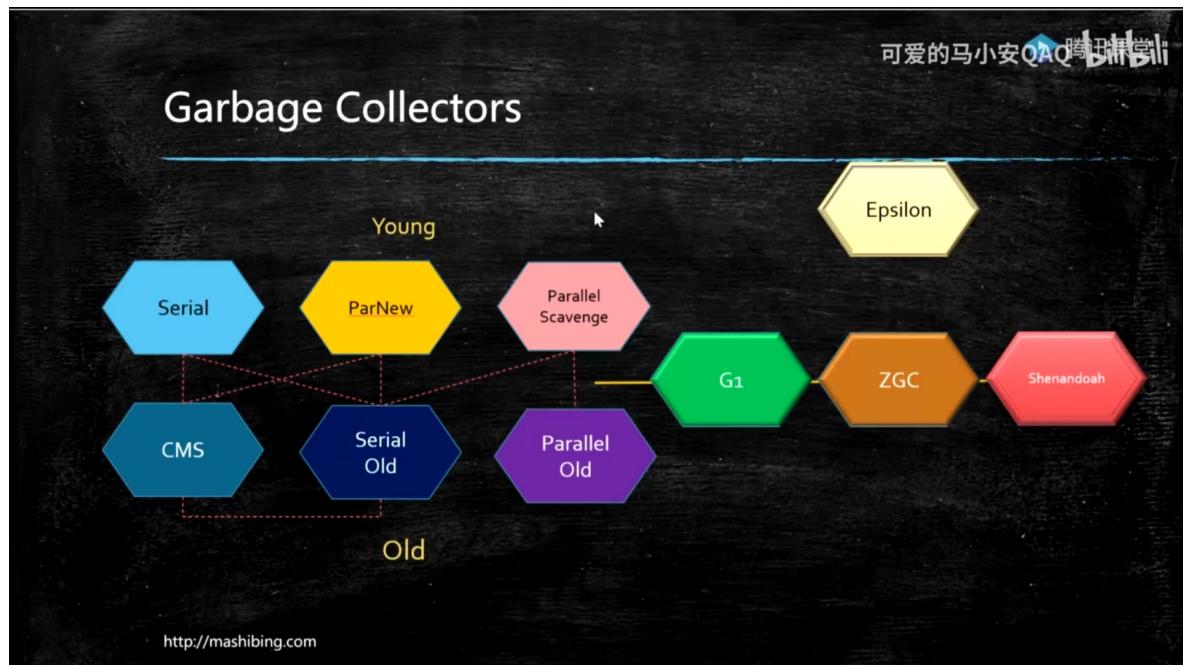
Parallel算法 几个G

CMS 几十个G-承上启下 开始并发回收 --三色标记

G1 上百G内存-逻辑分代，物理不分代

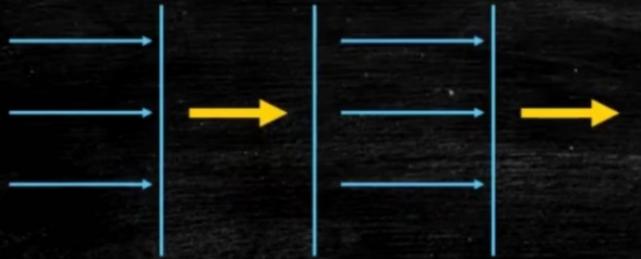
ZGC - Shenandoah --4T(  $2^{42}$  ) 逻辑物理都不分代

Epsilon 啥也不干 (调试，确认不用GC参与就能干完活)



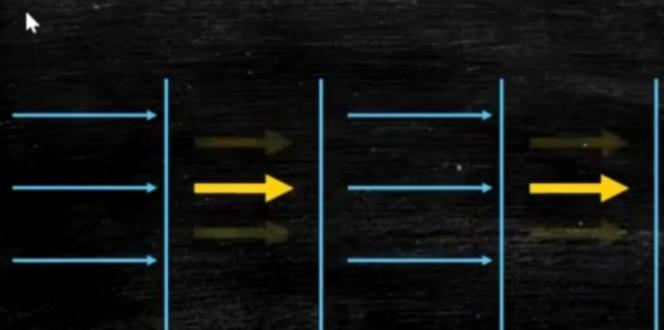
## Serial

- a stop-the-world, copying collector which uses a single GC thread



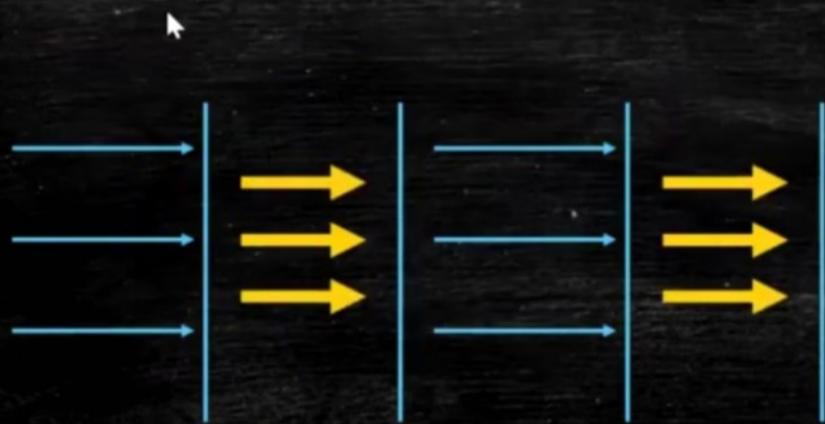
## Serial Old

- a stop-the-world, mark-sweep-compact collector that uses multiple GC threads.



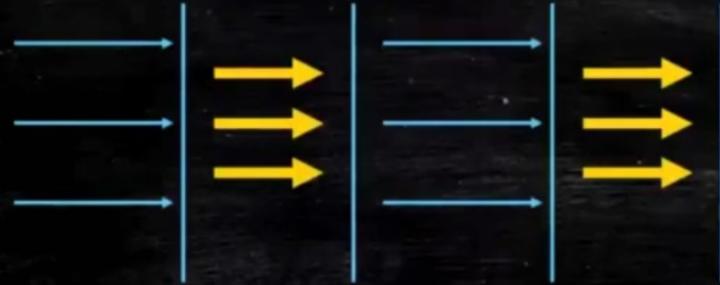
## Parallel Scavenge

- a stop-the-world, copying collector which uses multiple GC threads



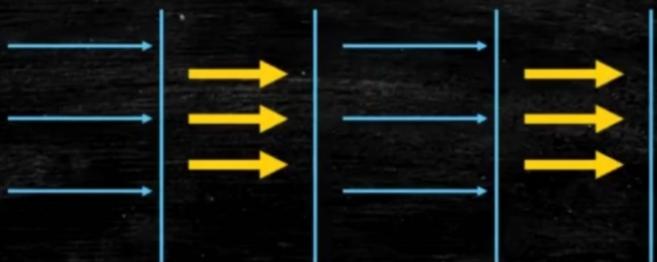
## parallel old

- a compacting collector that uses multiple GC threads.



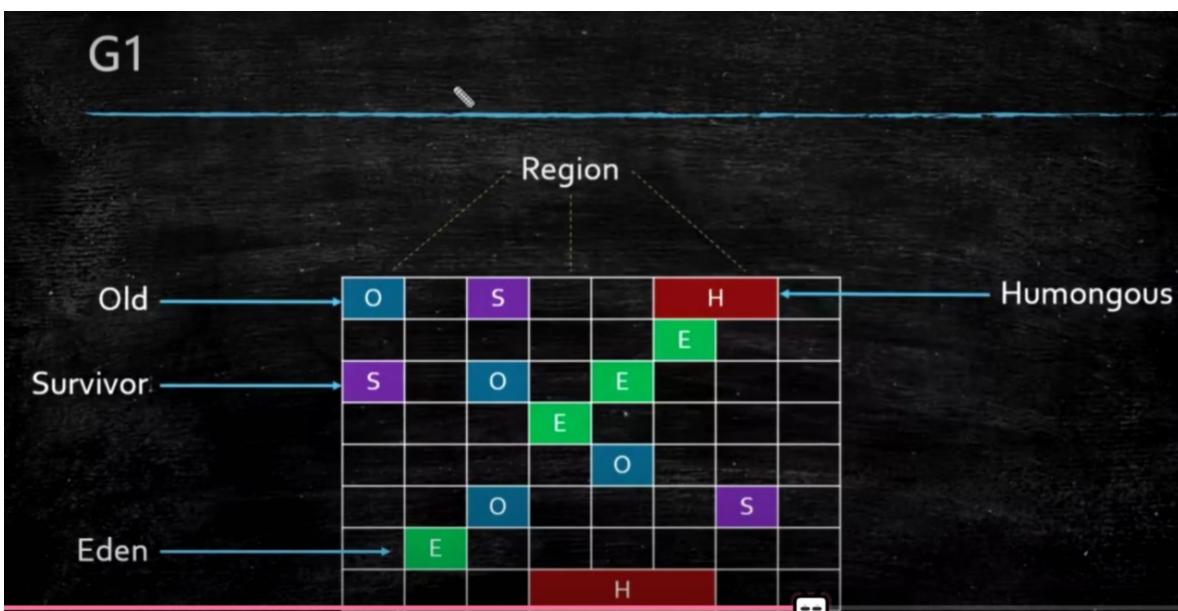
## ParNew

- a stop-the-world, copying collector which uses multiple GC threads
- It differs from "Parallel Scavenge" in that it has enhancements that make it usable with CMS
- For example, "ParNew" does the synchronization needed so that it can run during the concurrent phases of CMS.



CMS(Concurrent Mark-Sweep):垃圾回收线程与工作线程一起执行。

## 堆内存逻辑分区



```
java -XX:+PrintFlagsWithComments //只有debug版本能用
```

```
java -XX:+PrintFlagsFinal
```

## YGC FGC

- YGC

Young GC 或 Minor GC

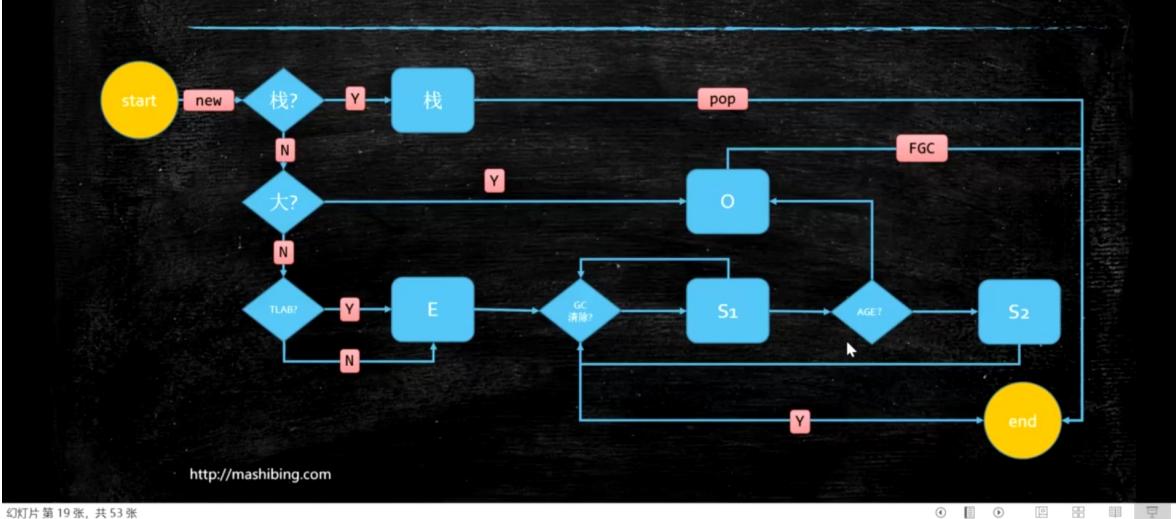
Eden区不足

- FGC

Full GC或Major GC

Old空间不足

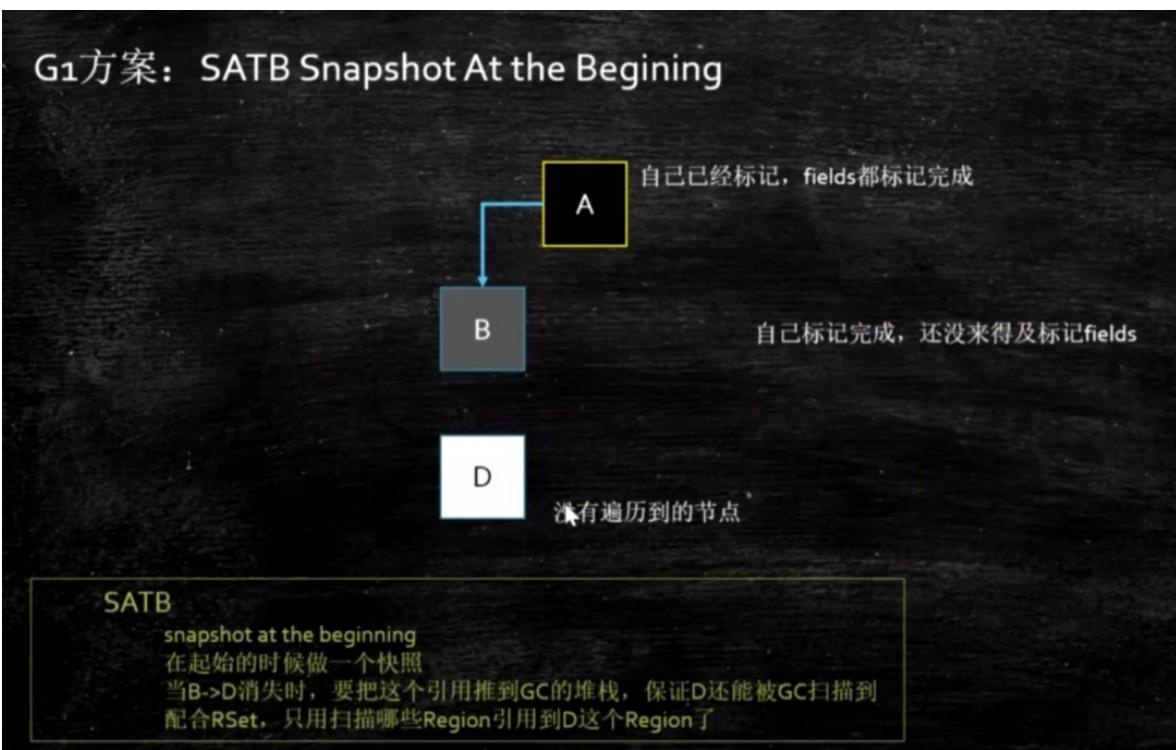
## 总结



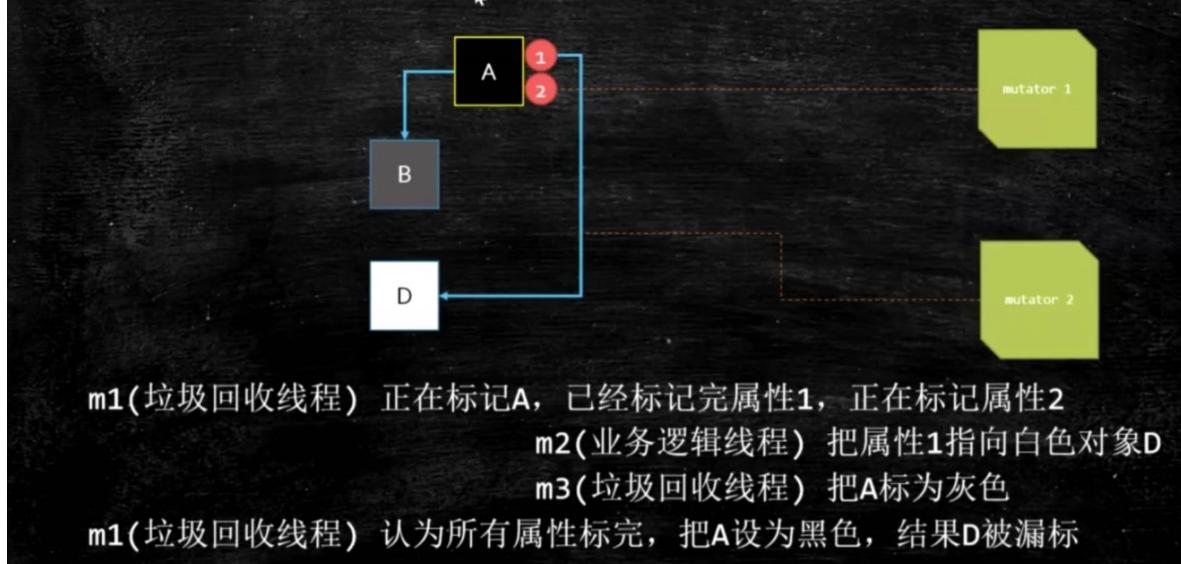
幻灯片第 19 张，共 53 张

解决没有被标记到的节点,

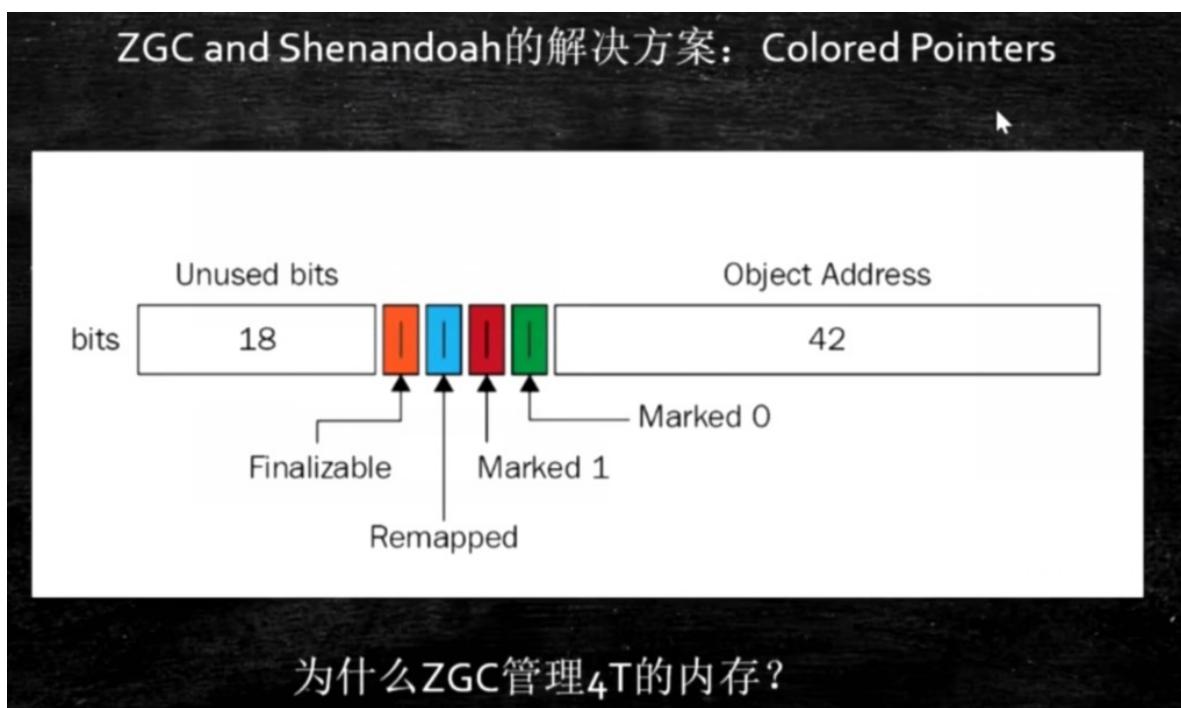
### 三色方案

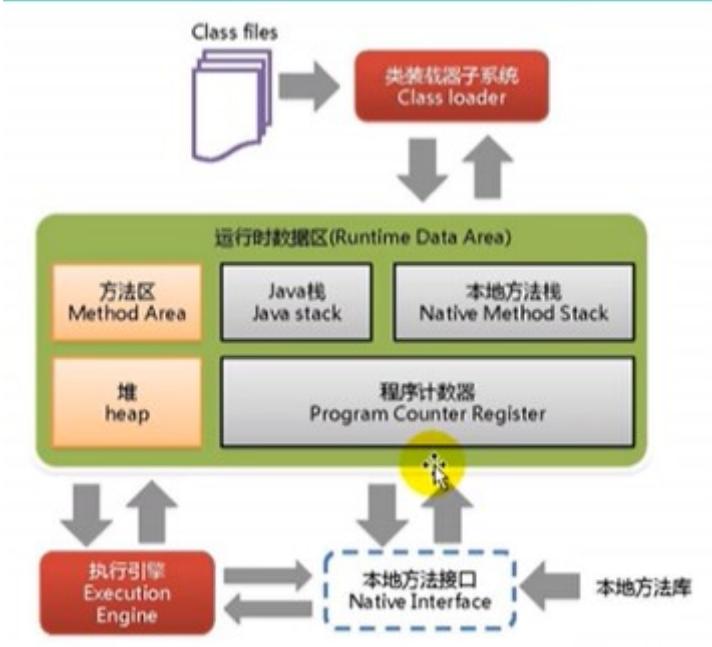


CMS方案: Incremental Update的非常隐蔽的问题:  
并发标记, 产生漏标



### ZGC颜色指针





## 一.JVM与Java体系

### JVM的架构模型

Java编译器输入的指令流基本上是一种**基于栈的指令集架构**,另外一种指令集架构则是**基于寄存器的指令集架构**.

#### 基于栈式架构的特点:

- 设计和实现更简单,适用于资源受限的系统
- 避开了寄存器的分配难题:使用零地址指令方式分配
- 指令流中的指令大部分是零地址指令,其执行过程依赖于操作栈.指令集更小,编译器容易实现.
- **不需要硬件支持,可移植性更好,更好实现跨平台**

#### 基于寄存器架构的特点

- 典型的应用是x86的二进制指令集.
- 指令集架构则完全依赖硬件,可移植性差
- 性能优秀和执行更高效
- 花费更少的指令去完成一项操作.

### 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

一个运行中的Java虚拟机有着一个清晰的任务:执行Java程序

程序开始执行时他才运行,程序结束时他就停止.

执行一个所谓的Java程序的时候,真真正正在执行的是一个叫**Java虚拟机的进程**.

### JVM发展历程

**Sun Classic VM**,第一款商用的Java虚拟机,只提供了解释器,没有JIT编译器,jdk1.4淘汰

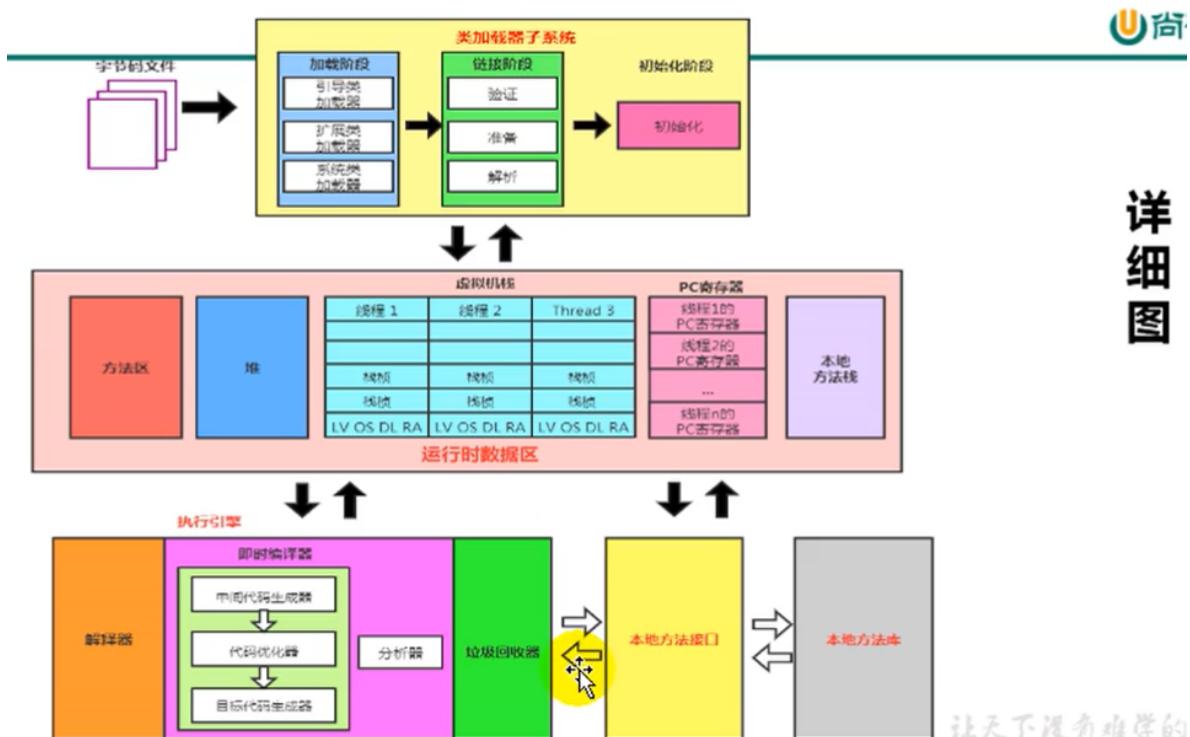
**Exact VM**:

**HotSpot**: jdk1.3就成为Java默认的虚拟机

**BEA的JRockit**: 专注于服务器端应用,BEA公司被Oracle公司收购.

**IBM的J9**

## 二.类加载子系统(Class Loader)



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.

ClassLoader只负责class文件的加载,至于它是否可以运行,则有Execution Engine决定.

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).

## 类的加载过程



**加载:**

- 1.通过一个类的全限定名获取定义此类的二进制字节流.
- 2.将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构.
- 3.在内存中生成一个代表这个类的java.lang.Class对象,作为方法区这个类的各种数据的访问入口.

**验证(Verify) :**

- 目的在于确保Class文件的字节流中包含信息符合当前虚拟机要求，保证被加载类的正确性，不会危害虚拟机自身安全。
- 主要包括四种验证，文件格式验证，元数据验证，字节码验证，符号引用验证。

**准备(Prepare) :**

- 为类变量分配内存并且设置该类变量的默认初始值，即零值。
- 这里不包含用final修饰的static，因为final在编译的时候就会分配了，准备阶段会显式初始化；
- 这里不会为实例变量分配初始化，类变量会分配在方法区中，而实例变量是会随着对象一起分配到Java堆中。

**解析(Resolve) :**

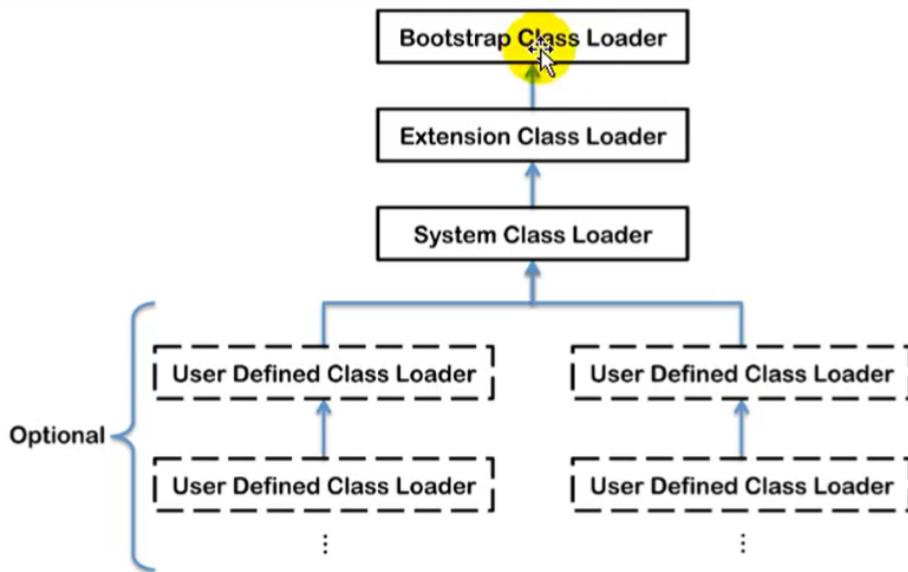
- 将常量池内的符号引用转换为直接引用的过程。
- 事实上，解析操作往往伴随着JVM在执行完初始化之后再执行。
- 符号引用就是一组符号来描述所引用的目标。符号引用的字面量形式明确定义在《java虚拟机规范》的class文件格式中。直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。
- 解析动作主要针对类或接口、字段、类方法、接口方法、方法类型等。对应常量池中的CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info等。

**初始化:****初始化**

- 初始化阶段就是执行类构造器方法<clinit>()的过程。
- 此方法不需定义，是javac编译器自动收集类中的所有类变量的赋值动作和静态代码块中的语句合并而来。
- 构造器方法中指令按语句在源文件中出现的顺序执行。
- <clinit>()不同于类的构造器。(关联：构造器是虚拟机视角下的<init>())
- 若该类具有父类，JVM会保证子类的<clinit>()执行前，父类的<clinit>()已经执行完毕。
- 虚拟机必须保证一个类的<clinit>()方法在多线程下被同步加锁。

**类加载器的分类**

- JVM支持两种类型的类加载器，分别为**引导类加载器** (Bootstrap ClassLoader) 和**自定义类加载器** (User-Defined Class loader)。
- 从概念上来讲，自定义类加载器一般指的是程序中由开发人员自定义的一类类加载器，但是Java虚拟机规范却没有这么定义，而是**将所有派生于抽象类ClassLoader的类加载器都划分为自定义类加载器**。



这里的四者之间的关系是包含关系。不是上层下层，也不是子父类的继承关系。

## 虚拟机自带的加载器

### 1. 启动类加载器（引导类加载器，Bootstrap ClassLoader）

这个类加载使用C/C++语言实现的，嵌套在JVM内部

它用来加载Java的核心库（JAVA\_HOME/jre/lib/rt.jar、resources.jar或sun.boot.class.path路径下的内容），用于提供JVM自身需要的类

并不继承自java.lang.ClassLoader，没有父加载器

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。

出于安全考虑，BootStarp启动类加载器只加载包名为java, javax, sun等开头的类。

### 2. 扩展类加载器（Extension ClassLoader）

Java语言编写，由sun.misc.Launcher\$ExtClassLoader实现。

派生于ClassLoader类

父类加载器为启动类加载器

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，也会自动由启动类加载器加载。

### 3. 应用程序类加载器（系统类加载器，AppClassLoader）

java语言编写，由sun.misc.Launcher\$AppClassLoader实现

派生于ClassLoader类

父类加载器为扩展类加载器

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

该类加载器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。

通过ClassLoader#getSystemClassLoader()方法可以获取到该类加载器。

## 用户自定义类加载器

防止源码泄露

## 关于ClassLoader

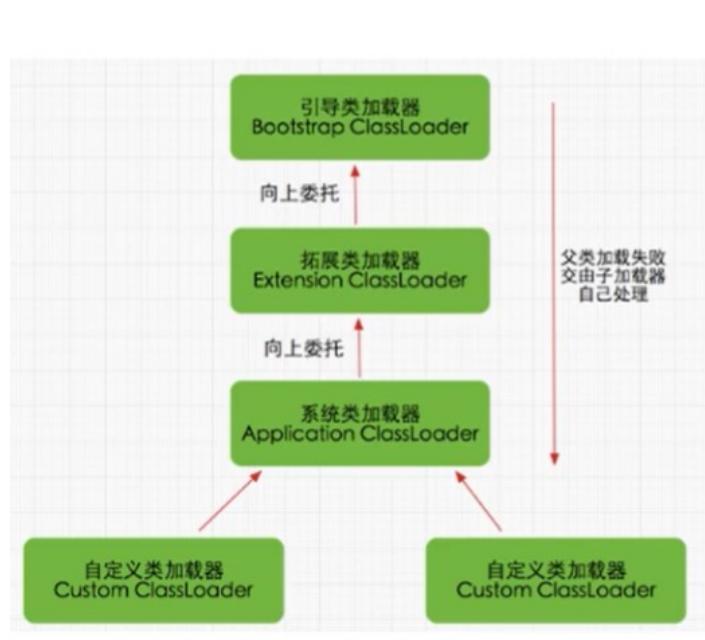
ClassLoader类，它是一个抽象类，其后所有的类加载器都继承自ClassLoader（**不包括启动类加载器**）

## 双亲委派机制

Java虚拟机对class文件采用的是**按需加载**的方式，也就是说当需要使用该类时才会将他的class文件加载到内存中生成class对象。而且加载某个类的class文件时，Java虚拟机采用的是**双亲委派模式**，即把请求交由父类处理，它是一种任务委派模式。

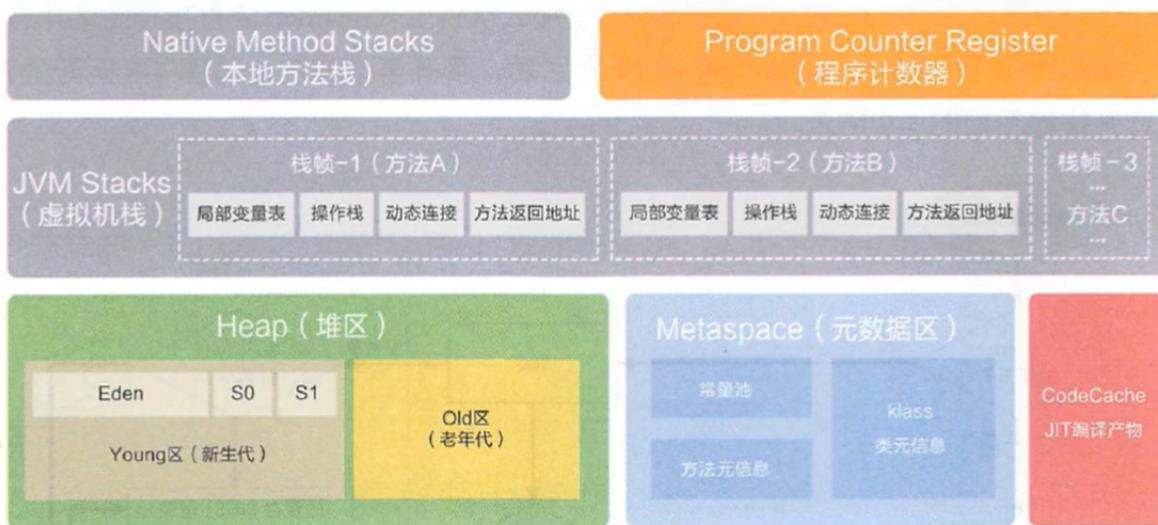
### 工作原理

- 1.如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行
- 2.如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器。
- 3.如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派机制。



避免类的重复加载，保护程序的安全，防止核心API被随意的篡改。

## 三.运行时数据区



## 1.程序计数器(PC寄存器)

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.在JVM规范中,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.

使用PC寄存器存储字节码指令地址有什么用?

因为CPU需要不停的切换各个线程,这时候切换回来以后,需要知道接着从哪开始继续执行.

PC寄存器为什么会被设定为线程私有?

为了能够准确的记录各个线程正在执行的当前字节码指令地址,最好的办法就是为每一个线程分配一个PC寄存器.

## 2.虚拟机栈

栈是运行时的单位,而堆是存储的单位

Java虚拟机栈(Java Virtual Machine Stack),早期也叫Java栈.

每个线程在创建时都会创建一个虚拟机栈,内部保存一个个的栈帧(Stack Frame),对应着一个一个的Java方法调用.

生命周期和线程一致.

主管Java程序的运行,它保存方法的局部变量(8种基本数据类型,对象的引用地址),部分结果,并参与方法的调用和返回.

### 栈中可能出现的异常

Java虚拟机允许Java栈的大小是动态的或者是固定不变的.

- 如果是固定大小的Java虚拟机栈,那每个线程的Java虚拟机栈容量可以在线程创建的时候独立指定.如果线程请求分配的栈容量超过Java虚拟机允许的最大容量,Java虚拟机将会抛出一个`StackOverflowError`异常
- 如果是动态扩展的,并且在尝试扩展的时候无法申请到足够的内存,那Java虚拟机将会抛出一个`OutOfMemoryError`异常.

可以使用参数`-Xss1024k`来设置线程的最大栈空间,栈的大小直接决定了函数调用的最大可达深度.

### 栈中存储的数据

每个线程都有自己的栈,栈中的数据都是以栈帧的格式存在.

在这个线程上正在执行的每个方法都各自对应一个栈帧(Stack Frame).

栈帧是一个内存区块,是一个数据集,维系着方法执行过程中的各种数据信息.

在一条活动的线程上只有一个活动的栈帧有效,称为当前栈帧.

### 栈运行原理

不同线程中所保护的栈帧是不允许存在相互引用的,即不可能在一个栈帧之中引用另外一个线程的栈帧.

如果当前方法调用了其他方法,方法返回之际,当前栈帧会传回此方法的执行结果给前一个栈帧,接着,虚拟机会丢弃当前栈帧,使得前一个栈帧重新成为当前栈帧.

Java方法有两种返回函数的方式,一种是正常的函数返回,使用`return`指令;另外一种是抛出异常.不管使用哪种方式,都会导致栈帧被弹出.

### 栈帧的内部结构

- 局部变量表(Local Variables)

- 操作数栈(Operand Stack),或表达式栈
- 动态链接(Dynamic Linking),或指向运行时常量池的方法引用
- 方法返回地址(Return Address),或方法正常退出或者异常退出的定义
- 一些附加信息

## 局部变量表(Local Variables)

- 局部变量表也被称为局部变量数组或本地变量表  
局部变量表最基本的单位是slot(变量槽).32位以内的类型只占用一个slot
- 定义为一个数字数组,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.
- 由于局部变量表是建立在线程的栈上,是线程的私有数据,因此不存在数据安全问题.
- 局部变量表所需的容量大小是在编译期确定下来的,并保存在方法的Code属性的maximum local variables数据项中.在方法运行期间是不会改变局部变量表的大小的.
- 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

## 操作数栈(Operand Stack)

- 在方法执行过程中,根据字节码指令,往栈中写入数据或提取数据,即入栈/出栈.
- Java虚拟机的解释引擎是基于栈的执行引擎,其中的栈指的是操作数栈.
- 操作数栈,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.

## 动态链接(Dynamic Linking)

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.比如invokedynamic指令.

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.比如:描述一个方法调用了另一个方法时,就是通过常量池中指向方法的符号引用表示的,那么,动态链接的作用就是为了将这些符号引用转换为调用方法的直接引用.

## 方法的调用

在JVM中,将符号引用转换为调用方法的直接引用与方法的绑定机制相关.

- 静态链接

当一个字节码文件被装载进JVM内部时,如果被调用的目标方法在编译器可知,且运行期保存不变时,这种情况下将调用方法的符号引用转换为直接引用的过程称为静态链接

- 动态链接

如果被调用的方法在编译期无法被确定下来,也就是说,只能在程序运行期将调用方法的符号引用转换为直接引用,由于这种引用转换过程具备动态性,因此也被称为动态链接.

## 方法返回地址

存放调用该方法的pc寄存器的值.

本质上,方法的退出就是当前栈帧出栈的过程,此时,需要恢复上层方法的局部变量表,操作数栈,将返回值压入调用者栈帧的操作数栈,设置pc寄存器等,让调用者方法继续执行下去.

## 3.本地方法接口

### 什么是本地方法?

简单讲,一个 Native Method 就是一个Java调用非Java代码的接口.一个Native Method是这样一个Java方法:该方法的实现由非Java语言实现,比如C.这个特征并非Java所特有,很多其他的编程语言都有这一机制,比如在C++中,可以使用extent "C" 告知c++编译器去调用一个C的函数.

本地接口的作用是融合不同的编程语言为Java所用,它的初衷是融合C/C++程序.

### 为什么使用Native Method?

有时Java应用需要与Java外面的环境交互,这是本地方法存在的主要原因.

## 4.本地方法栈(Native Method Stack)

Java虚拟机栈用于管理Java方法的调用,而本地方法栈用于管理本地方法的调用.

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.

### 5.堆(heap) ◇

- 一个JVM实例只存在一个堆内存,堆也是Java内存管理的核心区域.Java堆区在JVM启动的时候即被创建,其间大小也就确定了.是JVM管理的最大一块内存空间,堆内存的大小是可以调节的.**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是连续的.**
- 所有的线程共享堆.
- 几乎所有的对象实例以及数组都应当在运行时分配在堆上
- 数组和对象可能永远不会存储在栈上,因为栈帧中保存引用,这个引用指向对象或数组在堆中的位置.
- 在方法结束后,堆中的对象不会马上被移除,仅仅在垃圾收集的时候才会被移除.
- 堆是GC执行垃圾回收的重点区域.

#### 堆的内存细分

### 堆的核心概述 : 内存细分



现代垃圾收集器大部分都基于分代收集理论设计 , 堆空间细分为 :

- 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**
  - Young Generation Space 新生区 Young/New
  - 又被划分为Eden区和Survivor区
- Tenure generation space 养老区 Old/Tenure
- Permanent Space 永久区 Perm

- 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**
  - Young Generation Space 新生区 Young/New
  - 又被划分为Eden区和Survivor区
- Tenure generation space 养老区 Old/Tenure
- Meta Space 元空间 Meta

约定: 新生区↔新生代↔年轻代 养老区↔老年区↔老年代 永久区↔永久代

#### 设置堆空间大小的参数

-Xms 用来设置堆空间(年轻代+老年代)的初始内存大小

-X 是jvm的运行参数

ms memory start

-Xmx 用来设置堆空间的最大内存

默认堆空间最大: 物理电脑内存大小/4

默认堆空间初始值: 物理电脑内存大小/64

手动设置: -Xms600m -Xmx600m

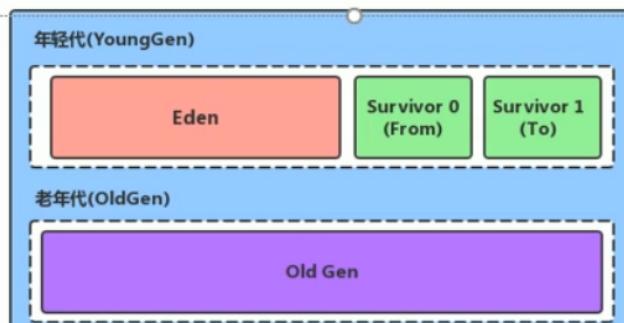
开发中建议把初始值和最大值设置为一样的,

## 年轻代与老年代

### 年轻代与老年代



- 存储在JVM中的Java对象可以被划分为两类:
  - 一类是生命周期较短的瞬时对象，这类对象的创建和消亡都非常迅速
  - 另外一类对象的生命周期却非常长，在某些极端的情况下还能够与JVM的生命周期保持一致。
- Java堆区进一步细分的话，可以划分为年轻代（YoungGen）和老年代（OldGen）
- 其中年轻代又可以划分为Eden空间、Survivor0空间和Survivor1空间（有时也叫做from区、to区）。



### 年轻代与老年代



下面这参数开发中一般不会调:



- 配置新生代与老年代在堆结构的占比。

- 默认-XX:NewRatio=2，表示新生代占1，老年代占2，新生代占整个堆的1/3
- 可以修改-XX:NewRatio=4，表示新生代占1，老年代占4，新生代占整个堆的1/5

## 对象分配过程

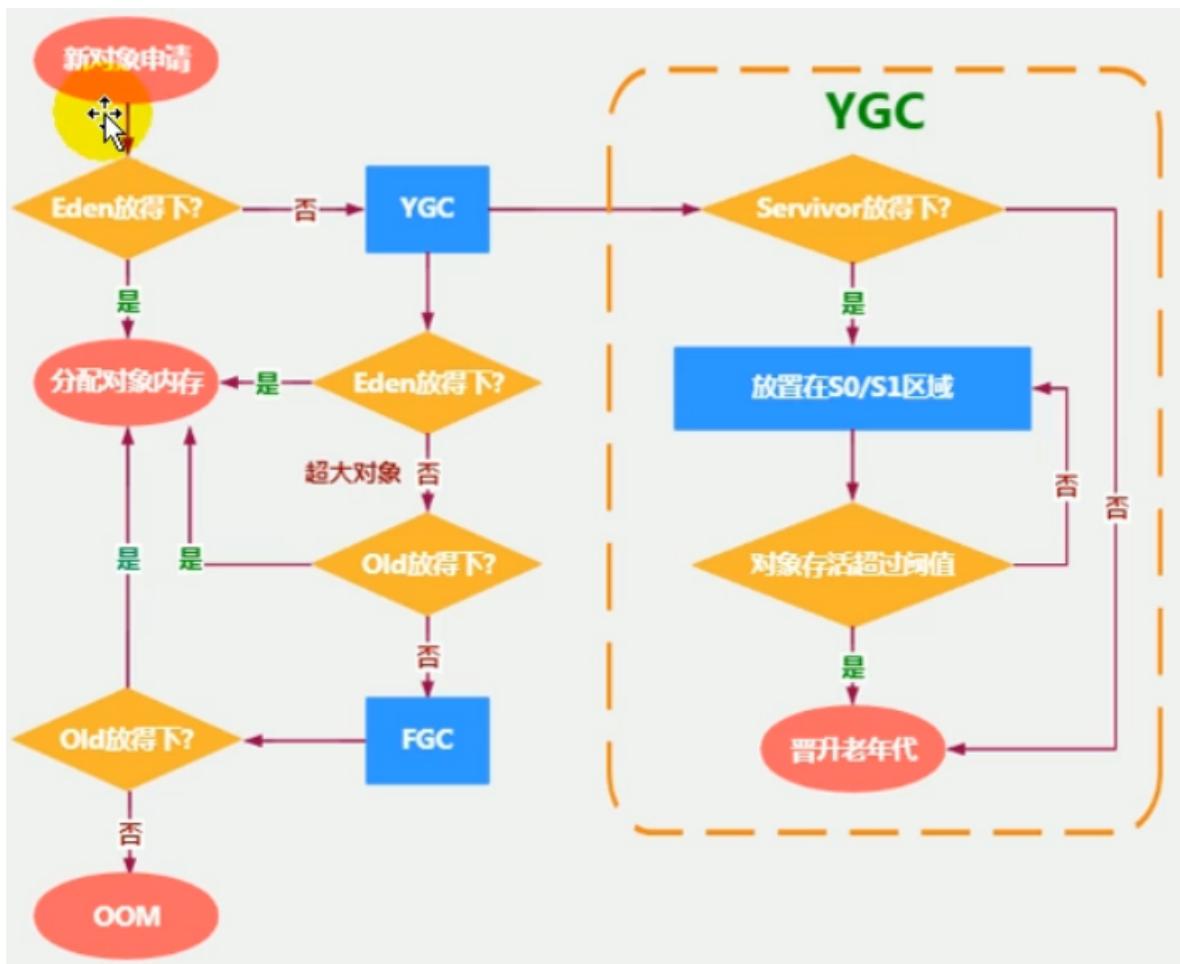
几乎所有的Java对象都是在Eden区new出来的.

绝大部分的Java对象的销毁都在新生代进行了.

针对幸存者s0,s1区的总结: 复制之后有交换,谁空谁是to

关于垃圾回收: 频繁在新生区收集,很少在养老区收集,几乎不在永久区/元空间收集.

YGC:



### Minor GC, Major GC, Full GC

- 新生代收集(Minor GC/Young GC):只是新生代(Eden/S0,S1)的垃圾收集,Eden满会触发Young GC, Survivor满不会触发GC
- 老年代收集(Major GC/Old GC):只是老年代的垃圾收集  
目前,只有CMS GC会有单独收集老年代的行为  
注意:很多时候Majro GC和Full GC混淆使用,需要具体分辨是老年代回收还是整堆回收.  
混合收集(Mixed GC):收集整个新生代以及部分老年代的垃圾收集,目前只有G1 GC会有这种行为.
- 整堆收集(Full GC):收集整个java堆和方法区的垃圾收集.

### TLAB(Thread Local Allocation Buffer)

- 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为快速分配策略.

### 堆空间的参数设置

[官网](#)

```
1 -XX:+PrintGCDetails
2 Enables printing of detailed messages at every GC. By default, this option
is disabled.
3
4 -XX:NewRatio=ratio
5 Sets the ratio between young and old generation sizes. By default, this
option is set to 2. The following example shows how to set the young/old
ratio to 1:
6 -XX:NewRatio=1
7
8 -XX:SurvivorRatio=ratio
9 Sets the ratio between eden space size and survivor space size. By default,
this option is set to 8. The following example shows how to set the
eden/survivor space ratio to 4:
10 -XX:SurvivorRatio=4
11
```

## 逃逸分析

逃逸分析的基本行为就是分析对象动态作用域:

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.例如作为参数传递到其他地方去.

没有发生逃逸的对象,则可以分配到栈上,随着方法执行的结束,栈空间就被移除.因此不用在堆上new对象,  
不用GC.

```
1 | -XX:+DoEscapeAnalysis 开启逃逸分析 默认已开启
```

## 逃逸分析代码优化



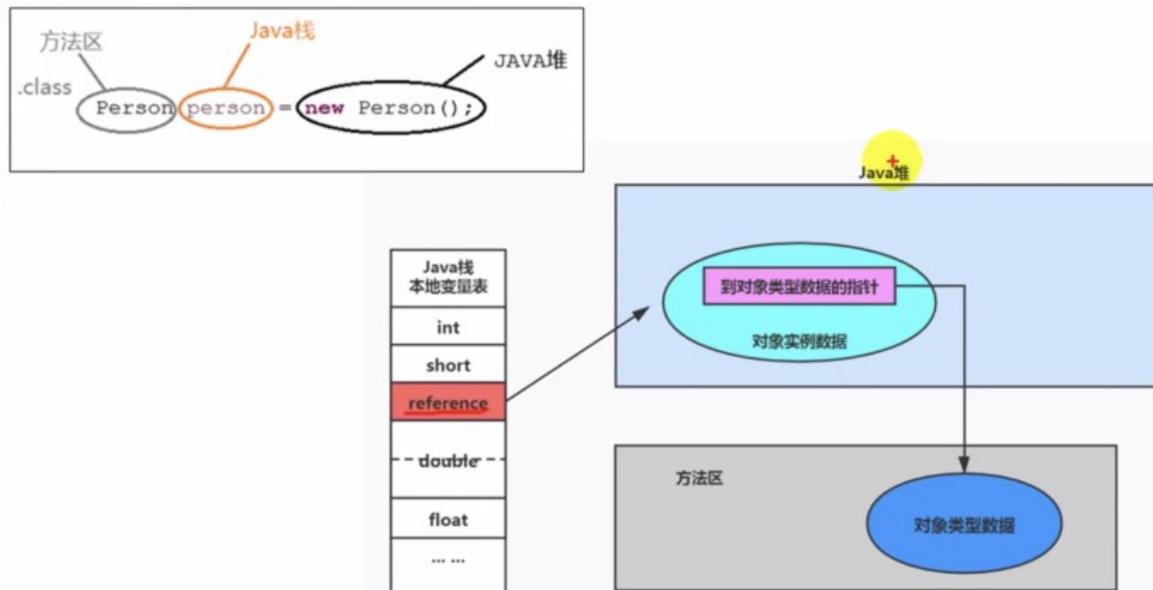
使用逃逸分析, 编译器可以对代码做如下优化:

**一、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逃逸, 对象可能是栈分配的候选, 而不是堆分配。

**二、同步省略。**如果一个对象被发现只能从一个线程被访问到, 那么对于这个对象的操作可以不考虑同步。

**三、分离对象或标量替换。**有的对象可能不需要作为一个连续的内存结构存在也可以被访问到, 那么对象的部分(或全部)可以不存储在内存, 而是存储在CPU寄存器中。

## 6.方法区



《Java虚拟机规范》中明确说明：“尽管所有的方法区在逻辑上属于堆的一部分，但一些简单的实现可能不会选择去进行垃圾收集或者进行压缩。”但对于HotSpotJVM而言，方法区还有一个别名叫做Non-Heap（非堆），目的就是要和堆分开。所以，**方法区看作是一块独立于Java堆的内存空间**

- 方法区（Method Area）与Java堆一样，是各个线程共享的内存区域。
- 方法区的大小决定了系统可以保存多少个类，如果系统定义了太多的类，导致方法区溢出，虚拟机同样会抛出内存溢出错误，`java.lang.OutOfMemoryError: PermGen space` 或者 `java.lang.OutOfMemoryError: Metaspace`，加载大量的第三方jar包，tomcat部署的工程过多（30-50个），大量动态的生成反射类

**在JDK7及以前，习惯上把方法区称为永久代，JDK8使用元空间取代了永久代。**

**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=100m` 和 `-XX:MaxMetaspaceSize=100m` 指定，Windows下 `-XX:MetaspaceSize` 是21M，`-XX:MaxMetaspaceSize` 的值是-1，即没有限制。

### 方法区内部结构

**方法区存储内容：**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**。

#### 类型信息：

对每个加载的类型(类Class,接口Interface,枚举Enum,注解annotation),JVM必须在方法区中存储以下类型信息：

- ①这个类型的完整有效名称(全名=包名+类名)
- ②这个类型直接父类的完整有效名(对于interface或是java.lang.Object,都没有父类)
- ③这个类型的修饰符(public,abstract,final的某个子集)
- ④这个类型直接接口的一个有序列表。

#### 域(Field)信息

JVM必须在方法区中保存类型的所有域的相关信息以及域的声明顺序。

域的相关信息包括：域名称,域类型,域修饰符(public,private,protected..)

#### 方法(Method)信息

方法名称,方法返回值,方法参数的数量和类型,方法的修饰符,异常表...

### 常量池:

一个有效的字节码文件中除了包含类的版本信息,字段,方法以及接口等描述信息外,还包含一项信息那就是**常量池表**(Constant Pool Table),包括各种字面量和对类型,域和方法的符号引用.

一个Java源文件中的类,接口,编译后产生一个字节码文件.而Java中的字节码需要数据支持,通常这种数据会很大以至于不能直接存到字节码里,换另一种方式,可以存到常量池,这个字节码包含了指向常量池的引用.在动态链接的时候会用到运行时常量池.

常量池,可以看作是一张表,虚拟机指令根据这张常量表找到要执行的类名,方法名,参数类型,字面量等类型

### 运行时常量池(Runtime Constant Pool)是方法区的一部分

常量池表(Constant Pool Table)是**Class文件的一部分**,用于存放编译期生成的各种字面量和符号引用,这部分内容将在**类加载后存放到方法区的运行时常量池中**.

JVM为每个已加载的类型(类或接口)都维护一个常量池.池中的数据项就像数组项一样,是通过**索引访问的**.

运行时常量池,相对于Class文件常量池的另一个重要特征是:**具备动态性**

### 方法区的演进细节

The diagram is titled "方法区的演进细节" (Evolution of the Method Area) and features a green header bar with the logo of "尚硅谷". Below the title, there is a circular navigation interface with arrows and a central icon. The main content area is divided into two sections: a top section with numbered points and a bottom section with a table comparing three Java versions (jdk1.6, jdk1.7, and jdk1.8).

**1. 首先明确: 只有HotSpot才有永久代。**  
BEA JRockit、IBM J9等来说, 是不存在永久代的概念的。原则上如何实现方法区属于虚拟机实现细节, 不受《Java虚拟机规范》管束, 并不要求统一。

**2. Hotspot中方法区的变化:**

jdk1.6及之前	有永久代( <b>permanent generation</b> ), 静态变量存放在永久代上
jdk1.7	有永久代, 但已经逐步“去永久代”, 字符串常量池、静态变量移除, 保存在堆中
jdk1.8及之后	无永久代, 类型信息、字段、方法、常量保存在本地内存的元空间, 但字符串常量池、静态变量仍在堆

让天下没有难学的技术

### 永久代为什么要被元空间替换?

永久代的元数据信息被移到了一个与**堆不相连的本地内存区域**,这个区域叫做元空间

由于类的元数据分配在本地内存中,元空间的最大可分配空间就是系统可用内存空间.

这项改动是很有必要的,原因有:

- 1.为永久代设置空间大小是很难确定的.
- 2.对永久代进行调优是很困难的.

### StringTable为什么要调整?

jdk7中将StringTable放到了堆空间中.因为永久代的回收效率很低,在full GC的时候才会触发,而Full GC是老年代的空间不足,永久代不足时才触发,这就导致了StringTable回收效率不高.而我们开发中会有大量的字符串被创建,回收效率低,导致永久代内存不足,放到堆里,能及时回收内存.

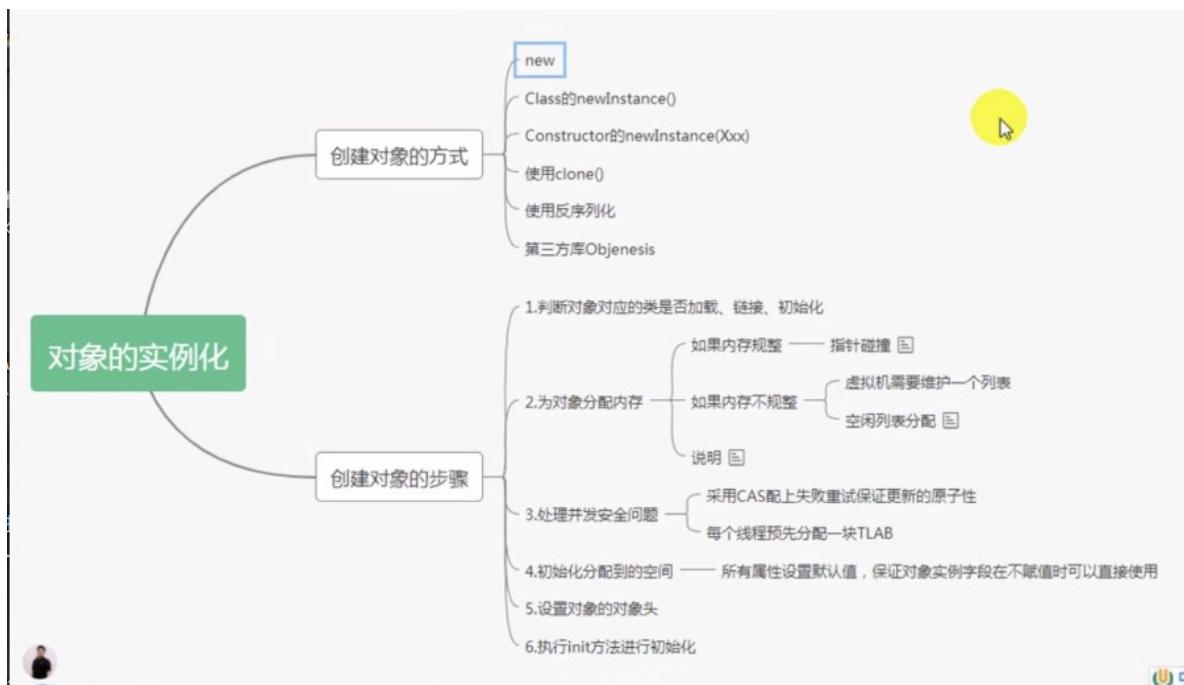
### 方法区的垃圾回收

## 方法区的垃圾收集主要回收两部分内容:常量池中废弃的常量和不再使用的类型

- 方法区内常量池之中主要存在的两大类常量:字面量和符号引用.字面量比较接近Java语言层次的常量概念,如文本字符串,被声明为final的常量值等.而符号引用则属于编译原理方面的概念,包括下面三类常量:
  - 1.类和接口的全限定名
  - 2.字段的名称和描述符
  - 3.方法的名称和描述符
- HotSpot虚拟机对常量池的回收策略是很明确的,==只要常量池中的常量没有被任何地方引用,就可以被回收.
- 回收废弃常量与回收Java堆中的对象非常类似

判断一个常量是否废弃相对简单,而要判定一个类型是否属于"不再被使用的类"的条件就比较苛刻了.

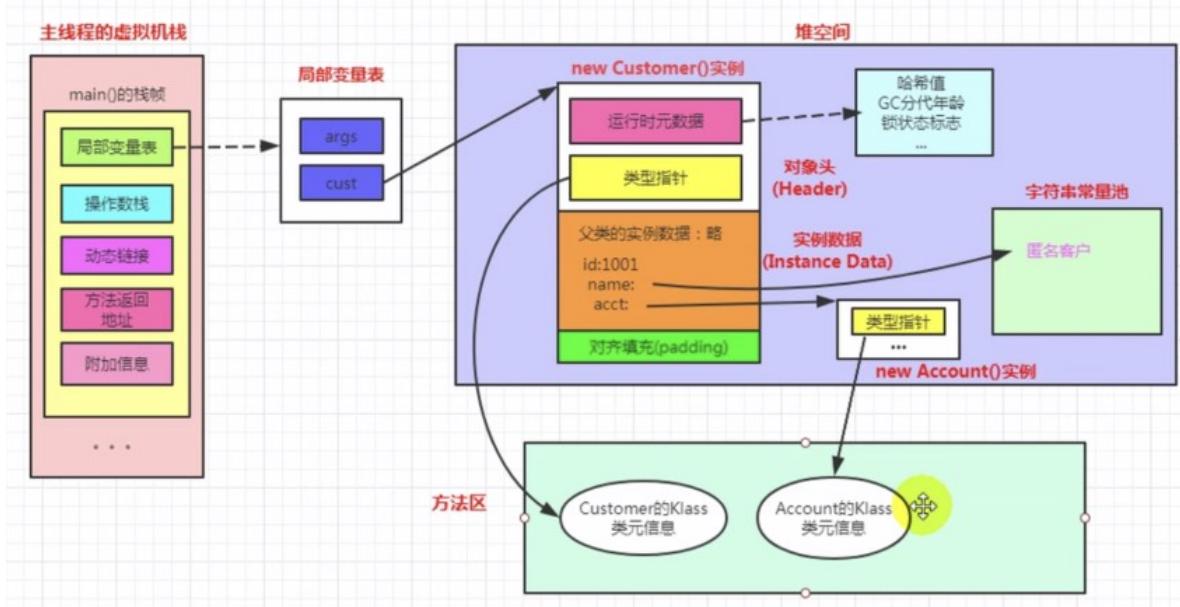
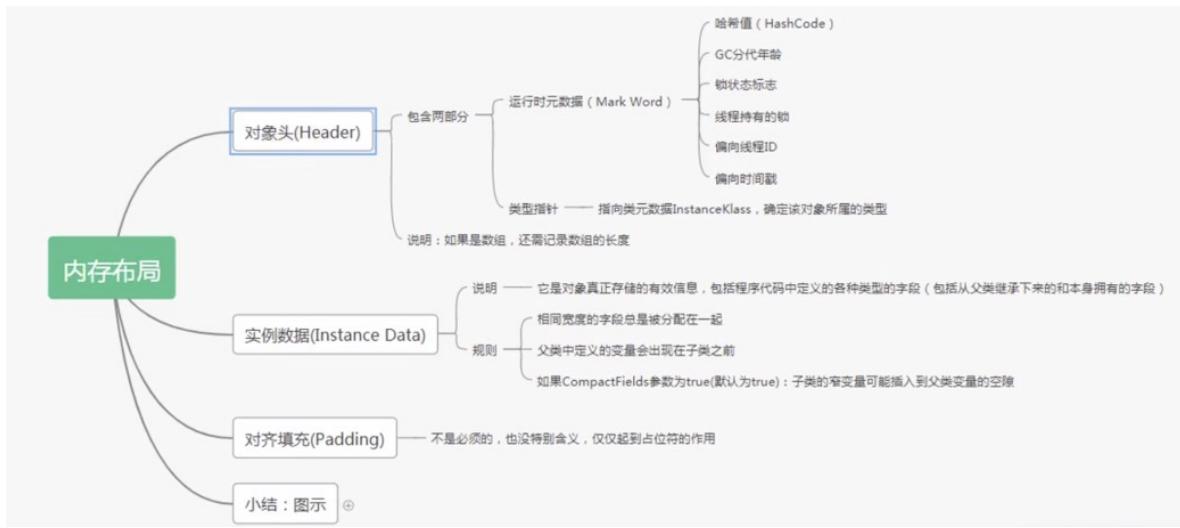
## 对象的实例化



1. 虚拟机遇到一条new指令,首先去检查这个指令的参数是否能在Metaspace的常量池中定位到一个类的符号引用,并且检查这个符号引用代表的类是否已经被加载,解析和初始化.(即判断类元信息是否存在),如果没有,那么在双亲委派模式下,使用当前类加载器以ClassLoader+包名+类名为key进行查找对应的.class文件.如果没有找到文件,则抛出ClassNotFoundException异常,如果找到,则进行类加载,并生成对应的class类对象.

2.首先计算对象占用空间大小,接着在堆中划分一块内存给新对象.如果实例成员变量是引用变量,仅分配引用变量空间即可,即4个字节大小.

## 对象内存布局



## 直接内存

不是虚拟机运行时数据区的一部分,是Java堆外,直接向系统申请的内存空间.访问直接内存的速度会优于Java堆.

元空间放在直接内存上.

直接内存也可能产生 **outofMemory**

## 四.执行引擎

### 1.执行引擎概述

执行引擎是Java虚拟机核心的组成部分之一.

JVM的主要任务就是负责**装载字节码到其内部**,执行引擎的任务就是**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.简单来说,JVM的执行引擎充当了将高级语言翻译为机器语言的译者.

### 2.Java代码的编译和执行过程

**解释器**:当Java虚拟机启动时会根据预定义的规范**对字节码采用逐行解释的方式执行**,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.

**JIT(Just In Time Compiler)编译器**:就是虚拟机将源码直接编译成和本地机器平台相关的机器语言.

为什么说Java是半解释型半编译型语言?

JVM在执行Java代码的时候,通常都会将解释执行与编译执行二者结合起来进行.

## JIT编译器

当程序启动后,解释器可以马上发挥作用,省去编译的时间,立即执行.

编译器想要发挥作用,把代码编译成本地代码,需要一定的执行时间.但编译为本地代码后,执行效率高.

- Java语言的“编译期”其实是一段“不确定”的操作过程,因为它可能是指一个**前端编译器**(如javac 其实叫“编译器的前端”更准确一些)把.java文件转变成.class文件的过程.
- 也可能是指虚拟机的**后端运行期编译器**(JIT编译器),把字节码转变为机器码的过程
- 还可能是使用**静态提前编译器**(AOT编译器)直接把.java文件编译成本地机器代码的过程.

前端编译器: Sun的javac,eclipse的ECJ

JIT编译器: HotSpot VM的C1,C2编译器

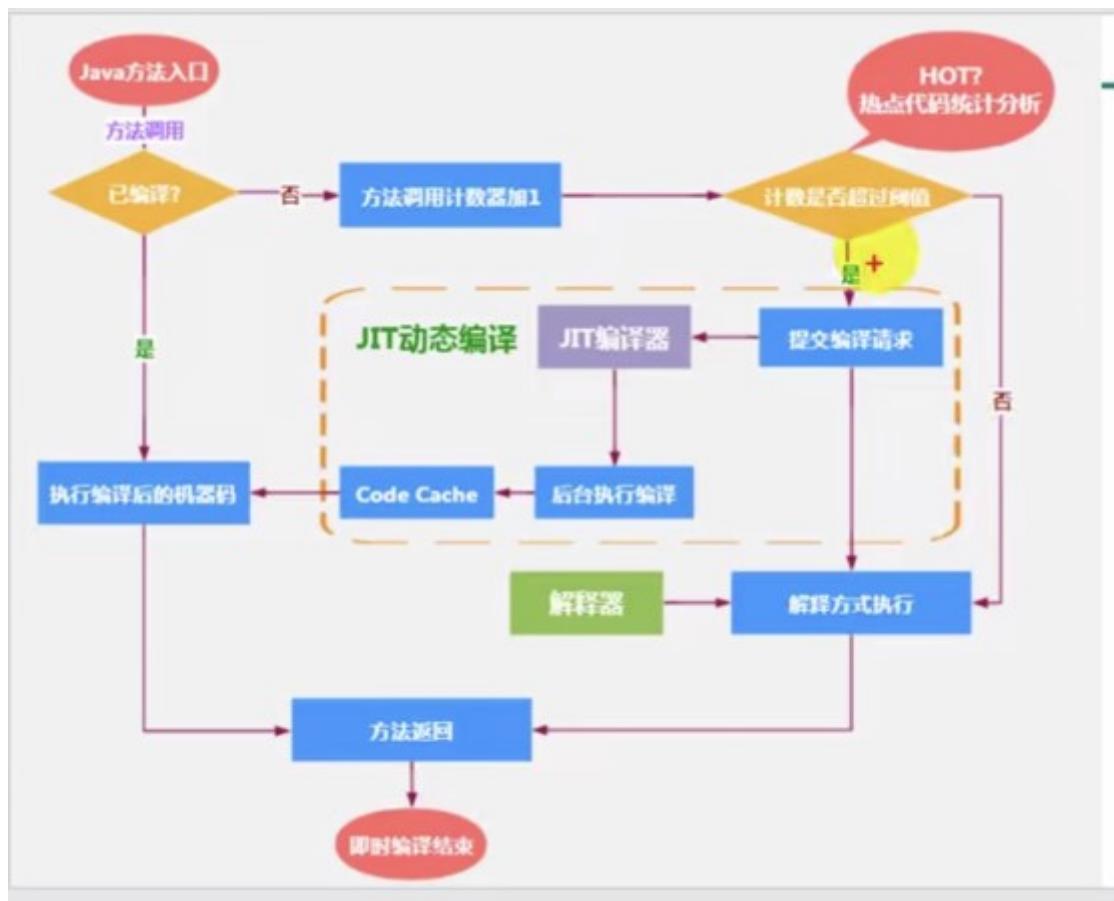
AOT编译器:GCJ,Excelsior JET

**C1和C2编译器不同的优化策略：**

- 在不同的编译器上有不同的优化策略, C1编译器上主要有方法内联, 去虚拟化、冗余消除。
  - 方法内联: 将引用的函数代码编译到引用点处, 这样可以减少栈帧的生成, 减少参数传递以及跳转过程
  - 去虚拟化: 对唯一的实现类进行内联
  - 冗余消除: 在运行期间把一些不会执行的代码折叠掉
- C2的优化主要是在全局层面, 逃逸分析是优化的基础。基于逃逸分析在C2上有如下几种优化:
  - 标量替换: 用标量值代替聚合对象的属性值
  - 栈上分配: 对于未逃逸的对象分配对象在栈而不是堆
  - 同步消除: 清除同步操作, 通常指synchronized

## 热点代码探测技术

- 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, 因此也被称之为栈上替换, 或简称为OSR (On Stack Replacement) 编译。
- 一个方法究竟要被调用多少次, 或者一个循环体究竟需要执行多少次循环才可以达到这个标准? 必然需要一个明确的阈值, JIT编译器才会将这些“热点代码”编译为本地机器指令执行。这里主要依靠**热点探测功能**。
- 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**。
- 采用基于计数器的热点探测, HotSpot VM将会为每一个方法都建立2个不同类型的计数器, 分别为方法调用计数器(Invocation Counter)和回边计数器(Back Edge Counter)。
  - 方法调用计数器用于统计方法的调用次数
  - 回边计数器则用于统计循环体执行的循环次数



### 热度衰减

- 如果不做任何设置，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，即一段时间之内方法被调用的次数。当超过一定的时间限度，如果方法的调用次数仍然不足以让它提交给即时编译器编译，那这个方法的调用计数器就会被减少一半，这个过程称为方法调用计数器热度的衰减 (Counter Decay)，而这段时间就称为此方法统计的半衰周期 (Counter Half Life Time)。
- 进行热度衰减的动作是在虚拟机进行垃圾收集时顺便进行的，可以使用虚拟机参数 `-XX:-UseCounterDecay` 来关闭热度衰减，让方法计数器统计方法调用的绝对次数，这样，只要系统运行时间足够长，绝大部分方法都会被编译成本地代码。
- 另外，可以使用 `-XX:CounterHalfLifeTime` 参数设置半衰周期的时间，单位是秒。

### 补充

自JDK10起,HotSpot加入全新的即时编译器:Graal编译器,编译效果短短几年时间就追平了C2编译器,未来可期.

#### AOT编译器:

即时编译指的是在程序运行过程中,将字节码转换为可在硬件上直接运行的机器码.

## StringTable

jdk9之后String使用byte[]加上编码标记取代了之前的char[]

String:代表不可变的字符序列.简称:不可变性

字符串常量池中是不会存储相同内容的字符串的

String的String Pool是一个固定大小的HashTable.

## String的内存分配

在Java7中对字符串常量池的逻辑做了很大的改变,即将字符串常量池的位置调整到java堆内

## 字符串拼接

1. 常量与常量拼接结果在常量池中, 原理是编译期优化
2. 常量池中不会存在相同内容的常量
3. 只要其中有一个是变量 (如果拼接符号的前后出现了变量, 则相当于在堆空间中new String()) , 结果就在堆中。变量拼接的原理是StringBuilder。先new StringBuilder(), 再调用append() 拼接, 最后toString(), 就类似于new String()。
4. 如果拼接的结果调用intern()方法, 则主动将常量池中还没有的字符串对象放入池中 (有就直接返回地址) , 并返回此对象地址。

## intern()的使用

如果不是用双引号声明的String对象, 可以使用String提供的intern方法: intern方法会从字符串常量池中查询当前字符是否存在, 若不存在就会将当前字符串放入常量池中。

比如: String s = new String("Hello").intern();

Intern String就是确保字符串在内存中只有一份拷贝, 这样可以节约内存空间, 加快字符串操作任务的执行速度。

总结String的intern()的使用:

- jdk1.6中,将这个字符串对象尝试放入常量池
  - 如果有,则不会放入,返回已有的常量池中的对象的地址
  - 如果没有,会把此对象复制一份,放入常量池,并返回常量池对象地址
- jdk1.7起,将这个字符串对象尝试放入常量池
  - 如果有,返回已有的常量池的对象地址
  - 如果没有,则会把对象的引用地址复制一份,放入常量池,返回常量池中的引用地址.

```
1  @Test
2  public void test1(){
3      //String s = new String("ab"); 这个会把ab放入常量池
4      String s = new String("a") + new String("b");//在常量池中没有
5      String s2 = s.intern();//现在放入常量池
6      System.out.println(s2 == "ab");//jdk1.6:true jdk1.7:true
7      System.out.println(s == "ab");//jdk1.6:false jdk1.7:true
8 }
```

## G1的String去重操作

## 垃圾回收窗

### 1.垃圾回收概述

垃圾是指在运行程序中没有任何指针指向的对象.

### 2.垃圾回收相关算法

①垃圾标记阶段算法--引用计数算法

引用计数法(Reference Counting)比较简单,对每个对象保存一个整型的引用计数器属性,用于记录对象被引用的情况.

对于一个对象A,只要有任何一个对象引用了A,则A的引用计数器+1,当引用失效时,引用计数器-1,只要对象A的引用计数器为0,即表示对象A不可能再被使用,可进行回收.

无法处理循环引用,未被Java使用

## ②垃圾标记阶段算法--可达性分析(或根搜索算法,追踪性垃圾集)

可以有效解决循环引用的问题,防止内存泄漏的发生.

所谓"GC Roots"根集合就是一组必须活跃的引用.

- 可达性分析算法是以根对象集合(GC Roots)为起始点,按照从上至下的方式搜索被根对象集合所连接的目标对象是否可达.
- 使用可达性分析算法后,内存中的存活对象都会被根对象集合直接或间接连接着,搜索所走过的路径称为引用链(Reference Chain)
- 如果目标对象没有任何引用链相连,则是不可达的,就意味着该对象已经死亡,可以标记为垃圾对象.
- 在可达性分析算法中,只有能够被根对象集合直接或间接连接的对象才是存活对象.

### GC Roots包括的元素

- 虚拟机栈中引用的对象,比如各个线程调用的方法中使用的参数,局部变量
- 本地方法栈中JNI(通常说的本地方法)引用的对象
- 方法区中常量引用的对象,如:字符串常量池(String Table)里的引用

如果使用可达性分析算法来判断内存是否可回收,那么分析工作必须在一个保证一致性的快照中进行.这点也是导致GC进行时必须"Stop The World"的一个重要原因.

## ③对象的finalization机制

Java语言提供了对象终止(finalization)机制来允许开发人员提供对象被销毁之前的自定义处理逻辑

当垃圾回收器发现没有引用指向一个对象,即:垃圾回收对象之前,总会先调用这个对象的finalize()方法.

finalize()方法允许在子类中被重写,用于在对象被回收时进行资源释放,通常在这个方法中进行一些资源释放和清理工作,比如关闭文件,套接字和数据库连接等.

- 永远不要主动调用某个对象的finalize()方法,应该交给垃圾回收机制调用.理由包括下面三点:
  - 在finalize()时可能会导致对象复活。|I
  - finalize()方法的执行时间是没有保障的,它完全由GC线程决定,极端情况下,若不发生GC,则finalize()方法将没有执行机会。
  - 一个糟糕的finalize()会严重影响GC的性能。
- 从功能上来说, finalize()方法与C++中的析构函数比较相似,但是Java采用的是基于垃圾回收器的自动内存管理机制,所以finalize()方法在本质上不同于C++中的析构函数。
- 由于finalize()方法的存在,虚拟机中的对象一般处于三种可能的状态。

- 如果从所有的根节点都无法访问到某个对象，说明对象已经不再使用了。一般来说，此对象需要被回收。但事实上，也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段。**一个无法触及的对象有可能在某一个条件下“复活”自己**，如果这样，那么对它的回收就是不合理的，为此，定义虚拟机中的对象可能的三种状态。如下：
- **可触及的**: 从根节点开始，可以到达这个对象。
- **可复活的**: 对象的所有引用都被释放，但是对象有可能在**finalize()**中复活。
- **不可触及的**: 对象的**finalize()**被调用，并且没有复活，那么就会进入不可触及状态。不可触及的对象不可能被复活，因为**finalize()**只会被调用一次。
- 以上3种状态中，是由于**finalize()**方法的存在，进行的区分。只有在对象不可触及时才可以被回收。

 **具体过程** 

判定一个对象objA是否可回收，至少要经历两次标记过程：

- 如果对象objA到GC Roots没有引用链，则进行第一次标记。
- 进行筛选，判断此对象是否有必要执行**finalize()**方法
  - 如果对象objA没有重写**finalize()**方法，或者**finalize()**方法已经被虚拟机调用过，则虚拟机视为“没有必要执行”，objA被判定为不可触及的。
  - 如果对象objA重写了**finalize()**方法，且还未执行过，那么objA会被插入到F-Queue队列中，由一个虚拟机自动创建的、低优先级的Finalizer线程触发其**finalize()**方法执行。
  - finalize()方法是对象逃脱死亡的最后机会**，稍后GC会对F-Queue队列中的对象进行第二次标记。如果objA在**finalize()**方法中与引用链上的任何一个对象建立了联系，那么在第二次标记时，objA会被移出“即将回收”集合。之后，对象会再次出现没有引用存在的情况。在这个情况下，**finalize**方法不会被再次调用，对象会直接变成不可触及的状态，也就是说，一个对象的**finalize**方法只会被调用一次。

#### ④MAT与jprofiler

**MAT**是Memory Analyzer的简称，它是一款功能强大的Java堆内存分析器，用于查找内存泄漏以及查看内存消耗情况。

**jprofiler**的GC Root溯源

### 3. 垃圾清除算法

#### ①. 标记-清除算法(Mark-Sweep)

当堆中的有效空间被耗尽的时候，就会停止整个程序(**Stop The World**)，然后进行两项工作，第一项是标记，第二项是清除。

- 标记**: Collector从引用根节点开始变量，**标记所有被引用的对象**。一般是在对象的Header中记录为可达对象
- 清除**: Collector对堆内存从头到尾进行线性遍历，如果发现某个对象在Header中没有标记为可达对象，则将其回收。

**缺点**: 效率不高，在进行GC的时候，需要停止整个应用程序，产生内存碎片

#### ② 复制算法(Copying)

将活着的内存空间分为两块，每次只使用其中一块，在垃圾回收时将正在使用的内存中的存活对象复制到未被使用的内存块中，之后清除正在使用的内存块中的所有对象，交换两个内存的角色，最后完成垃圾回收。

**优点:**没有标记和清除的过程,实现简单,运行高效;不会出现"碎片"问题

**缺点:**需要两倍的内存空间,对于G1这种拆分成为大量region的GC,复制而不是移动,意味着GC需要维护region之间对象引用关系,不管是内存占用或者时间开销也不小.

### ③ 标记-压缩算法(Mark-Compact)

第一阶段和标记-清除算法一样,从根节点开始标记所有被引用对象,第二阶段将所有的存活对象压缩到内存的一端,按顺序排放,之后,清理边界外所有的空间.

标记-压缩算法的最终效果等同于标记-清除算法执行后,再进行依次内存碎片整理,因此,也可以把她称为**标记-清除-压缩算法**.

## 分代收集算法

目前几乎所有的GC都是采用分代收集(Generational Collecting)算法执行垃圾回收的.

- 年轻代(Young Gen)

年轻代的特点:区域相对老年代较小,对象生命周期短,存活率低,回收频繁

这种情况复制算法的收集整理,速度是最快的.

- 老年代(Tenure Gen)

老年代的特点:区域大,对象生命周期长,存活率高,回收不及年轻代频繁.

这种情况存在大量存活率高的对象,复制算法明显变得不合适.一般是由标记-清除或者是标记-压缩的混合实现.

## 增量收集算法

如果一次性将所有的垃圾进行清理,需要造成系统长时间的停顿,那么就可以让垃圾收集线程和应用程序线程交替执行,每次,**垃圾收集线程只收集一小片区域的内存空间,接着切换到应用程序线程.依次反复,直到垃圾收集完成**

总的来说,增量收集算法的基础仍是传统的标记-清除和复制算法.增量收集算法通过对线程间冲突的妥善处理,允许垃圾收集线程以分阶段的方式完成标记,清理或复制工作.

**缺点:**频繁的线程切换会降低系统的吞吐量

## 分区算法

将一块大的内存区域划分成多个小块,根据目标的停顿时间,每次合理地回收若干个小区间,而不是整个堆空间,从而减少一次GC所产生的停顿.

分代算法将按照对象的生命周期长短划分成两个部分,分区算法将整个堆空间划分成连续的不同小区间region.

每一个小区间都独立使用,独立回收.

## 垃圾回收相关概念

### ① System.gc()

System.gc()会显式触发Full GC.仅仅是提醒JVM进行一次GC.不一定会成功.

### ② 内存溢出和内存泄漏

**内存溢出:**javadoc中对**OutOfMemoryError** 的解释是:**没有空闲内存,并且垃圾收集器也无法提供更多内存**

**内存泄漏(Memory Leak):****只有对象不会再被程序用到了,但是GC又不能回收它们的情况,才叫内存泄漏**

## 举例:

- 1.单例模式:单例的生命周期和应用程序是一样长的,所以单例程序中,如果持有对外部对象的引用的话,那么这个外部对象是不能被回收的,则会导致内存泄漏的产生.
- 2.一些提供close的资源未关闭导致的内存泄漏,数据库连接,网络连接(socket)和io连接必须手动close,否则是不能被回收的.

## ③ Stop The World

简称STW,指的是GC事件发生过程中,会产生应用程序的停顿.**停顿产生时整个应用程序都会被暂停,没有任何响应**,有点像卡死的感觉.

STW事件和采用哪款GC无关,所有的GC都有这个事件.

## ④ 垃圾回收的并行与并发

并发与并行,在谈论垃圾回收器的上下文语境中,它们可以解释如下:

- **并行(Parallel)**:指多条垃圾收集线程并行工作,但此时用户线程仍处于等待状态.如:ParNew,Parallel Scavenge,Parallel Old;
- **串行**:单线程执行
- **并发**:指用户线程与垃圾线程同时执行(但不一定是并行的,可能会交替执行),垃圾回收线程在执行时不会停顿用户程序的运行.如:CMS,G1.

## 强软弱虚引用

1 | java.lang.ref.Reference

- **强引用(Strong Reference)**:最传统的"引用"的定义,是指在程序代码之中普遍存在的引用赋值,即类似"Object o = 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,GC就永远不会回收掉被引用的对象**.虚拟机宁愿抛出OOM异常,也不会回收强引用所指向的对象.强引用可能导致内存泄漏.
- **软引用(Soft Reference)**:在系统将要发生内存溢出之前,将会把这些对象列入回收范围之中进行第二次回收,如果这次回收后还没有足够的内存,才会抛出内存溢出异常.软引用通常用来实现内存敏感的缓存.比如**高速缓存**
- **弱引用(Weak Reference)**:**被弱引用关联的对象只能生存到下一次垃圾收集之前**.当GC工作时,无论内存空间是否足够,都会回收掉被弱引用关联的对象.由于垃圾回收线程的优先级较低,因此,并不一定能很快发现持有弱引用的对象.在这种情况下,弱引用对象可以存在较长的时间.软引用和弱引用都非常适合来保存那些可有可无的缓存数据.
- **虚引用(Phantom Reference)--对象回收跟踪**:一个对象是否有虚引用的存在,完全不会对其生存时间构成影响,也无法通过虚引用获得一个对象的实例.为一个对象设置虚引用关联的唯一目的就是**能在这个对象被收集器回收时收到一个系统通知**.

## 垃圾回收器

### 评估GC的性能指标

- **吞吐量**:运行用户代码的时间占总运行时间的比例
- **暂停时间**:执行垃圾收集时,程序的工作线程被暂停的时间
- **内存占用**:Java堆区所占的内存大小

**在最大吞吐量优先的情况下,降低停顿时间.**

## 垃圾回收器



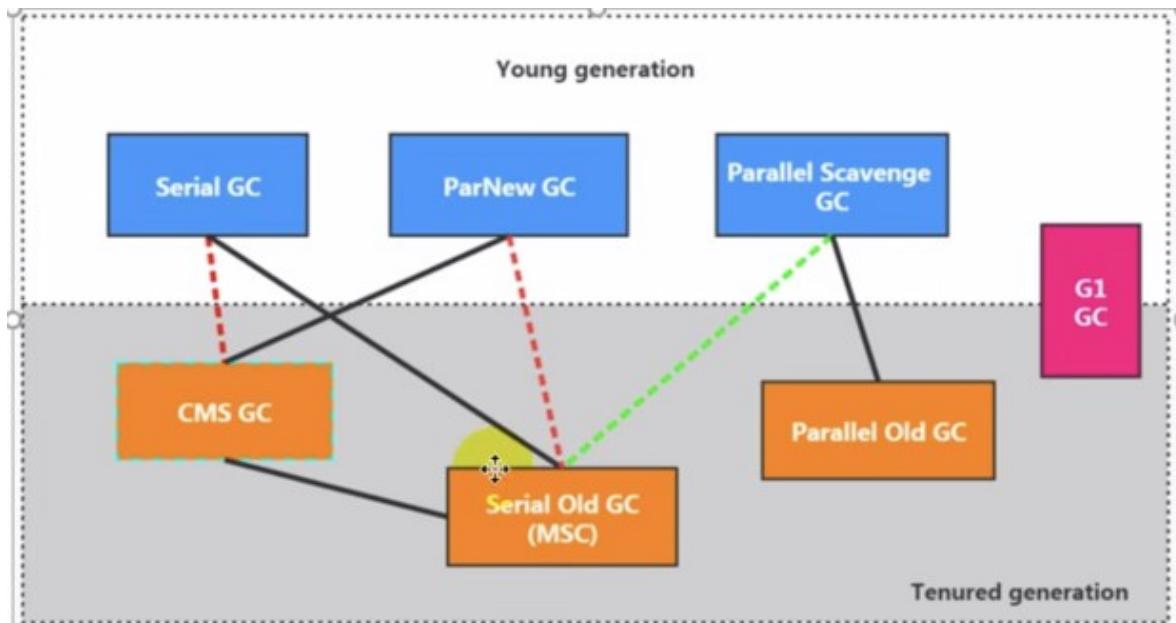
有了虚拟机，就一定需要收集垃圾的机制，这就是Garbage Collection，对应的产品我们称为Garbage Collector。

- 1999年随JDK1.3.1一起来的是串行方式的Serial GC，它是第一款GC。ParNew垃圾收集器是Serial收集器的多线程版本
- 2002年2月26日，Parallel GC 和Concurrent Mark Sweep GC跟随JDK1.4.2一起发布
- Parallel GC在JDK6之后成为HotSpot默认GC。
- 2012年，在JDK1.7u4版本中，G1可用。
- 2017年，JDK9中G1变成默认的垃圾收集器，以替代CMS。
- 2018年3月，JDK10中G1 垃圾回收器的并行完整垃圾回收，实现并行性来改善最坏情况下的延迟。
- 2018年9月，JDK11发布。引入Epsilon 垃圾回收器，又被称为"No-Op（无操作）"回收器。同时，引入ZGC：可伸缩的低延迟垃圾回收器(Experimental)。
- 2019年3月，JDK12发布。增强G1，自动返回未用堆内存给操作系统。同时，引入Shenandoah GC：低停顿时间的GC(Experimental)。
- 2019年9月，JDK13发布。增强ZGC，自动返回未用堆内存给操作系统。
- 2020年3月，JDK14发布。删除CMS垃圾回收器。扩展ZGC在macOS和Windows上的应用

让天下没有难学的技术

## 7款经典的垃圾回收器

- 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

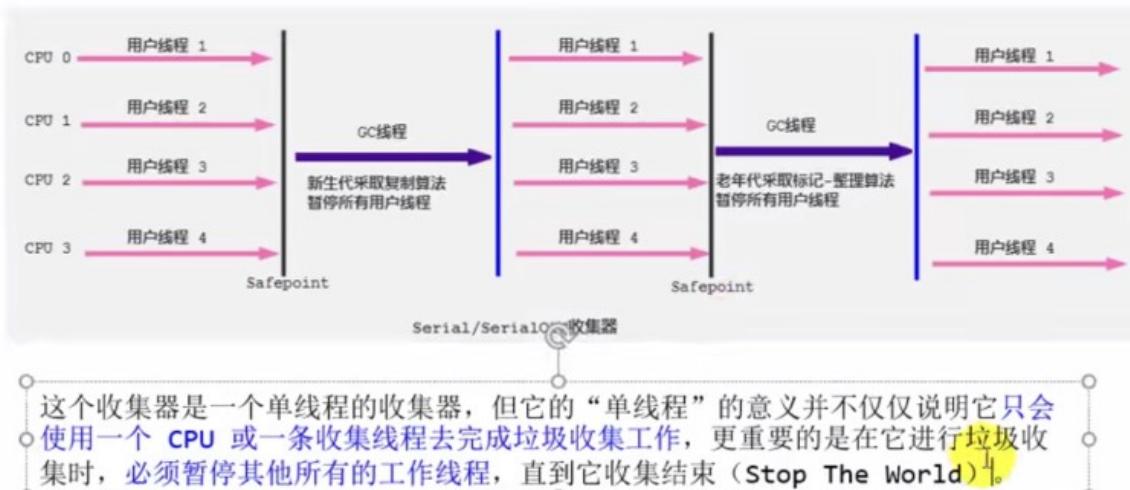


- 1 -XX:+PrintCommandLineFlags 查看命令行相关参数(包含使用的垃圾收集器)
- 2 使用命令行指令：jinfo -flag 相关垃圾回收器参数 进程id

## Serial-串行回收

Serial收集器采用复制算法,串行回收和"Stop The World"机制的方式执行内存回收

除了年轻代之外,Serial收集器还提供用于执行老年代垃圾收集的Serial Old收集器,Serial Old收集器同样采用了串行回收和"Stop The World"机制,只不过内存回收算法使用的是标记-压缩算法



**优点:**简单而高效.

使用 -XX:+UseSerialGC 参数可以指定年轻代和老年代都使用串行收集器.

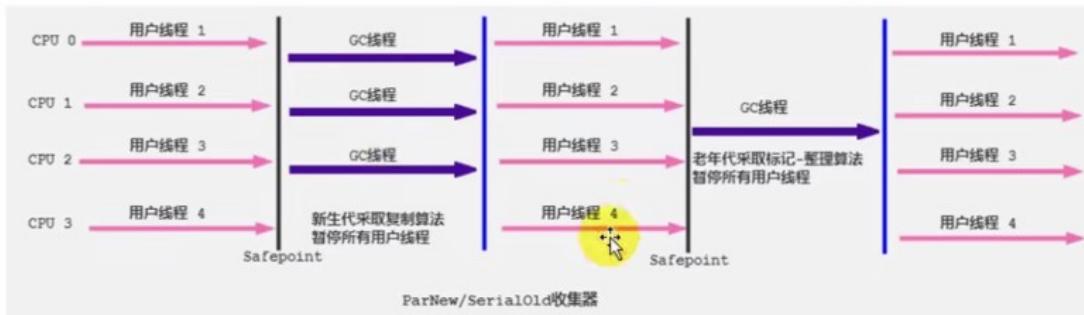
这种垃圾回收器限定在单核CPU中,对于交互较强的程序现在已经不使用了.

### ParNew-并行回收

如果说Serial GC是年轻代的单线程收集器,那么ParNew收集器则是Serial收集器的**多线程版本**.

Par是parallel的缩写,New:只能处理新生代

ParNew除了采用**并行回收**的方式执行内存回收外,与Serial几乎没有区别.ParNew收集器在年轻代同样采用**复制算法**,**Stop-The-World**机制.



- 对于新生代，回收次数频繁，使用并行方式高效。
- 对于老年代，回收次数少，使用串行方式节省资源。（CPU并行需要切换线程，串行可以省去切换线程的资源）

除了Serial外,目前只有ParNew能与CMS收集器配合

使用 -XX:+UseParNewGC 参数手动指定使用ParNew

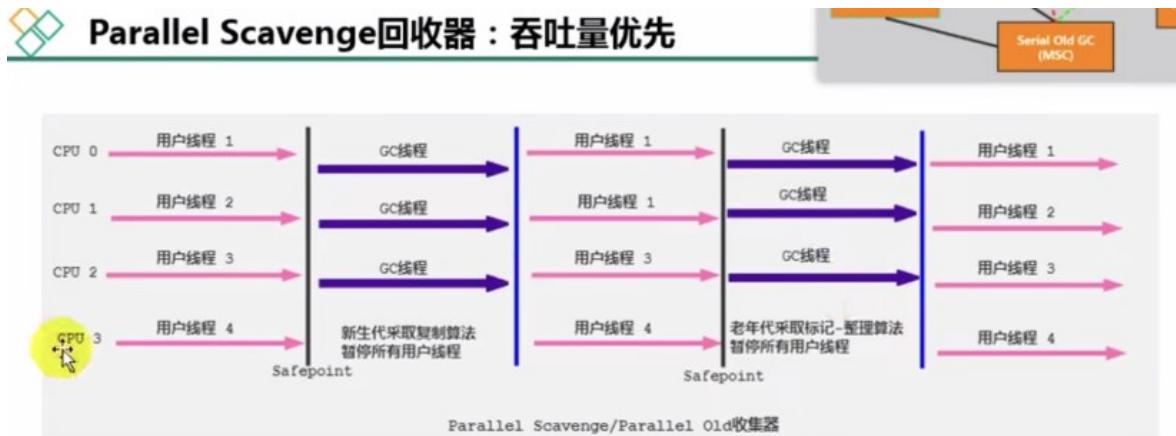
### Parallel Scavenge-吞吐量优先

Parallel Scavenge同样采用了**复制算法并行回收**和**Stop-The-World**机制.

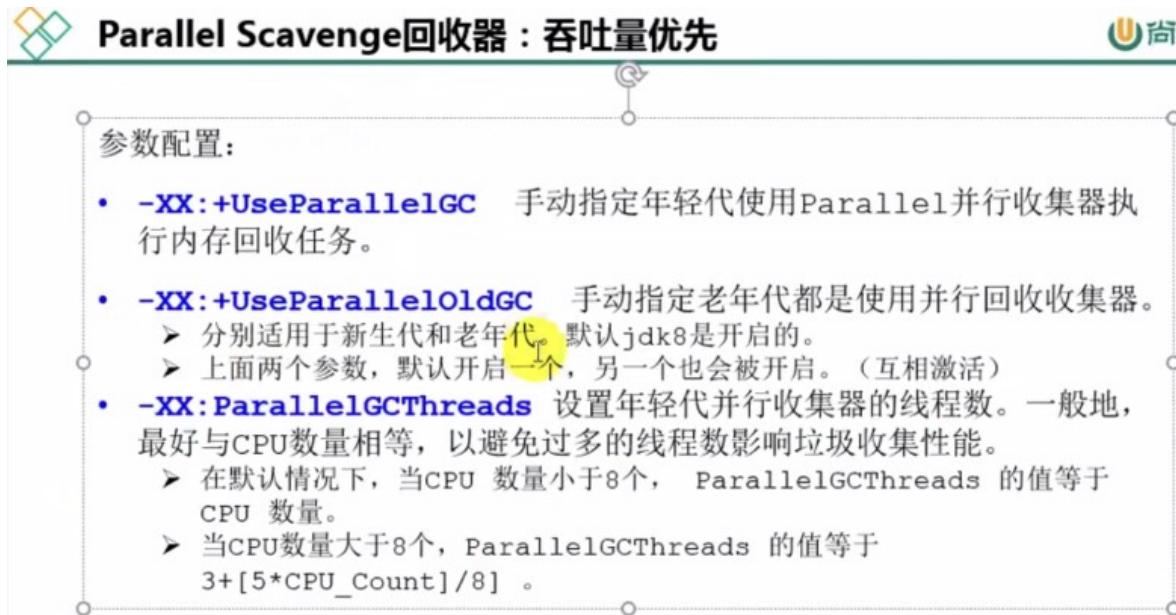
- 与ParNew不同的是,Parallel Scavenge目标是达到一个可控制的**吞吐量**,因此也被称为**吞吐量优先**的垃圾回收器.
- 自适应调节策略也是Parallel Scavenge与ParNew一个重要区别.

高吞吐量则可以高效的利用CPU时间,尽快完成程序的运算任务,主要适合在后台运算而不需要太多交互的任务,例如:批量处理,订单处理,科学计算的引用程序.

Parallel 收集器在jdk1.6提供用于执行老年代垃圾收集的Parallel Old收集器,用来替代老年代的Serial Old收集器.



在jdk8中,是默认垃圾收集器.



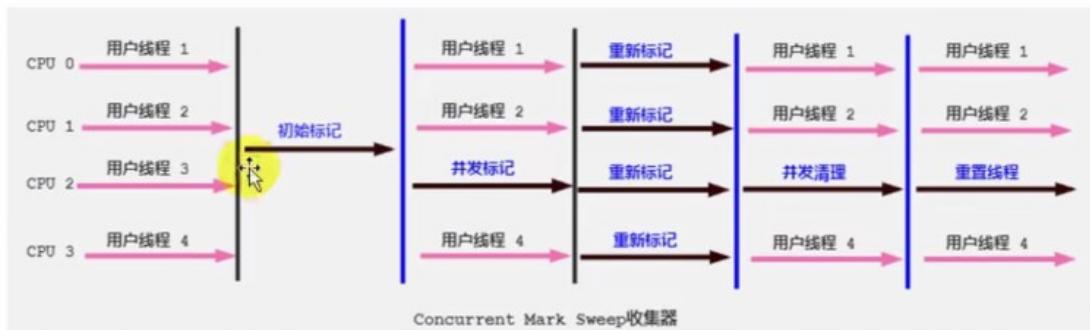
### CMS回收器-低延迟

在jdk1.5时期,HotSpot推出了一款在**强交互应用**中几乎可认为有划时代意义的垃圾收集器--CMS(Concurrent-Mark-Sweep)收集器,这款收集器是HotSpot虚拟机第一款真正意义上的**并发收集器**,它第一次实现了让垃圾收集线程和用户线程同时工作,

CMS收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间,停顿时间越短(低延迟)就越适合与用户交互的程序,良好的响应速度能提升用户体验.

CMS也会**Stop-The-World**.

不幸的是,CMS作为老年代的收集器,却无法与JDK1.4中已经存在的新生代收集器Parallel Scavenge配合工作,所以在jdk1.5中使用CMS来收集老年代的时候,新生代只能选择ParNew或Serial.



CMS整个过程比之前的收集器要复杂，整个过程分为4个主要阶段，即初始标记阶段、并发标记阶段、重新标记阶段和并发清除阶段。

- 初始标记（Initial-Mark）阶段：在这个阶段中，程序中所有的工作线程都将会因为“Stop-the-World”机制而出现短暂的暂停，这个阶段的主要任务仅仅只是标记出GC Roots能直接关联到的对象。一旦标记完成之后就会恢复之前被暂停的所有应用线程。由于直接关联对象比较小，所以这里的速度非常快。
- 并发标记（Concurrent-Mark）阶段：从GC Roots的直接关联对象开始遍历整个对象图的过程，这个过程耗时较长但是不需要停顿用户线程，可以与垃圾收集线程一起并发运行。
- 重新标记（Remark）阶段：由于在并发标记阶段中，程序的工作线程会和垃圾收集线程同时运行或者交叉运行，因此为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间通常会比初始标记阶段稍长一些，但也远比并发标记阶段的时间短。
- 并发清除（Concurrent-Sweep）阶段：此阶段清理删除掉标记阶段判断的已经死亡的对象，释放内存空间。由于不需要移动存活对象，所以这个阶段也是可以与用户线程同时并发的。

尽管CMS收集器采用的是并发回收（非独占式），但是在其初始化标记和再次标记这两个阶段中仍然需要执行“**Stop-the-World**”机制暂停程序中的工作线程，不过暂停时间并不会太长，因此可以说明目前所有的垃圾收集器都做不到完全不需要“**Stop-the-World**”，只是尽可能地缩短暂停时间。

由于最耗费时间的并发标记与并发清除阶段都不需要暂停工作，所以整体的回收是低停顿的。

另外，由于在垃圾收集阶段用户线程没有中断，所以在CMS回收过程中，还应该确保应用程序用户线程有足够的内存可用。因此，CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，而是当堆内存使用率达到某一阈值时，便开始进行回收，以确保应用程序在CMS工作过程中依然有足够的空间支持应用程序运行。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次“**Concurrent Mode Failure**”失败，这时虚拟机将启动后备预案：临时启用 **Serial Old** 收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。

由于CMS采用的是标记-清除算法，所以会产生一些内存碎片。那么CMS在为新对象分配内存的空间时，将无法使用指针碰撞(Bump the Pointer)，而只能选择空闲列表(Free List)执行内存分配。

**优点：**并发收集，低延迟

**缺点：**

- CMS的弊端：

- 1) 会产生内存碎片，导致并发清除后，用户线程可用的空间不足。在无法分配大对象的情况下，不得不提前触发Full GC。
- 2) CMS收集器对CPU资源非常敏感。在并发阶段，它虽然不会导致用户停顿，但是会因为占用了一部分线程而导致应用程序变慢，总吞吐量会降低。
- 3) CMS收集器无法处理浮动垃圾。可能出现“Concurrent Mode Failure”失败而导致另一次 Full GC 的产生。在并发标记阶段由于程序的工作线程和垃圾收集线程是同时运行或者交叉运行的，那么在并发标记阶段如果产生新的垃圾对象，CMS将无法对这些垃圾对象进行标记，最终会导致这些新产生的垃圾对象没有被及时回收，从而只能在下一次执行GC时释放这些之前未被回收的内存空间。

## CMS收集器可以设置的参数

- **-XX:+UseConcMarkSweepGC** 手动指定使用CMS 收集器执行内存回收任务。
  - 开启该参数后会自动将-XX:+UseParNewGC打开。即：ParNew(Young区用)+CMS(Old区用)+Serial Old的组合。
- **-XX:CMSInitiatingOccupancyFraction** 设置堆内存使用率的阈值，一旦达到该阈值，便开始进行回收。
  - JDK5及以前版本的默认值为68，即当老年代的空间使用率达到68%时，会执行一次CMS 回收。JDK6及以上版本默认值为92%
  - 如果内存增长缓慢，则可以设置一个稍大的值，大的阈值可以有效降低CMS的触发频率，减少老年代回收的次数可以较为明显地改善应用程序性能。反之，如果应用程序内存使用率增长很快，则应该降低这个阈值，以避免频繁触发老年代串行收集器。因此通过该选项便可以有效降低Full GC 的执行次数。

- **-XX:+UseCMSCompactAtFullCollection** 用于指定在执行完Full GC后对内存空间进行压缩整理，以此避免内存碎片的产生。不过由于内存压缩整理过程无法并发执行，所带来的问题就是停顿时间变得更长了。
- **-XX:CMSFullGCsBeforeCompaction** 设置在执行多少次Full GC后对内存空间进行压缩整理。
- **-XX:ParallelCMTThreads** 设置CMS的线程数量。
  - CMS 默认启动的线程数是  $(ParallelGCThreads + 3) / 4$ , **ParallelGCThreads** 是年轻代并行收集器的线程数。当CPU 资源比较紧张时，受到CMS收集器线程的影响，应用程序的性能在垃圾回收阶段可能会非常糟糕。

## JDK14删除了CMS

### G1回收器-区域化分代式

官方给G1设定的目标是在**延迟可控的情况下获得尽可能高的吞吐量**,所以才担当起"全功能收集器"的重任与期望.

- 因为G1是一个并行回收器,它把堆内存分割为很多不相关的区域(region)(物理上不连续).使用不同的region来表示Eden,Survivor0区,Survivor1区,老年代等.
- G1有计划地避免在整个Java堆中进行全区域的垃圾收集.G1跟踪各个region里面的垃圾堆积的价值大小(回收所获得的空间大小以及回收所需时间的经验值),在后台维护一个**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**.
- 由于这种方式侧重点在于回收垃圾最大量的区间(region),所以我们给G1一个名字:垃圾优先(**Garbage First**).

### G1回收器是jdk9的默认垃圾处理器

G1将内存分为一个个的region.内存的回收是以region作为基本单位的.**region之间是复制算法**,但整体上实际可看作是**标记-压缩算法**,两种算法都可以避免内存碎片.

**G1回收器的参数设置**

- **-XX: +UseG1GC** 手动指定使用G1收集器执行内存回收任务。
- **-XX:G1HeapRegionSize** 设置每个Region的大小。值是2的幂，范围是1MB到32MB之间，目标是根据最小的Java堆大小划分出约2048个区域。默认是堆内存的1/2000。
- **-XX:MaxGCPauseMillis** 设置期望达到的最大GC停顿时间指标 (JVM会尽力实现，但不保证达到)。默认值是200ms
- **-XX:ParallelGCThread** 设置STW工作线程数的值。最多设置为8
- **-XX:ConcGCThreads** 设置并发标记的线程数。将n设置为并行垃圾回收线程数(ParallelGCThreads)的1/4左右。
- **-XX:InitiatingHeapOccupancyPercent** 设置触发并发GC周期的Java堆占用率阈值。超过此值，就触发GC。默认值是45。

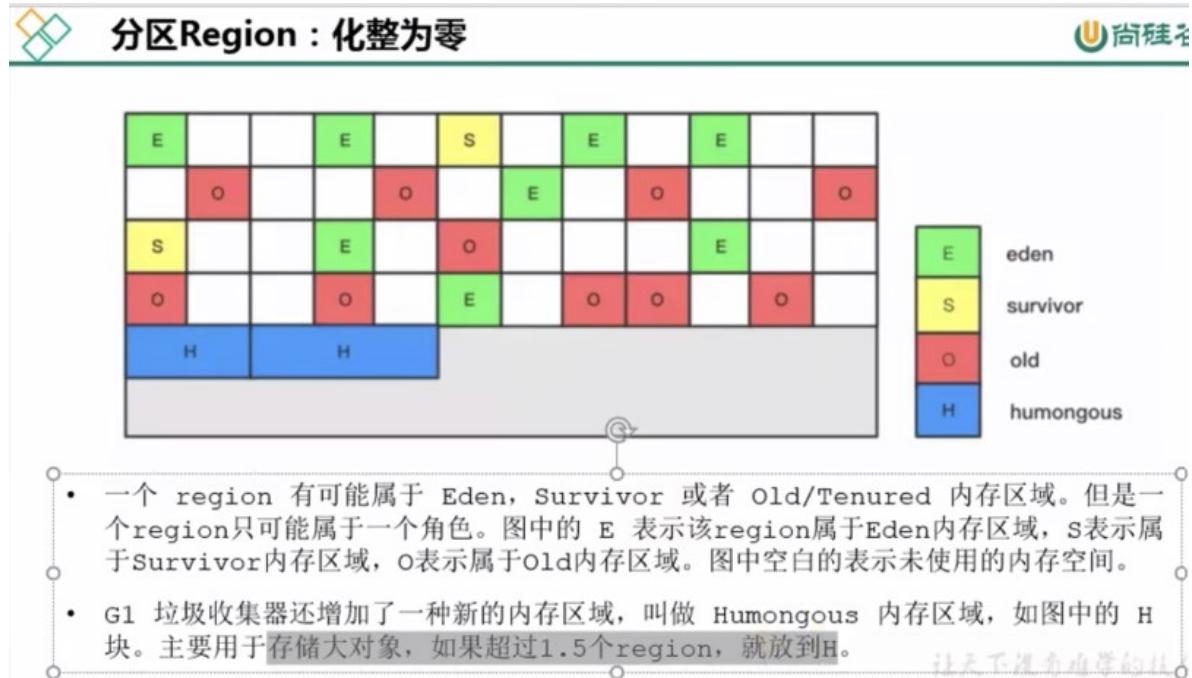
### G1回收器的常见操作步骤

1. 开启G1垃圾收集器

2. 设置堆的最大内存
3. 设置最大的停顿时间

### 分区region-化整为零

使用G1收集器时,它将整个Java堆划分成约2048个大小相同的独立region块,每个region块大小根据堆空间的实际大小而定,整体被控制在1MB到32MB之间,且为2的N次幂,可以通过`-xx:G1HeapRegionSize`设定,所有的region大小相同,且在JVM生命周期内不会被改变.



设置H的原因:

对于堆中的大对象,默认直接会被分配到老年区,但是如果它是一个短期存在的大对象,就会对垃圾收集器造成负面影响.为了解决这个问题,G1划分了一个Humongous区,它用来专门存储大对象.如果一个H区装不下一个大对象,那么G1会寻找连续的H区来存储.为了能找到连续的H区,有时候不得不启动Full GC,G1的大多数行为都把H区作为老年区的一部分来看待.

### G1回收器回收过程

G1的垃圾回收过程主要包括如下三个环节

1. 年轻代(Young GC)
2. 老年代并发标记过程(Concurrent Marking)
3. 混合回收(Mixed GC)
4. (如果需要,单线程,独占式,高强度的Full GC还是继续存在的.它针对GC的评估失败提供了一种失败保护机制,即强力回收)



应用程序分配内存，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。在年轻代回收期，G1 GC暂停所有应用程序线程，启动多线程执行年轻代回收。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。

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。

标记完成马上开始混合回收过程。对于一个混合回收期，G1 GC从老年区间移动存活对象到空闲区间，这些空闲区间也就成为了老年代的一部分。和年轻代不同，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。同时，这个老年代Region是和年轻代一起被回收的。

举个例子：一个Web服务器，Java进程最大堆内存为4G，每分钟响应1500个请求，每45秒钟会新分配大约2G的内存。G1会每45秒钟进行一次年轻代回收，每31个小时整个堆的使用率会达到45%，会开始老年代并发标记过程，标记完成后开始四到五次的混合回收。



- 一个对象被不同区域引用的问题
- 一个Region不可能是孤立的，一个Region中的对象可能被其他任意Region中对象引用，判断对象存活时，是否需要扫描整个Java堆才能保证准确？
- 在其他的分代收集器，也存在这样的问题（而G1更突出）
- 回收新生代也不得不同时扫描老年代？
- 这样的话会降低Minor GC的效率；
  
- 解决方法：
- 无论G1还是其他分代收集器，JVM都是使用Remembered Set来避免全局扫描：
  - 每个Region都有一个对应的Remembered Set；
  - 每次Reference类型数据写操作时，都会产生一个Write Barrier暂时中断操作；
  - 然后检查将要写入的引用指向的对象是否和该Reference类型数据在不同的Region（其他收集器：检查老年代对象是否引用了新生代对象）；
  - 如果不同，通过CardTable把相关引用信息记录到引用指向对象的所在Region对应的Remembered Set中；
  - 当进行垃圾收集时，在GC根节点的枚举范围加入Remembered Set；就可以保证不进行全局扫描，也不会有遗漏。



然后开始如下回收过程：

#### 第一阶段，扫描根。

根是指static变量指向的对象，正在执行的方法调用链条上的局部变量等。根引用连同RSet记录的外部引用作为扫描存活对象的入口。

#### 第二阶段，更新RSet。

处理dirty card queue(见备注)中的card，更新RSet。此阶段完成后，RSet可以准确的反映老年代对所在的内存分段中对象的引用。

#### 第三阶段，处理RSet。

识别被老年代对象指向的Eden中的对象，这些被指向的Eden中的对象被认为是存活的对象。

#### 第四阶段，复制对象。

此阶段，对象树被遍历，Eden区内存段中存活的对象会被复制到Survivor区中空的内存分段，Survivor区内存段中存活的对象如果年龄未达阈值，年龄会加1，达到阈值会被复制到Old区中空的内存分段。如果Survivor空间不够，Eden空间的部分数据会直接晋升到老年代空间。

#### 第五阶段，处理引用。

处理Soft, Weak, Phantom, Final, JNI Weak 等引用。最终Eden空间的数据为空，GC停止工作，而目标内存中的对象都是连续存储的，没有碎片，所以复制过程可以达到内存整理的效果，减少碎片。



## G1回收过程二：并发标记过程



**1. 初始标记阶段：**标记从根节点直接可达的对象。这个阶段是STW的，并且会触发一次年轻代GC。

**2. 根区域扫描 (Root Region Scanning) :** G1 GC扫描Survivor区直接可达的老年代区域对象，并标记被引用的对象。这一过程必须在young GC之前完成。

**3. 并发标记(Concurrent Marking):** 在整个堆中进行并发标记(和应用程序并发执行)，此过程可能被young GC中断。在并发标记阶段，**若发现区域对象中的所有对象都是垃圾，那这个区域会被立即回收**。同时，并发标记过程中，会计算每个区域的对象活性(区域中存活对象的比例)。

**4. 再次标记(Remark):** 由于应用程序持续进行，需要修正上一次的标记结果。是STW的。G1中采用了比CMS更快的初始快照算法:snapshot-at-the-beginning (SATB)。

**5. 独占清理(cleanup, STW)：**计算各个区域的存活对象和GC回收比例，并进行排序，识别可以混合回收的区域。为下阶段做铺垫。是STW的。

➤ 这个阶段并不会实际上去做垃圾的收集

**6. 并发清理阶段：**识别并清理完全空闲的区域。

让天下没有难学的技术



## G1回收过程三：混合回收



- 并发标记结束以后，老年代中百分百为垃圾的内存分段被回收了，部分为垃圾的内存分段被计算了出来。默认情况下，这些老年代的内存分段会分8次（可以通过-XX:G1MixedGCCoountTarget设置）被回收。
- 混合回收的回收集（Collection Set）包括八分之一的老年代内存分段，Eden区内存分段，Survivor区内存分段。混合回收的算法和年轻代回收的算法完全一样，只是回收集多了老年代的内存分段。具体过程请参考上面的年轻代回收过程。
- 由于老年代中的内存分段默认分8次回收，G1会优先回收垃圾多的内存分段。**垃圾占内存分段比例越高的，越会被先回收**。并且有一个阈值会决定内存分段是否被回收，-XX:G1MixedGCLiveThresholdPercent，默认为65%，意思是垃圾占内存分段比例要达到65%才会被回收。如果垃圾占比太低，意味着存活的对象占比高，在复制的时候会花费更多的时间。|
- 混合回收并不一定要进行8次。有一个阈值-XX:G1HeapWastePercent，默认值为10%，意思是允许整个堆内存中有10%的空间被浪费，意味着如果发现可以回收的垃圾占堆内存的比例低于10%，则不再进行混合回收。因为GC会花费很多的时间但是回收到的内存却很少。

让天下没有难学的技术

## 总结



## 7种经典垃圾回收器总结



截止JDK 1.8，一共有7款不同的垃圾收集器。每一款不同的垃圾收集器都有不同的特点，在具体使用的时候，需要根据具体的情况选用不同的垃圾收集器。

垃圾收集器	分类	作用位置	使用算法	特点	适用场景
<b>Serial</b>	串行运行	作用于新生代	复制算法	响应速度优先	适用于单CPU环境下的client模式
<b>ParNew</b>	并行运行	作用于新生代	复制算法	响应速度优先	多CPU环境Server模式下与CMS配合使用
<b>Parallel</b>	并行运行	作用于新生代	复制算法	吞吐量优先	适用于后台运算而不需要太多交互的场景
<b>Serial Old</b>	串行运行	作用于老年代	标记-压缩算法	响应速度优先	适用于单CPU环境下的Client模式
<b>Parallel Old</b>	并行运行	作用于老年代	标记-压缩算法	吞吐量优先	适用于后台运算而不需要太多交互的场景
<b>CMS</b>	并发运行	作用于老年代	标记-清除算法	响应速度优先	适用于互联网或B/S业务
<b>G1</b>	并发、并行运行	作用于新生代、老年代	标记-压缩算法、复制算法	响应速度优先	面向服务端应用

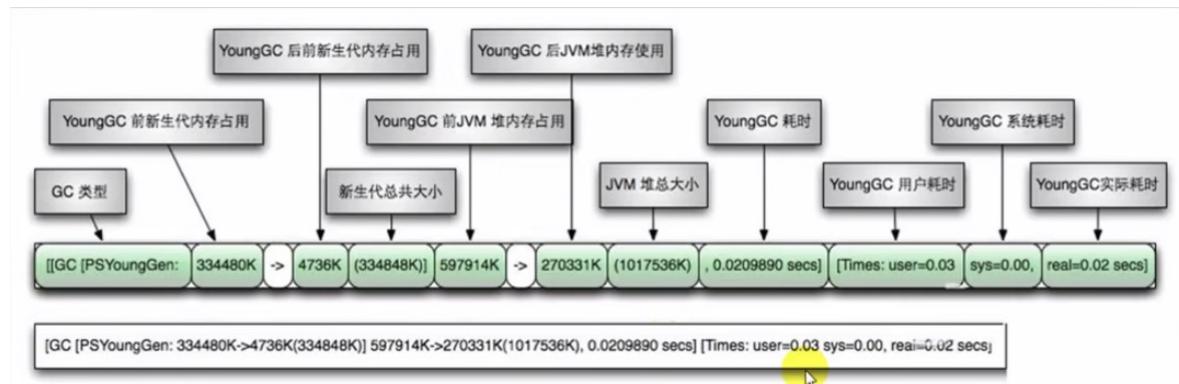


通过阅读GC日志，我们可以了解Java虚拟机内存分配与回收策略。

### 内存分配与垃圾回收的参数列表

- XX:+PrintGC** 输出GC日志。类似: **-verbose:gc**
- XX:+PrintGCDetails** 输出GC的详细日志
- XX:+PrintGCTimeStamps** 输出GC的时间戳（以基准时间的形式）
- XX:+PrintGCDateStamps** 输出GC的时间戳（以日期的形式，如2013-05-04T21:53:59.234+0800）
- XX:+PrintHeapAtGC** 在进行GC的前后打印出堆的信息
- Xloggc:.../logs/gc.log** 日志文件的输出路径

### YoungGC



### Full GC

