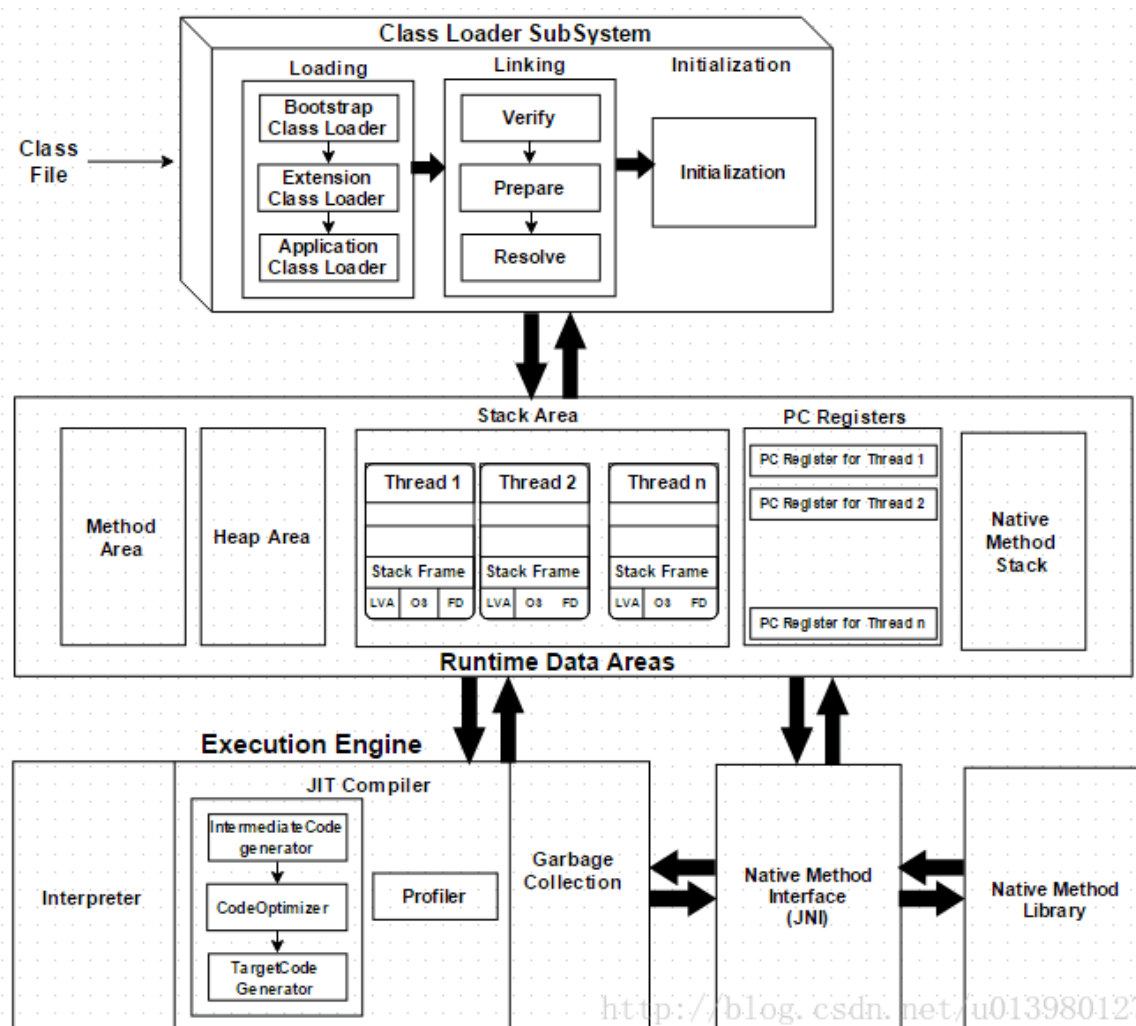
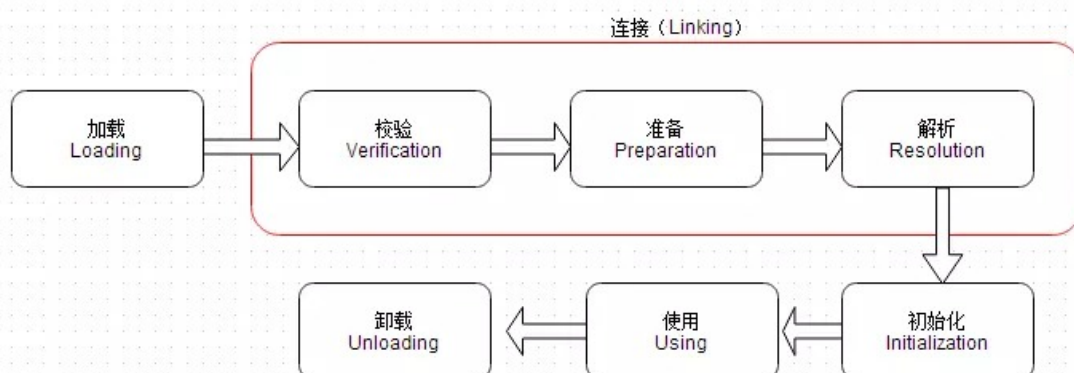


分享：Java虚拟机

1、虚拟机的构成



2、类加载子系统



2.1、加载

过程

加载阶段，虚拟机完成3件事情：

1. 通过一个类的全限定名来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（HotSpot为方法区）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

加载完成之后，类的二进制字节流就按照虚拟机的格式规范存储在内存中。

类加载器

- 启动类加载器 BootStrap ClassLoader

用C++语言编写并嵌入到VM内部，主要加载JAVA_HOME/jre/lib目录下，或者加载由选项-Xbootclasspath或系统变量sun.boot.class.path指定的目录，并且被虚拟机识别的（指定的文件名，如rt.jar），名称不符合的即便在目录中也不能被加载

文件名	描述
rt.jar	运行环境包，rt即runtime，J2SE 的类定义都在这个包内
charsets.jar	字符集支持包
jce.jar	是一组包，它们提供用于加密、密钥生成和协商以及 Message Authentication Code（MAC）算法的框架和实现
jsse.jar	安全套接字拓展包Java(TM) Secure Socket Extension
classlist	该文件内表示是引导类加载器应该加载的类的清单
net.properties	JVM 网络配置信息

- 扩展类加载器 Extension ClassLoader

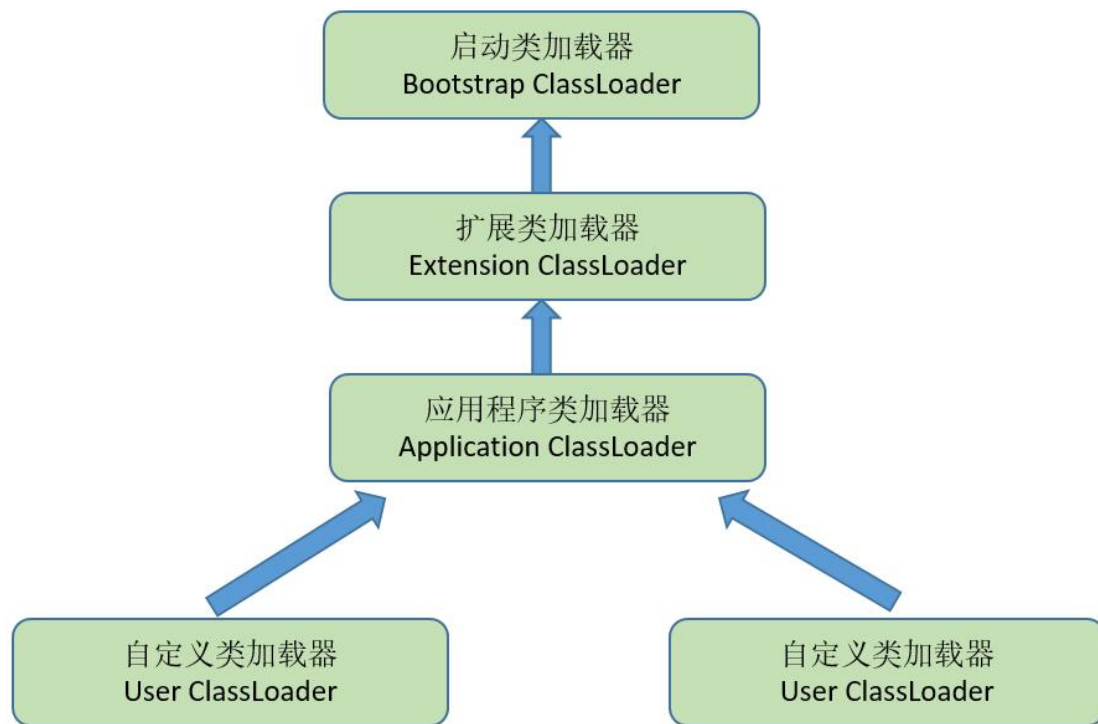
继承ClassLoader类，由sun.misc.Launcher\$ExtClassLoader实现，加载JAVA_HOME/jre/lib/ext目录中的所有类库，或者系统变量java.ext.dirs所指定的路径中的所有类库

- 应用程序类加载器 Application ClassLoader

由sun.misc.Launcher\$AppClassLoader实现，由于这个类加载器是ClassLoader中的getSystemClassLoader()方法的返回值类型，所以一般也称它为系统类加载器。它负责加载用户类路径（ClassPath）上所指定的类库，如果程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认类加载器

- 自定义加载器

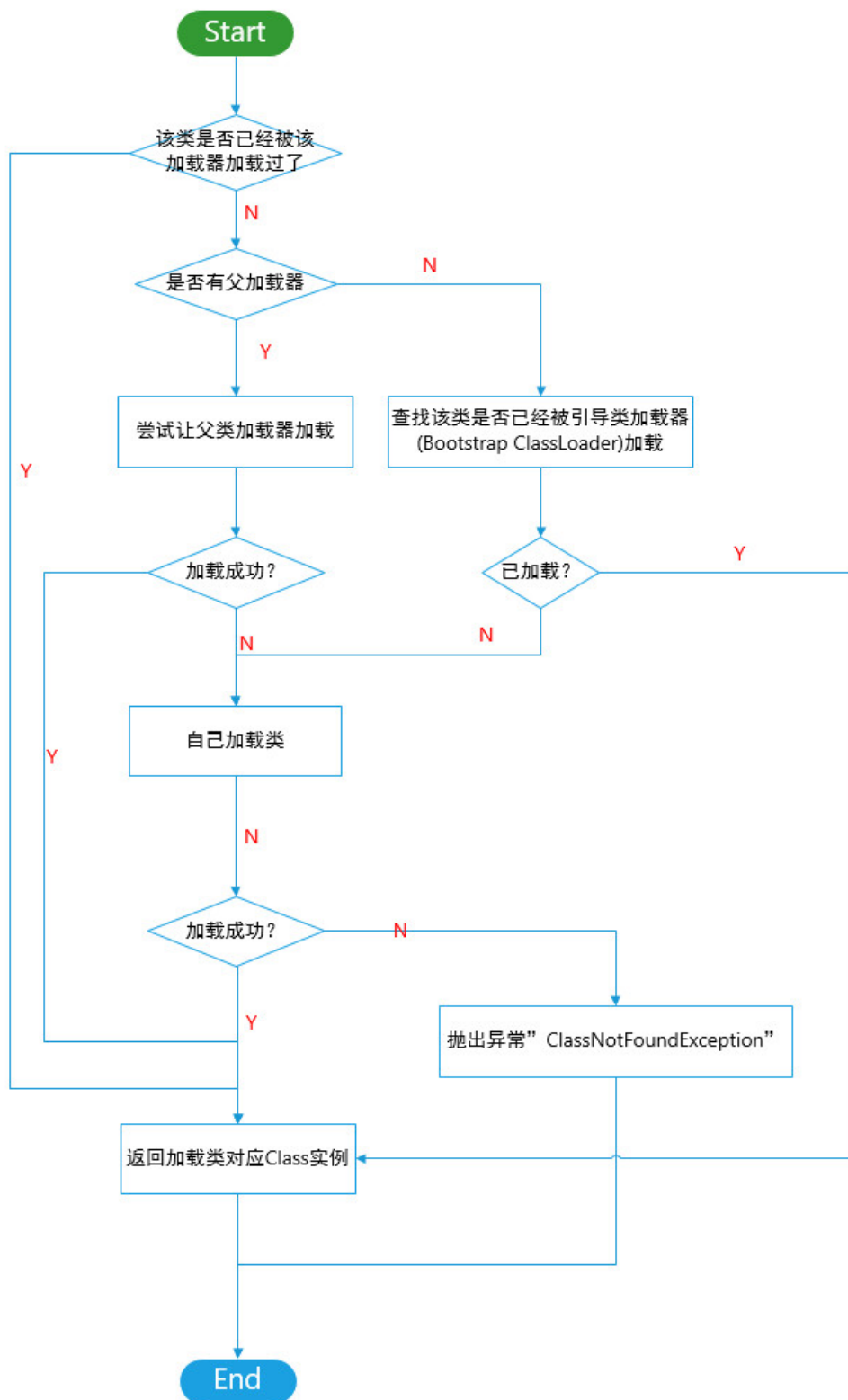
双亲委派机制



- 原理

1. 类加载器收到加载类的请求
2. 将加载请求委派给它的父类加载器，直至启动类加载器
3. 父类加载器无法完成加载请求时（搜索范围内无该类），子加载器尝试自己加载
4. repeat #3

应用类加载器(AppClassLoader)工作流程图



Designed by LouLuan
<http://blog.csdn.net/luanlouis>

- 优势
 1. 类加载和类本身确定其在虚拟机中的**唯一性**
 2. Java类随着它的类加载器一起具备了一种带有优先级的层次关系
 3. 保持了虚拟机的安全，比如rt.jar中的类，即使用户定义了相同的类，也无法完成加载
- 破坏

1. Tomcat
2. JNDI
3. 热部署

2.2、验证

验证阶段主要包括四个校验过程

1. 文件格式验证：字节流是否符合Class文件格式规范
 - 魔数：是否0xCAFEBABE开头
 - 版本：主次版本号是否在虚拟机处理范围内
 - ...
2. 元数据验证：对字节码描述的信息进行语义分析
 - 类是否有父类（Object除外）
 - 是否继承了不允许被继承的类（如：final修饰的）
 - 如果其父类是抽象类，是否实现了父类或接口要求实现的所有方法
 - 类中字段、方法是否与父类产生矛盾

覆盖了父类的final字段

重载方法参数类型一致返回值类型不一致

这个过程把类、字段和方法看做组成类的一个个元数据，然后根据JVM规范，对这些元数据之间的关系进行验证。所以，元数据验证阶段并未深入到方法体内。

3. 字节码验证：通过数据流和控制流分析程序语义的合法性，即类的方法体的校验分析
 - 保证时刻操作数栈与指令代码序列能配合工作
 - 保证跳转指令不会跳转到方法体以外的字节码指令上
 - 保证方法体的类型转换是有效的

例如可以把一个子类对象赋值给父类数据类型，这是安全的，但不能把一个父类对象赋值给子类

这个过程是保证被校验类的方法在运行时不会做出危害虚拟机安全的行为。

4. 符号引用验证：类的常量池中各种符号引用的信息进行匹配性校验
 - 符号引用中通过字符串描述的全限定名是否能找到对应类
 - 指定类中是否存在符合方法的字段描述符以及简单名称所描述的方法和字段
 - 符号引用中的类、字段和方法的访问性是否可被当前类所访问

这个过程做的工作主要是验证字段、类方法以及接口方法的访问权限、根据类的全限定名是否能定位到该类等。

2.3、准备

1. 类变量分配空间

2. 类变量分配零值

数据类型	零 值
int	0
long	0L
short	(short) 0
char	'\u0000'
byte	(byte) 0
boolean	false
float	0.0f
double	0.0d
reference	null

3. 类常量分配初始值：由类字段的ConstantValue属性进行赋值

2.4、解析

实际上就是把常量池中的符号引用替换为直接引用的过程

• 符号引用

在常量池中即**非字面量**的类型

```
CONSTANT_Class_info
CONSTANT_Fieldref_info
CONSTANT_Methodref_info
CONSTANT_InterfaceMethodref_info
...
```

• 直接引用

直接指向目标的虚拟机内存中的指针
相对偏移量
能够定位到目标的句柄

• 符号引用的解析

1. 类或接口的解析
2. 字段解析
3. 类方法解析
4. 接口方法解析

[详解参考](#)

2.5、初始化

初始化就是执行<clinit>()方法的过程

关于<clinit>()

- 是编译期生成在Class字节码中的，由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static{...}中的语句合并而成
- 是类构造器，与实例构造器<init>()不同，虚拟机保证会在调用前先调用其父类的<clinit>()，因此不需要显式调用父类构造器
- 父类的<clinit>()对类变量的赋值操作优先于子类的<clinit>()执行
- <clinit>()并非必需，如果类中无静态代码块或对类变量的赋值操作，那么编译器可以不生成<clinit>()方法，Class字节码中也就没有<clinit>()方法

- 接口无静态代码块但是可以用类变量赋值操作，因此也会生成<clinit>方法，但是不需要先调用父接口的<clinit>()方法，只有父接口的类变量使用时才调用<clinit>()方法初始化父接口
- 虚拟机会保证多线程环境下类的<clinit>()方法可以阻塞地调用，即线程T1调用类C的<clinit>()方法初始化C的过程中，线程T2会阻塞而无法进入类C的<clinit>()方法中的

初始化时机

- 在类没有进行过初始化的前提下，当执行new、getStatic、setStatic、invokeStatic字节码指令时，类会立即初始化。对应的java操作就是new一个对象、读取/写入一个类变量（非final类型）或者执行静态方法
- 在类没有进行过初始化的前提下，当一个类的子类被初始化之前，该父类会立即初始化
- 在类没有进行过初始化的前提下，当包含main方法时，该类会第一个初始化
- 在类没有进行过初始化的前提下，当使用java.lang.reflect包的方法对类进行反射调用时，该类会立即初始化
- 在类没有进行过初始化的前提下，当使用JDK1.5支持时，如果一个java.lang.reflect.MethodHandle实例最后的解析结果REF_getStatic、REF_putStatic、REF_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化

3、运行时数据区

3.1、程序计数器

Program Counter Register，当前线程所执行的字节码的指令地址

1. 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理
2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了
3. 如果线程正在执行一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Native方法，这个计数器的值则为(Undefined)

线程私有，不会出现OutOfMemoryError

3.2、虚拟机栈

VM Stack，描述的是Java方法执行的内存模型

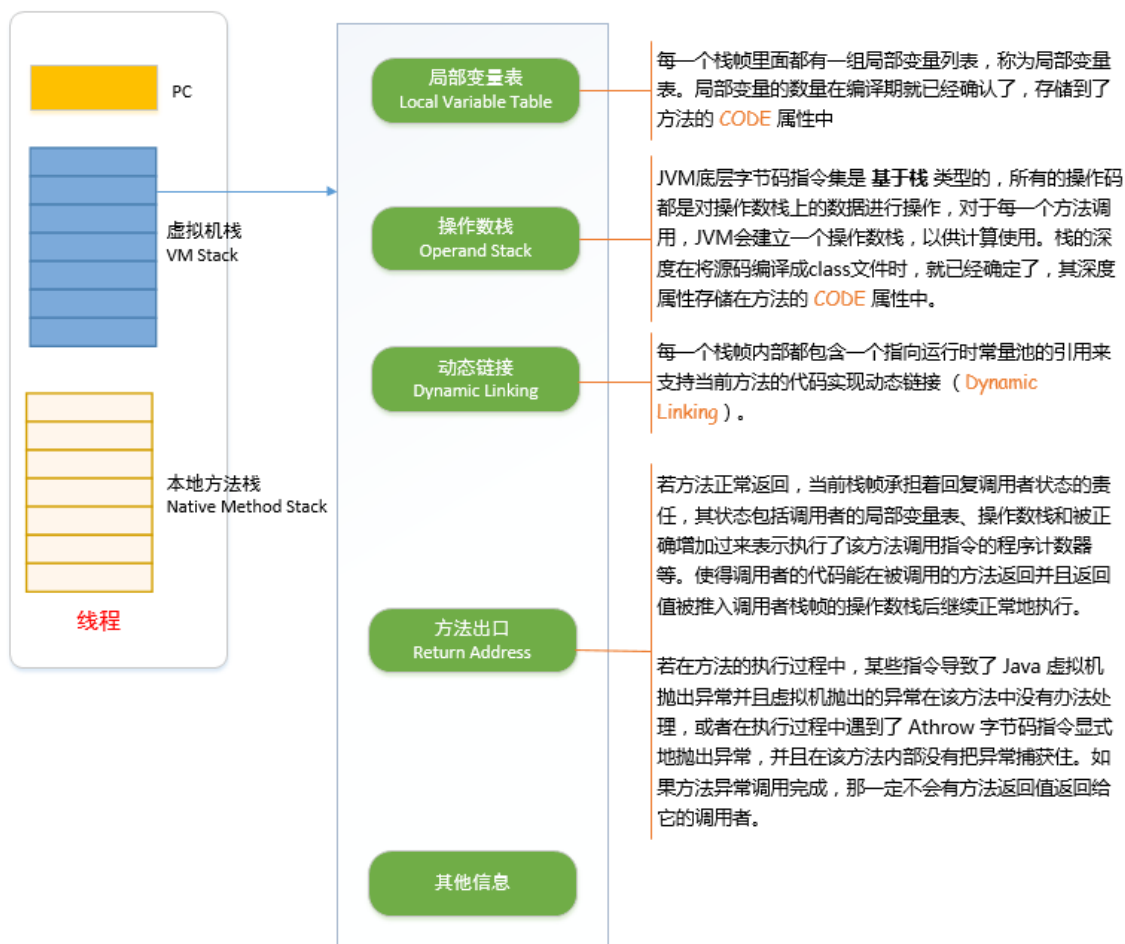
栈帧

每个线程有一个私有的栈，每个方法执行时都会创建一个栈帧（Stack Frame），而每个栈帧中都拥有局部变量表（基本数据类型和对象引用）、操作数栈、动态链接、方法出口信息

每个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中的入栈到出栈的过程

Topic 3. 栈帧是什么？栈帧里有什么？

栈帧 Stack Frame



Designed by LouLuan
<http://blog.csdn.net/luanlouis>

局部变量表

主要存放方法参数和方法内部定义的局部变量，是编译器可知的各种数据类型 (boolean、byte、char、short、int、float、long、double)、对象引用 (reference类型，它不同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置)

两种异常

- StackOverflowError

-Xss

若Java虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前Java虚拟机栈的最大深度的时候，就抛出StackOverflowError异常

- OutOfMemoryError

若 Java 虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完了，无法再动态扩展了，此时抛出OutOfMemoryError异常

实际很难模拟

线程私有

3.3、本地方法栈

Native Method Stack，与虚拟机栈作用相似，区别不过是虚拟机栈为虚拟机执行Java方法服务，而本地方法栈为使用到的Native方法服务

在HotSpot虚拟机中和Java虚拟机栈合二为一实现

线程私有，同虚拟机栈一样，会抛出StackOverflowError和OutOfMemoryError两种异常

3.4、堆

Java Heap，用于存放对象实例

所有的对象
数组

内存分配方式

- 指针碰撞

原理

假设堆内存绝对连续规整，用过的内存放在一边，空闲内存存在另一边，中间放着一个指针作为分界点指示器；分配内存，即是指针向空闲方向移动与对象大小相等的距离

实现

带compact过程的回收器

- 空闲列表

原理

虚拟机维护一个列表，记录哪些内存块是可用的，分配时选择足够大的内存块给对象实例，并更新列表

实现

Mark-Sweep算法的回收器：CMS

分代

- 新生代 Young Generation

- Eden
- Survivor
 - From Survivor
 - To Survivor

内存配比 8:1:1

GC回收器：Serial、ParNew、Parallel Scavenge、G1

Minor GC (YGC)：Eden->Survivor->老年代(对象GC age达阈值，默认Max=15)

- 老年代 Old/Tenured Generation

大对象直接进入

GC回收器：Serial Old、Parallel Old、CMS、G1

Major GC (Full GC)：

- 永久代 Permanent Generation

Hotspot专属 ≤1.7

线程共享，会抛出OutOfMemoryError异常

3.5、方法区

Method Area，用于存储被虚拟机加载的类信息、常量、静态变量（字符串常量池）、即时编译器（JIT）编译后的代码等数据

虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它有个别名：Non-Heap（非堆），目的应该是与Java堆区分开

Hotspot把GC分代收集扩展到方法区，或者说用**永久代**来实现方法区

Topic 4. 方法区是什么？方法区里有什么？



Designed by LouLuan
<http://blog.csdn.net/luanlouis>

运行时常量池

Runtime Constant Pool，是方法区的一部分

用于存放编译期生成的各种字面量和符号引用，在类加载后



线程共享，会抛出OutOfMemoryError异常

3.6、Native Memory

不属于运行时数据区的部分，在JDK8中，Native Memory 包括Metaspace和C-Heap

- Metaspace

>1.7

接管了原永久代的：

1. 字面量：转移到Java heap
2. 类的静态变量：转移到Java heap
3. 符号引用：转移到Native heap

- Native heap

就是C-Heap。对于32位的JVM，C-Heap的容量=4G-Java Heap-PermGen；对于64位的JVM，C-Heap的容量=物理服务器的总RAM+虚拟内存-Java Heap-PermGen

关于Native heap介绍很少，[参考文章](#) 略有提及。

- 直接内存区

不在运行时数据区，1.4新加入NIO，使用Native函数分配堆外内存，通过一个存储在Java 堆里面的DirectByteBuffer 对象作为这块内存的引用进行操作

4、垃圾回收

4.1、GC Roots

- 判断对象存活的算法

- 引用计数算法 Reference Counting

给对象添加一个引用计数器，有引用，计数器+1；引用失效，计数器-1；引用为0即是要回收的

存在相互循环引用的问题

- 可达性分析算法 Reachability Analysis

通过GC Roots对象为起点，向下搜索，当一个对象到GC Roots没有任何引用链时，则证明此对象是不可用的。

可作为GC Roots的对象

- 虚拟机栈（栈帧中的本地变量表）中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中常量引用的对象

■ 本地方法栈中JNI (Native方法) 引用的对象

• finalize

如果对象没有与GC Roots关联的引用链，会作为**第一次标记**并经过一次“筛选”，筛选的原理：判断对象是否有必要执行finalize()方法，**不必要**的条件有：

- 对象没有覆盖finalize()方法
- finalize()方法已经被虚拟机调用过，只会被虚拟机调用1次

有必要时的执行过程

1. 对象放置在叫做F-Queue的对象
2. Finalizer线程执行队列中对象的finalize()方法，此线程由虚拟机创建，低优先级
3. GC对F-Queue中的对象进行第二次标记，如果对象逃脱回收，则从队列移除；否则，将被回收

4.2、GC 算法

4.2.1 标记-清除算法

Mark-Sweep，分“标记”和“清除”两个阶段。

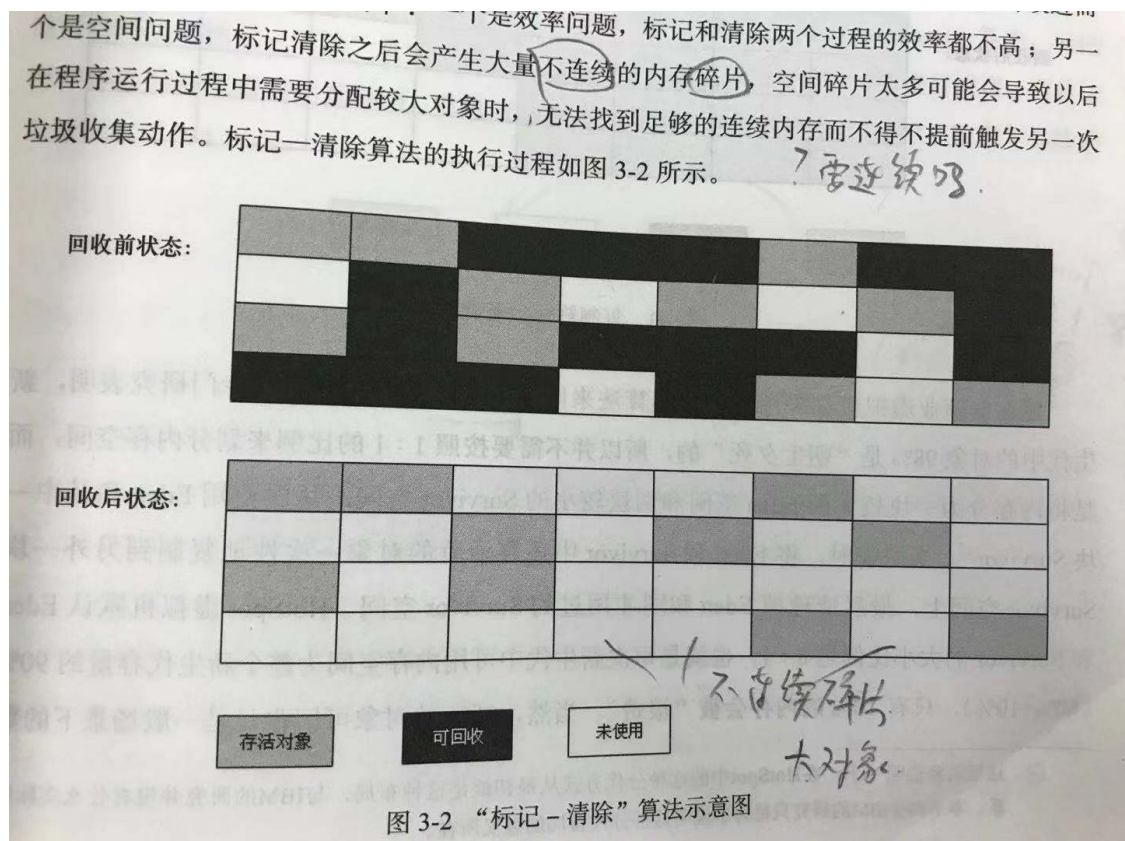
• 过程

首先，标记出所有需要回收的对象（遍历GC Roots）
标记完成后，统一回收**被标记**的对象

• 缺点

两个过程效率都不高
产生大量不连续的内存碎片，碎片太多可能会导致分配大对象内存时，无法找到足够的连续空间而提前触发一次GC

• 示意图



4.2.2 复制算法

Copying, 为解决效率问题, 将可用内存按容量划分为**大小相等**的两块, 每次只用其中的一块。当这一块内存用完了, 将还存活的对象复制到另一块上面, 然后再把原内存空间一次清理

- 优点

- 无碎片
 - 实现简单
 - 高效

- 缺点

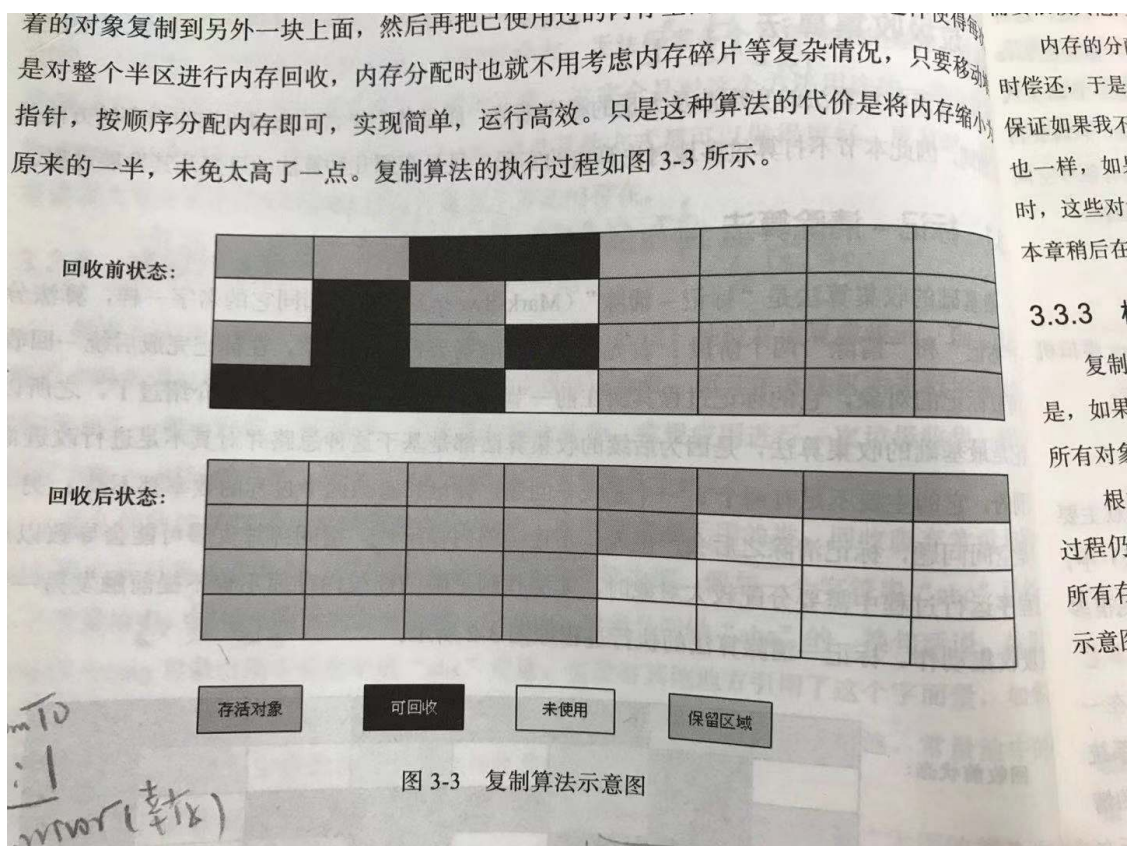
- 预留一半的空闲内存

- 应用

- 新生代: 98%朝生夕死, Eden->Survivor

- 分配担保: 如果Survivor没有足够空间存在Eden存活的对象, 将直接进入老年代

- 示意图



4.2.3 标记-整理算法

Mark-Compact, 分为标记-整理-清除3个阶段, 因为垃圾回收最终目的都是清除可回收对象, 所以略去了“清除” (个人总结)

- 过程

- 标记过程仍与“标记-清除”算法一样, 但后续步骤不是直接对可回收对象进行清除
 - 将存活对象都移向一端
 - 清理掉“端边界”以外的内存

- 优点

- 解决了“标记-清除”算法产生的碎片问题

不存在“复制”算法内存成本问题和空间担保问题

- 示意图

过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存，“标记-整理”算法的示意图如图 3-4 所示。

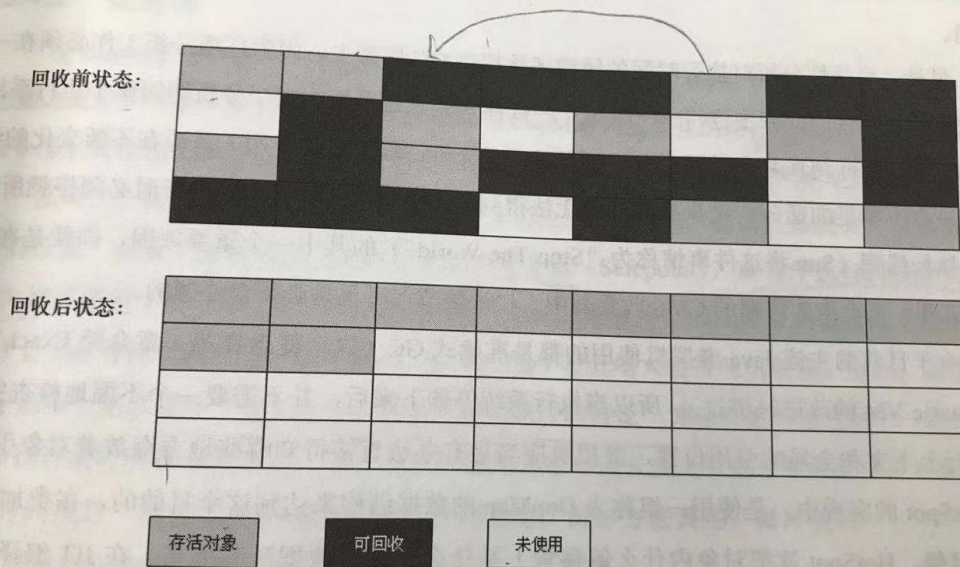


图 3-4 “标记-整理”算法示意图

4.2.4 分代收集算法

Generational Collection，并不是一种具体的算法实现，而是对算法利用的策略（个人总结）。

思想是根据对象**存活周期的不同**将内存划分为几块

一般是把Java堆分为新生代和老年代，根据各个年代的特点采用适当的收集算法

4.3、GC 收集器

算法是方法论，收集器是对内存回收的具体实现。

- 分代及组合

用特点和要求组合出各个年代所使用的收集器。这里讨论的收集器基于 JDK 1.7 Update 14 之后的 HotSpot 虚拟机（在这个版本中正式提供了商用的 G1 收集器，之前 G1 仍处于实验状态），这个虚拟机包含的所有收集器如图 3-5 所示。

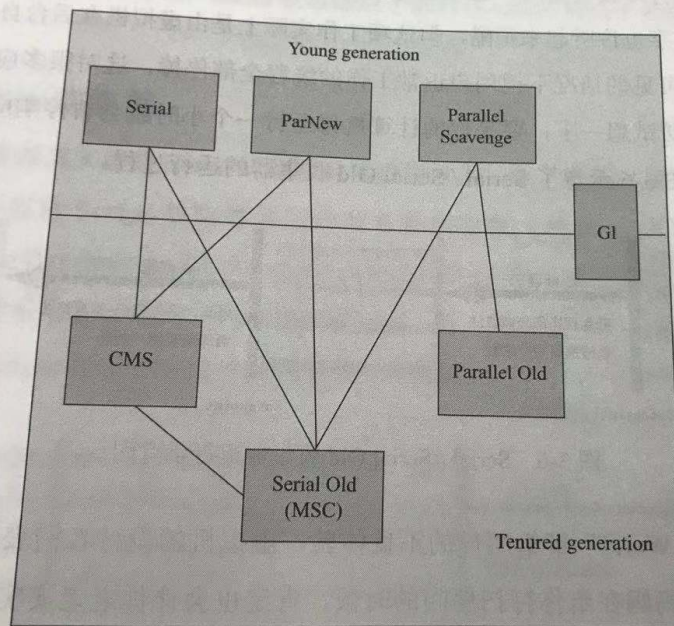


图 3-5 HotSpot 虚拟机的垃圾收集器^②

4.3.1 Serial

是一个**单线程**的收集器，Client模式下的默认**新生代**收集器；有新生代和老年代版本；**中断**用户线程

- 特点

只会用一个CPU或一个线程完成垃圾收集工作

垃圾收集时，暂停其它所有的工作线程（Stop The World）

- 分代及算法

Serial New：年轻代收集器，使用复制算法

Serial Old（MSC）：老年代收集器，使用Mark-Compact算法

- 示意图

“单线程”的意义并不仅仅说明它只会使用一个 CPU 或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。“Stop The World”这个名字也许听起来很酷，但这项工作实际上是由虚拟机在后台自动发起和自动完成的，在用户不可见的情况下把用户正常工作的线程全部停掉，这对很多应用来说都是可以接受的。读者不妨试想一下，要是你的计算机每运行一个小时就会暂停响应 5 分钟，你会有什么样的心情？图 3-6 示意了 Serial / Serial Old 收集器的运行过程。

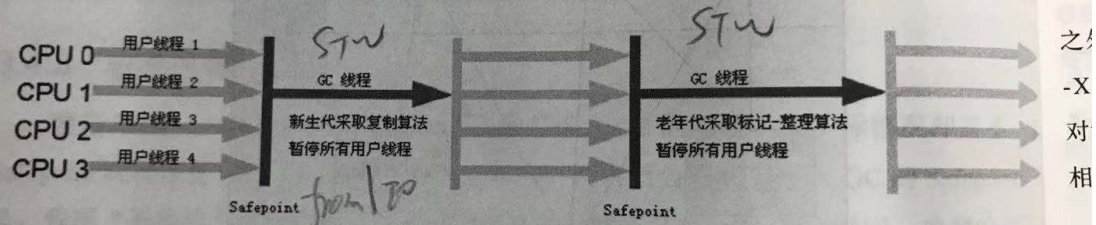


图 3-6 Serial / Serial Old 收集器运行示意图

对于“Stop The World”带给用户的不良体验，虚拟机的设计者们表示完全理解，但也表示非常委屈：“你妈妈在给你打扫房间的时候，肯定也会让你老老实实地在椅子上或者房间外待着，如果她一边打扫，你一边乱扔纸屑，这房间还能打扫完？”这确实是一个合情合理的矛盾，虽然垃圾收集这项工作听起来和打扫房间属于一个性质的，但实际上肯定还要比

4.3.2 ParNew

是Serial收集器的多线程版本，同时并行多个垃圾收集线程，会中断用户线程；是新生代收集器。

- 特点

新生代只有它和Serial收集器能与CMS（收集老年代）配合使用
单CPU环境下，因线程交互的开销，性能不见得会比Serial要好

- 算法

复制算法

- 配置

使用ParNew收集器配置：-XX:+UseConcMarkSweepGC或者-XX:+UseParNewGC
限制线程数：-XX:ParallelGCThreads

- 示意图

...是可以接受的。所以，Serial 收集器对于运行在 Client 模式下的虚拟机来说是一个很好的选择。

3.5.2 ParNew 收集器

ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括 Serial 收集器可用的所有控制参数（例如：-XX:SurvivorRatio、-XX:PretenureSizeThreshold、-XX:HandlePromotionFailure 等）、收集算法、Stop The World、对象分配规则、回收策略等都与 Serial 收集器完全一样，在实现上，这两种收集器也共用了相当多的代码。ParNew 收集器的工作过程如图 3-7 所示。

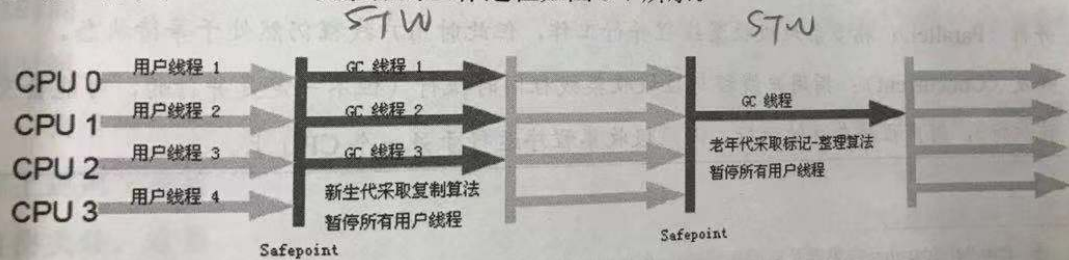


图 3-7 ParNew / Serial Old 收集器运行示意图

4.3.3 Parallel Scavenge

新生代收集器，多线程并行，中断用户线程，吞吐量优先

- 算法

复制算法

- 特点

注重吞吐量，提供了参数用于控制吞吐的参数

GC自适应调节策略

- 参数

- -XX:MaxGCPauseMills: 收集器尽可能的保证回收时间不超过设定值。缩短GC停顿时间是以牺牲吞吐量和新生代空间换取的，GC次数相对会多【关注停顿时间时设置这个】
- -XX:GCTimeRatio: 垃圾回收时间占总时间占比【关注吞吐量时设置这个】
- -XX:+UseAdaptiveSizePolicy: 开启后，不需要手工设定新生代大小 (-Xmn)、Eden 和Survivor比例 (-XX:SurvivorRatio)、晋升老年代对象大小 (-XX:PretenureSizeThreshold) 等参数，虚拟机会根据当前运行情况收集监控信息，动态调整以提供最合适的停顿时间或最大吞吐量

4.3.4 Serial Old

是Serial收集器的老年代版本，单线程

- 特点

主要意义是Client模式虚拟机使用

Server模式用法：≤1.5 与Parallel Scavenge配合使用；作为CMS发生Concurrent Mode Failure时后备方案

- 算法

标记 - 清除

- 示意图

同Serial

4.3.4 Parallel Old

是Parallel Scavenge的**老年代**版本，**多线程并行**，**中断**用户线程

- 特点

≥1.6

之前，老年代收集器只能选择Serial Old，单线程“拖累”了整体性能

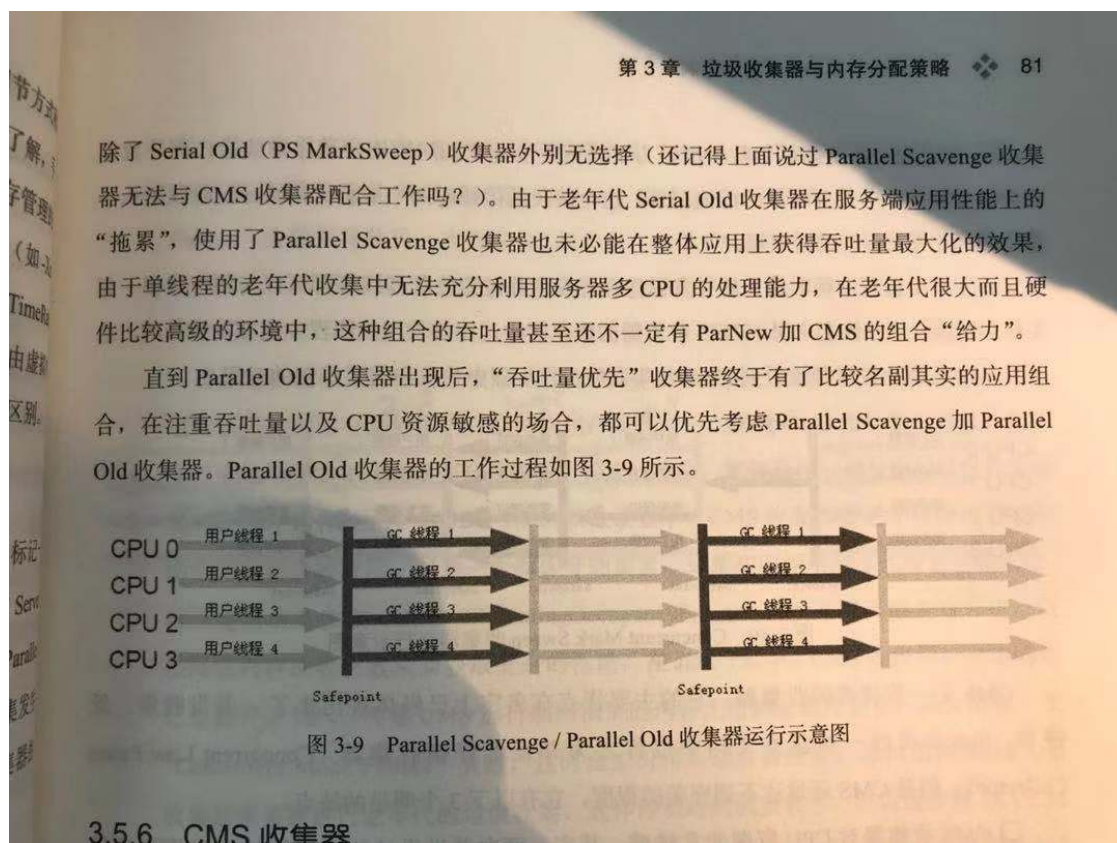
- 算法

标记 - 整理

- 组合方案

Parallel Scavenge + Parallel Old

- 示意图



4.3.5 CMS

Concurrent Mark Sweep，以获得最短回收停顿时间为目标；≥ 1.5

- 特点

并发收集

低停顿

- 缺点

有碎片

CPU资源敏感，默认线程数=(CPU数+3)/4

浮动垃圾：并发标记时用户线程新产生的垃圾，可能出现"Concurrent Mode Failure"

不能像其他收集器那样等老年代几乎完全填满才收集，需要预留空间提供并发收集时程序运行

- 算法

标记 - 清除

- 步骤

初始标记 (CMS initial mark)： **中断** 用户线程，仅标记出 GC Roots 能【直接关联】到的对象，速度快

并发标记 (CMS concurrent mark)： 进行 GC Roots Tracing 的过程，就是需要标记出 GC roots 关联到的对象的引用对象

解释区别：比如 A->B (A 引用 B)，A 是 **初始标记** 过程标记的与 GC Roots **直接关联** 的对象，那么 **并发标记** 过程就是标记出 B

重新标记 (CMS remark)： **中断** 用户线程，为了修正并发标记期间因用户程序继续运作而导致标记变动的那一部分对象的标记记录； **多线程** 执行

补充区别：停顿时间比初始标记稍长，但远比并发标记短，且是多线程

并发清除 (CMS concurrent sweep)：回收垃圾对象

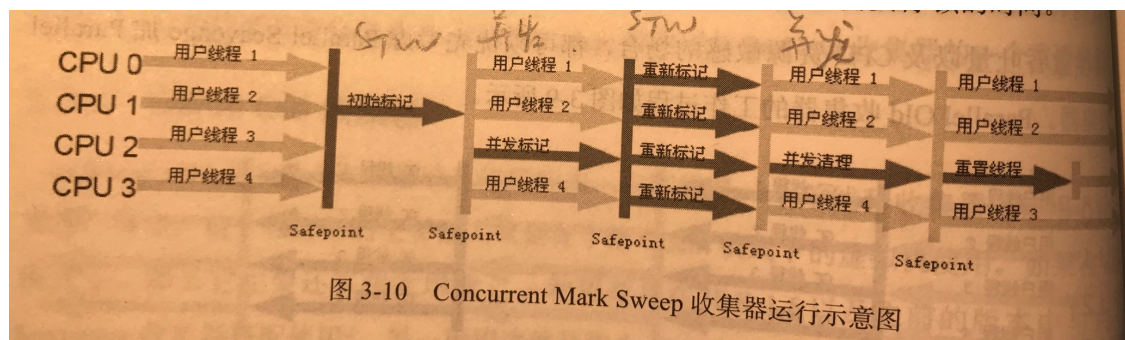
- 参数

-XX:CMSInitiatingOccupancyFraction：触发比例，1.5 默认 68% 1.6 默认 92%

-XX:+UseCMSCompactAtFullCollection：FGC 之后执行一次碎片整理，默认开启

-XX:CMSFullGCsBeforeCompaction：执行多少次不压缩的 FGC 后，进行一次压缩；默认 0：每次都整理

- 示意图



[详解参考](#)

4.3.6 G1

Garbage First, Since JDK7U9, 在 JDK9 中作为默认收集器

- Why Garbage-First

G1 performs a concurrent global marking phase to determine the liveness of objects throughout the heap. After the mark phase completes, G1 knows which regions are mostly empty. It collects in these regions first, which usually yields a large amount of free space. This is why **this** method of garbage collection is called Garbage-First

并发标记后，G1知道哪些区域几乎为空，它首先回收会产出大量空闲空间的这些区域。

白话：优先回收那些可以释放大量空间的区域。

As the name suggests, G1 concentrates its collection and compaction activity on the areas of the heap that are likely to be full of reclaimable objects, that is, garbage. G1 uses a pause prediction (预测) model to meet a user-defined pause time target and selects the number of regions to collect based on the specified pause time target.

G1专注回收和压缩看似充满可被回收对象的区域。G1使用了一个停顿可预测模型来满足用户定义的停顿时间目标，基于这个时间目标选择一定数据的区域。

- G1 vs Old collectors



↑ All memory objects end up in one of these three sections ↑



↑ 整个堆被分为相同大小的Region ↑

- 新概念

分区 (Region)

- 相同大小 (split into many fixed sized regions)

Region size is chosen by the JVM at startup. The JVM generally targets around 2000 regions varying in size from 1 to 32Mb

- 保留分代概念 (spaces)

these regions are mapped into logical representations (描述; 表现) of Eden, Survivor, and old generation spaces

- 巨型分区 (Humongous regions)

直接分配在老年代分区，且一定是连续的

These regions are designed to hold objects that are 50% the size of a standard region or larger. They are stored as a set of contiguous (相邻的; 连续的) regions

- 空间不连续性
- 每种分代大小没有固定限定，可动态变化

收集集合 (Collection Sets or CSets)

the set of regions that will be collected in a GC . All live data in a CSet is evacuated (copied/moved; 转移) during a GC. Sets of regions can be Eden, survivor, and/or old generation. CSets have a less than 1% impact on the size of the JVM.

- 将被回收的分区集合，其中存活的数据被转移 (到另外Region)
- 可以是Eden、Survivor或者老年代
- 占用空间不到1%

已记忆集合 (Remembered Sets or RSets)

track object references into a given region. There is one RSet per region in the heap. The RSet enables the parallel and independent collection of a region. The overall footprint impact of RSets is less than 5%.

将对象引用追踪到指定的区域（记录谁引用了我）。堆中每个区域都有一个RSet，RSet使得并行和独立的收集一个region成为可能。RSets整体大小小于5%。

摘自R大的解释：G1 GC则是在points-out的card table之上再加了一层结构来构成points-into RSet：每个region会记录下到底哪些别的region有指向自己的指针，而这些指针分别在哪些card的范围内。这个RSet其实是一个hash table，key是别的region的起始地址，value是一个集合，里面的元素是card table的index。举例来说，如果region A的RSet里有一项的key是region B，value里有index为1234的card，它的意思就是region B的一个card里有引用指向region A。所以对region A来说，该RSet记录的是points-into的关系；而card table仍然记录了points-out的关系。

- 特点

多核、大内存服务器：heap sizes of around 6GB or larger

有限的GC延迟（limited GC latency）：stable and predictable pause time below 0.5 seconds

- 切换到G1

如果你的应用有以下至少一个特点

- Full GC durations are too long or too frequent：Full GC时间太长或太频繁
- The rate of object allocation rate or promotion varies significantly：创建对象的速率或者晋升比率差异较大
- Undesired long garbage collection or compaction pauses (longer than 0.5 to 1 second)：不希望垃圾回收或者压缩有较长的停顿

- GC原理

- Overview

G1 copies objects from one or more regions of the heap to a single region on the heap, and in the process both compacts and frees up memory. This evacuation is performed in parallel on multi-processors, to decrease pause times and increase throughput

G1将对象从1个或多个区域copy到单个区域，在这个过程中会压缩和释放内存。这种回收在多CPU上是并行的，以降低停顿时间、增加吞吐量。

注：清除采用copying算法

Thus, with each garbage collection, G1 continuously works to reduce fragmentation, working within the user defined pause times. This is beyond the capability of both the previous methods. CMS (Concurrent Mark Sweep) garbage collector does not do compaction. ParallelOld garbage collection performs only whole-heap compaction, which results in considerable pause times

G1持续地工作以减少碎片，在用户定义的停顿时间内工作。这超出了以往（回收）方式的能力，CMS收集器不会压缩，ParallelOld执行整个堆的压缩，需要相当多的停顿时间。

It is important to note that G1 is not a real-time collector. It meets the set pause time target with high probability but not absolute certainty.

很重要的一点是G1不是一个实时的收集器，它会尽可能的满足停顿时间目标，但并不绝对。

Based on data from previous collections, G1 does an estimate of how many regions can be collected within the user specified target time.

基于以往收集到（统计到）的数据，G1会估算在用户指定的时间目标内可以回收多少区域。

Thus, the collector has a reasonably accurate model of the cost of collecting the regions, and it uses **this** model to determine which and how many regions to collect **while** staying within the pause time target.

这样，收集器对收集这些区域的成本有一个精准模型，使用这个模型来确定在停顿时间内收集哪些和多少区域。

Note


G1 has both **concurrent** (runs along with application threads, e.g., refinement, marking, cleanup) and **parallel** (multi-threaded, e.g., stop the world) phases. Full garbage collections are still single threaded, but **if** tuned properly your applications should avoid full GCs.

G1有并发（与应用线程一起运行，如：~~refinement~~不懂，标记，清除）和并行阶段（多线程，如：STW）。

Full GC仍旧是单线程的，但是如果调教的比较好的应用，应该避免Full GCs。

- Step by Step
 - Young GC in G1

Live objects are **evacuated** (i.e., copied or moved) to one or more survivor regions. If the aging threshold is met, some of the objects are promoted to old generation regions.

（Eden）存活对象被转移到一个或多个幸存者区，如果达到晋升阈值，其中一些对象晋升到老年代。

This is a stop the **world** (STW) pause. Eden size and survivor size is calculated **for** the next young GC. Accounting information is kept to help calculate the size. Things like the pause time goal are taken into consideration

这阶段是STP停顿的。Eden和Survivor大小会被计算为下此YGC使用，统计信息被保存起来帮助计算大小，像停顿时间目标这事儿会被考虑。

This approach makes it very easy to resize regions, making them bigger or smaller as needed.

这种方式使得调整区域大小很方便，根据需要变大或者变小。



Summary

- The heap is a single memory space split into regions

堆是一个单独的内存空间，被分隔为多个region

- Young generation memory is composed of a set of non-contiguous regions. This makes it easy to resize when needed

新生代内存是由一些不连续的region集合构成的，方便按需调整大小

- Young generation garbage collections, or young GCs, are stop the world events. All application threads are stopped for the operation

YGC是STW的，所有的用户线程为了（FGC）这个操作而被停止

- The young GC is done in parallel using multiple threads

YGC是以多线程并行完成的

- Live objects are copied to new survivor or old generation regions

存活对象被复制到新的幸存者或者老年代region

■ Old Generation Collection with G1

1. Initial Mark (*Stop the World Event*)

This is a stop the world event. with G1, it is piggybacked (在..基础上; 借助于..) on a normal young GC. Mark survivor **regions** (root regions) which may have references to objects in old generation.

标记出Survivor区中有引用老年代对象的作为root regions

More: 通常初始标记阶段会跟一次新生代收集一起进行，换句话说就是，G1借用了新生代收集来完成初始标记的工作；在初始标记或新生代收集中被拷贝到survivor分区的对象，都需要被看做是根；

类似于CMS初始标记，仅标记直接可达

2. Root Region Scanning

Scan survivor regions **for** references into the old generation. This happens **while** the application continues to run. The phase must be completed before a young GC can occur.

扫描从上阶段标记的root regions（Survivor分区）。与应用程序并发执行，这个阶段必须在下次YGC触发前完成。

More: *why no ygc* 因为Survivor分区是作为root开始扫描，如果新生代执行收集，又有新的存活对象复制到Survivor，另一个原因猜测是上一个阶段已经YGC一次了（个人理解）。

3. Concurrent Marking

Find live objects over the entire heap. This happens **while** the application is running. This phase can be interrupted by young generation garbage collections.

标记整个堆中存活对象：被Survivor引用的老年代对象。可以被YGC打断

4. Remark (*Stop the World Event*)

Completes the marking of live object in the heap. Uses an algorithm called ~~snapshot-at-the-beginning~~ (SATB) which is much faster than what was used in the CMS collector.

跟踪上面阶段进行中又被更新存活的对象，找到未被标记的存活对象。完成这个堆的存活对象标记。

5. Cleanup (*Stop the World Event and Concurrent*)

1. Performs accounting on live objects and completely free regions. (Stop the world)
2. Scrubs the Remembered Sets. (Stop the world)
3. Reset the empty regions and **return** them to the free list. (Concurrent)

1. 统计存活对象和完全空闲的region

6. 清理RSet

7. 重置空region，释放到free list

(*) Copying (*Stop the World Event*)

These are the stop the world pauses to evacuate or copy live objects to **new** unused regions. This can be done with young generation regions which are logged as [GC **pause** (young)]. Or both young and old generation regions which are logged as [GC **Pause** (mixed)].

Initial Marking Phase 初始标记阶段

Initial marking of live object is piggybacked on a young generation garbage collection

初始标记出在年轻代垃圾收集存活的对象，在日志中记为：

GC **pause** (young)(initial-mark)



Concurrent Marking Phase 并发标记阶段

If empty regions are **found** (as denoted by the "X"), they are removed immediately in the Remark phase. Also, "**accounting**" information that determines liveness is calculated.

如果发现了空的region，它们在**重新标记**阶段直接被移除，“accounting”信息也会计算其存活性



Remark Phase 重新标记阶段

Empty regions are removed and reclaimed. Region liveness is now calculated for all regions.

空的region会被移除和回收，计算所有region的存活性。



Copying/Cleanup Phase 复制/清除阶段

G1 selects the regions with the lowest "liveness", those regions which can be collected the fastest. Then those regions are collected at the same time as a young GC. This is denoted in the logs as [GC pause (mixed)]. So both young and old generations are collected at the same time.

G1选择最低“存活性”的区域，这些区域会被最快速度回收。这些区域在YGC时也会被回收，所以新生代和老年代是同时回收的。



After Copying/Cleanup Phase 复制/清除之后

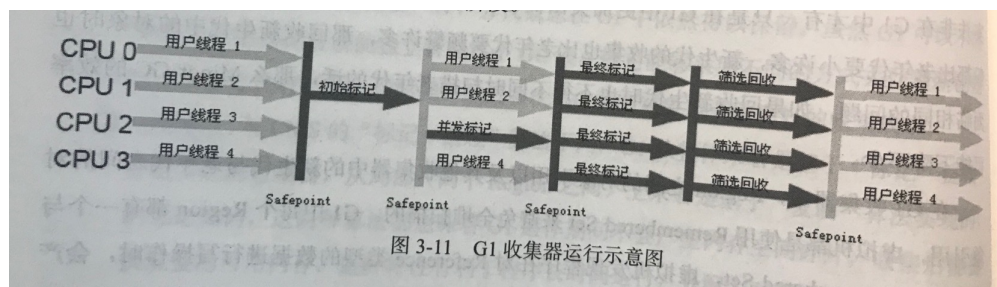
The regions selected have been collected and compacted into the dark blue region and the dark green region shown in the diagram.

被收集压缩到深蓝色和深绿色区域

Summary

- Concurrent Marking Phase 并发标记阶段
 - Liveness information is calculated concurrently while the application is running：在应用程序运行中（对象的）存活性被计算出来（并发地）
 - This liveness information identifies which regions will be best to reclaim during an evacuation pause：在暂停期间（不明白？），这些存活性信息确认哪些区域可以被更好的回收
 - There is no sweeping phase like in CMS：没有CMS的清除阶段
- Remark Phase 重新标记阶段
 - Uses the Snapshot-at-the-Beginning (SATB) algorithm which is much faster than what was used with CMS：使用SATB算法，更快
 - Completely empty regions are reclaimed：空region被回收
- Copying/Cleanup Phase 复制/清除阶段
 - Young generation and old generation are reclaimed at the same time：新生代和老年代同时被回收
 - Old generation regions are selected based on their liveness：基于它们的存活性选出老年代region

- 示意图



[Oracle tutorials](#)

4.4、Full GC

- Minor GC、Major GC、Full GC



从年轻代 (Eden、Survivor) 回收内存，叫Minor GC

对老年代回收，叫Major GC

Full GC是对整个堆、方法区 (永久代、Metaspace)

Full GC时经常伴随一次Minor GC，也非绝对。Major GC速度一般比Minor GC慢10倍以上

- 触发条件

System.gc(): 不一定执行

老年代空间不足:

- 不足以存放Survivor中满足晋升条件的对象
 1. 从Eden区诞生开始，每经过一次Survivor，age+1，达到设定阈值（默认15）后，晋升到老年代，若此时，晋升对象的大小超过老年代空间，则触发Full GC
 2. 统计 Survivor区域中的相同年龄的对象占到所有对象的一半以上时，就会将大于这个年龄的对象移动到老年代，若此时老年代空间不足
- 连续空间不足
 1. 大对象即是需要大的连续空间存储的对象，分配此类对象是在老年代，若无连续空间，则触发Full GC

CMS GC时出现promotion failed和concurrent mode failure:

- promotion failed 空间分配担保失败，Minor GC之前，会比较“老年代剩余空间”与“新生代所有对象之和”的大小，如果老年代剩余空间大于新生代之和，则本次Minor GC一定是安全的，反之，去看参数是否允许担保失败，不允许，则会触发一次Full GC
- concurrent mode failure CMS无法处理浮动垃圾，因为CMS运行和程序运行是并行的，标记之后产生的垃圾CMS处理不了，此时程序如果申请一块儿空间，内存不够，就会发生concurrent mode failure。此时虚拟机会启用serial old收集器来进行老年代的回收，停顿时间会很长

永久代空间不足

Metaspace达设定阈值: -XX:MetaspaceSize (默认约20.8M)，JVM会动态调整

4.5、GC log

5、高阶篇

5.1、JVM配置参数

5.2、Java内存模型

5.3、锁与同步机制

5.4、监控及排障

5.5、执行子系统

5.5.1、类文件结构

5.5.2、字节码指令集

