# JVM基础

jvm的组成

jvm的作用

# 类的加载

## 类的加载

### 什么是类的加载？

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其存放在运行时数据的方法区内

然后在堆区创建一个java.lang.Class对象，用来封装在方法区内的数据结构，作为各种方法的出入口

类加载的最终产物是位于堆区的Class对象。

### 加载（获取）.class文件的方式有哪些？

本地系统、网络（Socket）、jar或者zip、源文件编译、专有数据库

### 具体的加载类的方式有哪些？

命令行启动时由JVM加载

通过Class.forName()方法动态加载

通过ClassLoader.loadClass()方法动态加载

PS：Class.forName()方法会对类进行解释，执行类中的static块，而ClassLoader.loadClass()方法不会执行，除非使用参数控制

## 类的生命周期

### 类生命周期的五个阶段？

加载、验证、准备、解析、初始化、（结束）

### 五个阶段分别完成哪些事情？

加载：类加载过程的第一个阶段

1.通过一个类的全限定名获取其定义的二进制字节流；

2.将这个字节流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Java堆中生成一个代表这个类的java.lang.Class对象，作为方法区的这些数据的访问入口

**PS：相对于类加载的其他阶段，加载阶段获取二进制字节流的动作是可控性最强的阶段，因为这个阶段既可以使用系统提供的类加载器，也可以使用自定义的类加载器来完成。**

验证：类链接过程的第一步，是为了保证被加载类的正确性

1.**文件格式验证**：验证字节流是否符合Class**文件的格式规范**。比如指定的文件开头，版本号，常量池中的常量是否存在不支持的类型等；

2.**元数据验证**：对字节码进行语义分析，确保其符合Java**语言规范**的要求；

3.**字节码验证**：通过数据流和控制流分析语义合法且**符合逻辑**；

4.**符号引用验证**：确保解析动作能正确执行

PS：验证很重要，但是不是必须的。如果一个类已经经过了多次验证，可以通过-Xverifynone参数来关闭验证措施，用来缩短虚拟机的类加载时间

准备：正式为（**方法区中的**）类变量（static变量）分配内存并且设定初始值。

1.只为类变量分配内存，不包括实例变量（实例变量放在Java堆中）；

2.分配内存的时候会给类变量赋系统默认初始值（不是代码中指定的初始值），比如0，null，false之类，因为参数的赋值需要使用类构造器中的<clinit>()方法，而此时还没有执行任何的方法；

3.如果参数同时使用了static和final修饰（public static final int x = 3），则会直接把变量赋值为指定值（没有final这里x=0）

解析：把类中的符号引用（描述目标的字面量）转换为直接引用（指向目标的指针、相对偏移量或者间接定位到目标的句柄）

初始化：为类的静态变量赋初值（有static没有final的变量）（可能来自指定的初始值，比如public static int x = 3，也可能来自静态方法块，static{ //TODO}）

初始化步骤：

1.该类没有加载或连接->加载或连接该类；

2.父类没有初始化->初始化父类；

3.按步骤初始化

初始化时机：

1.new方法创建实例；

2.访问（或修改）类或者接口的静态变量；

3.调用类的静态方法；

4.反射；

5.初始化某个类的子类会引起父类的初始化；

6.JVM启动时被标明为启动的类。

### 什么是静态绑定？什么是动态绑定？

绑定：一个方法的调用与方法所在的类(方法主体)关联起来。

静态绑定：在程序执行前（编译阶段）完成绑定，Java中只有final，static和private方法采用静态绑定。

动态绑定：在运行时根据具体对象的类型进行绑定

**PS：动态绑定是多态的底层原理**

### 有哪几种结束类的生命周期的方式？

1. 执行System.exit()方法
2. 程序正常结束
3. 程序异常终止
4. 其他错误导致的jvm进程终止

## 类加载器

### 类加载器的层次关系？

按照层次顺序

启动类加载器（Bootstrap ClassLoader）最上层

扩展类加载器（ExtClassLoader），第二层

应用类加载器（AppClassLoader）第三层

自定义类加载器

### 最上层的类加载器是什么？

启动类加载器（Bootstrap ClassLoader）

### 为什么扩展类加载器没有父Loader了？

启动类加载器是由C语音实现的，ExtClassloader访问父加载器的时候会没有结果

### 没有父loader时访问父loader的方法返回什么？

null

### 几种类加载器的作用分别是什么？

启动类加载器：负责加载存放在/lib目录下，或者由-Xbootclasspath参数指定的目录中的类。启动类加载器无法被java程序直接引用

扩展类加载器：加载/ext目录下的或者java.ext.dirs系统变量所指定的路径中的类。可直接使用。

应用程序类加载器：加载用户路径类classpath所指定的类。可直接使用。如果程序中没有自定义其他的类加载器，那么此加载器是默认加载器。

## 类的加载机制

### 三种类加载机制及其解释

全盘负责：一个类加载器负责加载一个类时，该Class所依赖的其他class也有该加载器负责载入（除非显式指定了其他加载器）。

父类委托：先让父类加载器试图加载，父类加载失败以后再尝试从自己的类路径中加载

缓存机制：所有已经加载过的Class会被缓存，下次加载时可以直接调用缓存，缓存区不存在时再尝试加载。

**PS：当修改了Class以后，要重启JVM，否则可能一直读取的时缓存区中的旧Class**

## 双亲委派模型

### 简述双亲委派模型

当一个加载器面临一个加载任务时，不会主动尝试加载，而是先提交给父加载器（App ClassLoader交给ExtClass Loader，ExtClass Loader交给Bootstrap ClassLoader），当父类加载器找不到所需要的类时再返还给子加载器，交给子加载器自行完成。

如果AppClassLoader加载失败，会报出ClassNotFoundException异常

### 双亲委派模型的意义

防止出现重复加载的类

保证java程序安全运行（防止加载了某些自定义的类，覆盖了默认的类，比如重写了String类）

## 自定义的类加载器

### 什么时候需要使用自定义的类加载器？

当已有的三个加载器功能不能满足需要的时候，比如：

1. 执行不受信代码前进行数字签名验证
2. 动态创建符合需要的定制化构建类
3. 从特定场所取得类，比如socket或者数据库

### 父类是什么？

自定义类加载器一般继承自ClassLoader类

### 什么方法需要重写？为什么？

需要重写findClass方法

因为自定义加载器的最主要目的是从旧加载器找不到的地方获取类，或者对类进行特定的预处理

PS：需要自定义类加载器才能加载的类，最好不能被AppClassLoader直接加载，否则根据双亲委派模型，会绕过自定义类加载器，可能会丢失功能

### 什么方法最好不要重写？为什么？

最好不要重写loadClass方法，因为容易破坏双亲委派模式

# JVM内存结构

## JVM内存结构

### JVM内存分为哪几块？ （Java7及以前）

java堆（Java Heap）

方法区（Method Area）

程序计数器（Program Counter Regist）

虚拟机栈（JVM Stack）

本地方法栈（Native Method Stack）

### java8中产生了什么新变化？有什么改进？为什么要采取这样的变化？

方法区（和interned-strings组成持久代）在java8以前是jvm虚拟机内存的一部分

Java8开始不再使用持久代，改用直接放在本地内存中的MetaSpace（元数据区），使得可以使用的内存不再受到虚拟机内存的限制（可以通过其他参数去设置Metaspace的内存上限）

为什么：在相对早期的时候，实例对象频繁的产生和弃用，因此将对象的内容放在Java堆中，根据实例存活的时间又划分了新生代和老年代。但是类的卸载发生的频率很低，所以类的信息在jvm生存周期中认为是长期存在的，于是将存储类静态信息的方法区又成为永久代（持久代）。

然而随着自定义类加载器使用的增加，类的加载和卸载变得频繁。由永久代OOM内存不足引起的Full GC频率增加，永久代中的内容也不再变得“永久”。因此将此部分内容移交到本地内存中，改称为元数据区（Metaspace）。在没有指定上限的情况下可以脱离jvm内存大小的限制（如果指定了较小的上限，一样会影响jvm性能）。

此外，由于字面量（interned-strings）之前就对永久代产生较大影响容易引起Full GC，所以在Java7中就已经移走，转移到Java heap中。

## java堆（heap）

### 基本概念

Java堆是jvm所管理的内存中最大的一块，目的是存放实例变量。

可以不采用连续的物理内存

虚拟机启动的时候创建

### 堆内存放什么？

对象实例、以及从java7以后还负责存放字面量（interned-strings）

Java堆是GC 收集器管理的主要区域（Full GC时还要牵涉到方法区），因此又称为“GC堆”

### 堆的组成

新生代：Eden，From Survivor和To Survivor（两个Survivor区有时候又称为S0区和S1区）

老年代

### 如果堆的内存不够了怎么办？

可扩展时先进行扩展

无法扩展时抛出OutOfMemoryException异常

### 为谁所拥有？

java堆由所有线程所共享

## 方法区（Methods Area）

### 方法区存放什么？

已经被虚拟机所加载的类的信息、常量、静态变量、即时编译器编译后的代码等数据

### 方法区有什么别名？

非堆（No-Heap）

有时会被成为永久代（Permanent Generation）/持久代（PermGen），是因为GC分代收集的方法扩展到了方法区，所以提出了和新生代老年代类似的一个称呼。有时候也说使用永久代来实现方法区。

### 当方法区内存不够时会怎么样？

抛出OutOfMemoryException异常

### 为谁所拥有？

所有线程

## 程序计数器（Program Counter Register）

### 作用？

当前线程所进行的自己码的行号指示器，用来指示当前指令和选取下一条指令

分支、循环、转跳、异常处理、线程恢复等功能都以来这个计数器

### 为谁所拥有？

每一个线程拥有独立的程序计数器

生命周期和线程相同

如果该线程执行的时java方法，则程序计数器记录虚拟机字节码指令的地址

如果执行native方法，则计数器值为null

PS：唯一不会出现OutOfMemoryException异常的区域

## JVM栈（虚拟机栈，jvm stack）

### 存放了哪些东西？

存放java方法执行的内存模型（栈帧）

每个java方法执行时会创建一个栈帧，用于存放局部变量表，操作栈，动态链接，方法出口等内容。

方法调用时入栈，方法完成该栈帧出栈

### 什么情况会出现异常？会抛出哪些异常？

如果不允许虚拟机栈动态扩展内存大小，线程请求的栈深度过深，抛出StackOverFlowException异常

如果允许虚拟机栈的动态扩展，无法通过扩展申请到足够的内存时抛出OutOfMemoryException异常

### 为谁所拥有？

每个线程拥有独立的虚拟机栈

虚拟机栈的生命周期和线程相同

### 实例对象在Java堆中的生存过程

对象首先进入新生代的Eden区

当一次对象清理完成后，Eden区存活的对象和From Survivor区存活的对象全部进入To Survivor区，原本的Eden区和From Survivor区清空

把To survivor区标记成From，把原来的From清空后标记为To

存活的对象年龄+1

对象年龄超过给定阈值，则进入老年代

To survivor区满，则新生代中的全部对象进入老年代

较大对象（体积超过Survivor区一半）的年龄阈值会降低

超大对象不进入Eden区，直接进入老年区

## 本地方法栈（Native Method stack）

### 与虚拟机栈的区别？

本地方法栈为虚拟机执行的native方法（非Java方法）服务

### 为谁所拥有？

每个线程拥有独立的本地方法栈

本地方法栈的生命周期和线程相同

## 控制参数

### 常见的控制参数有哪些？分别有什么含义？

-Xms 设置Java堆初始化时分配的内存空间

-Xmx 设置java堆的最大内存空间

-XX:NewSize 设置新生代初始化时分配的内存空间

-XX:SurvivorRatio=3 设置Eden区和Survivor区大小的比值（默认值是8：1：1，这里变成3：1：1，也就是Eden区占据新生代空间的3/5）

（JVM默认采用了AdaptiveSizePolicy策略，所以为了使调整Eden区和Survivor区大小比值的参数生效，需要使用-XX:-UseAdaptiveSizePolicy。如果想要重新开启这个策略，使用-XX:+UseAdaptiveSizePolicy）

-XX:MaxNewSize 设置新生代的最大内存空间

（没有设置老年代空间的参数，老年代空间大小可以用java堆大小减去新生代大小）

-XX:NewRatio = 3 设置新生代和老年代大小比值（这里代表老年代是新生代的3倍。一般取2~3 的时候性能较高，默认值是2）

-XX:MaxTenuringThreshold 新生代晋升老年代的年龄阈值

-XX:PermSize 设置持久代初始化时分配的内存空间

-XX:MaxPermSize 设置持久代的最大内存空间（server选项下默认64m，client选项下默认32m）（以上两项保留到java1.7）

-XX:MetaspaceSize 设置元空间的初始化值（最小值）

-XX:MaxMetaspaceSize 设置元空间的最大值（如果不设置的话可以不断的申请本地内存）

-Xss 设置每个线程的堆栈大小

## OOM汇总（OutOfMemoryError）

### 有哪些可能抛出OOM的地方？

对象不能分配到堆内存中

类或者方法不能加载到永久代

创建的数组大于了堆的可用空间（如果是空间足够但是零散分配会引发GC）

栈区申请的深度过深

### （怎么判断是来自哪里的？）

统一格式：Exception in thread “main”: java.lang.OutOfMemoryError: reason

Java heap space：对象不能倍分配到堆内存中（堆内存已满且无法扩容，GC也不行）

PermGen space：类不能加载到持久区，可能是因为加载了过多的类或者引用了很多第三方库

Requested array size exceeds VM limit：想要创建的数组大于了堆内存空间

request <size> bytes for <reason>. Out of swap space? ：本地内存分配失败。可能是由于JNI（Java Native Interface，Java调用来自其他语言的本地已编译方法的接口，使用后可能会丧失平台可移植性），本地库或者jvm虚拟机的申请内存失败。

<reason> <stack trace>（Native method）：同样是本地内存分配失败，只不过是JNI或者本地方法或者Java虚拟机发现

# GC（Garbage Collection）基础

## 基本概念

### GC主要应用哪些内存结构中？

Java堆（新生代、老年代）和方法区（持久代/元数据区）

### 对象存活判断的方式？

引用计数：每个对象有一个引用计数属性，新增引用时+1，引用释放时-1，计数为0时回收。（思路简单，但是无法解决对象相互循环引用的问题）

可达性分析（Reachability Analysis）：从GC Roots开始向下搜索，搜索经过的路径称为引用链。如果一个对象到GC Roots没有任何引用链相连，则证明该对象是不可访问的。是不可达对象。

### GC Roots包括哪些？

方法区中静态属性实体引用的对象

方法区中常量引用的对象

虚拟机栈中引用的对象

本地方法栈中JNI引用的对象

### 引用从强到弱的四种状态

强引用

软引用

弱引用

虚引用

### 什么是安全点（Safe Point）？

从GC Roots开始进行可达性分析的时候，为了保持一致性，需要暂停线程（Stop The World，STW），如果线程处在一个引用状态频繁变更的阶段，则会对后续的可达性分析结果准确性产生很大影响（比如某个GC Root迅速变成不可达，但是依然以此为起点）

安全点就是线程可以保证引用状态不变的一个阶段

只有当线程进入安全点时，才可以STW

为了达到这个目标，有两种思路：

1.抢先式中断：即先把线程中断，如果发现中断的位置不是安全点，则恢复线程让其重新跑到安全点再中断（几乎没有被实际采用）；

2.主动式中断：GC需要进行中断时，仅仅是简单的设置一个中断标记。线程到达安全点时去访问这个中断标记，不需要中断就继续运行，发现需要中断则进行中断。

### 什么是安全区（Safe Region）？

主动式中断要求线程必须在执行中，如果一个线程处于阻塞状态时就无法进行主动式中断，但是阻塞状态下线程的引用状态也不会发生改变，所以也应该允许jvm发起GC。

安全区指的是一个线程引用状态不会发生变化的阶段，包括了运行中的线程引用状态稳定的阶段，也包括了没有被分配时间片的线程。因此安全区可以看作安全点的扩展。

当一个线程进入阻塞等确定引用状态不发生改变的阶段时，对外标记自己已经进入安全区，jvm在判断是否可以STW时就会略过这个线程。

当一个线程想退出安全区时，首先要去查询GC Roots的轮询是否已经结束了（保证GC Roots的正确性，引用链状态可能是会变的），如果没结束不得推出，结束了才可以推出安全区继续执行线程任务。

## 垃圾收集算法

### 常见的垃圾收集算法有哪些？

标记-清除（Mark-Swap）算法

复制（Copying）算法

标记-压缩（Mark-Compact）算法

（标记-整理算法）

分代收集算法

### 垃圾收集算法的简述

标记-清除（Mark-Swap）算法：首先标记出所有需要回收的对象，标记完成后统一清理。是最基础的算法，后续收集算法都是基于此思路上的改进。

复制（Copying）算法：将内存化为大小相等的两块，每次只使用其中一块。一块用完后将还存活的对象复制到另一块，把原本的一块一次性清除。

标记-整理（Mark-Compact）算法：

为了解决复制算法在对象存活率较高的时候执行较多的复制操作引起的效率降低问题，也为了处理一半空间浪费的问题，在老年代引入了标记-整理的算法。

该算法让所有存活对象向一段移动，利用连续的空间，然后直接清理掉端边界以外的内存。

分代收集算法：

基本假设：绝大多数对象的生命周期都非常短暂，存活时间短。

主要思想：把java堆分成新生代和老年代，根据各个年代的特点采用不同的收集方法。新生代中每次收集有大批量死去，少量存活，因此采用复制算法，只要付出少量的复制成本就可以完成收集。老年代中对象存活率高，没有额外空间进行分配担保，就要使用标记-清除或者标记-整理算法。

### 各种算法的适用场景

标记-清除（Mark-Swap）算法：老年代

复制（Copying）算法：新生代

标记-整理（Mark-Compact）算法：老年代

### 各种算法的优缺点

标记-清除（Mark-Swap）算法：

主要缺点：

1.时间问题，标记和清除的效率都不高；

2.空间问题，标记清除之后会产生大量的不连续内存碎片，以后要分配较大对象时可能产生连续内存分配不足问题

复制（Copying）算法：

主要优点：实现简单，回收内存高效，不需要考虑内存碎片的情况

主要缺点：代价是将内存缩小为原来的一半，而且长期生存的对象可能会被多次复制导致效率降低

标记-整理（Mark-Compact）算法：

主要优点：解决了复制算法的主要缺陷

主要缺点：只适用于对象生存周期较长的老年代

## GC事件

### GC事件的分类

Young GC（Minor GC）：新生代GC

Old GC（Major GC）：老年代GC，（只有CMS的并发收集是这个模式）

Mixed GC：新生代和部分老年代（G1收集器所特有）

Full GC：全局GC，包括新生代，老年代，方法区（1.7之前是永久区，1.8开始变成元数据区）

### 触发Young GC的条件

Eden区满。jvm无法为新对象在Eden区分配空间

CMS收集器使用了CMSScavengeBeforeRemark参数，在重新标记阶段前强制进行Young GC。

如果在Parallel Scavenge框架下，如果使用了ScavengeBeforeFullGC参数，Full GC之前也会调用一次Young GC（用来减轻Full GC的工作量）

### 触发Old GC的条件

预测Young GC引起的新生代向老年代晋升对象总size超出了老年代的剩余size（Serial Old或Parallel Scavenge Old执行）

老年代的内存使用到达一定阈值（CMS执行）

### 触发Full GC的条件

CMS中产生了Old GC不能解决的问题（并发失败--空间不足，晋升失败--空间碎片化），新生代GC交给原本和CMS收集器搭配的新生代收集器执行，老年代和方法区GC交给Serial Old执行

G1中产生了疏散失败，巨型对象分配失败等情况。

### 触发Mixed GC的条件

使用G1收集器时，老年代使用率到达阈值，触发对老年代的GC。在第一步初始标记开始之前等待一次Young GC，并发标记阶段也可能出现多次Young GC，所以实际上覆盖了新生代和部分老年代。

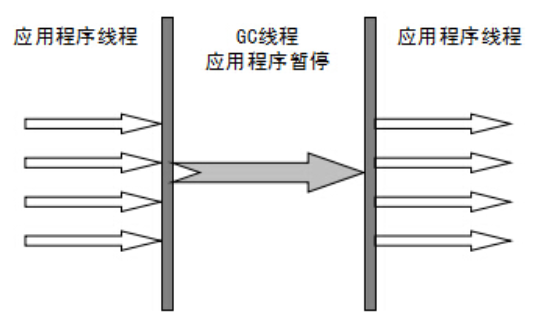
# 垃圾收集器

## Serial（串行）收集器

### 基本思路与特点

单线程垃圾收集器（一条线程收集、收集的同时必须暂停其他所有工作的线程）

最基本、历史最悠久的垃圾收集器。



### 应用场景与对应的垃圾收集算法

新生代：复制

Serial Young GC（单线程串行新生代GC）

### 优缺点

优点：简单高效（没有线程交互的开销），有很高的单线程效率，适合Client模式

缺点：单线程导致STW时间较长，会影响整体效率

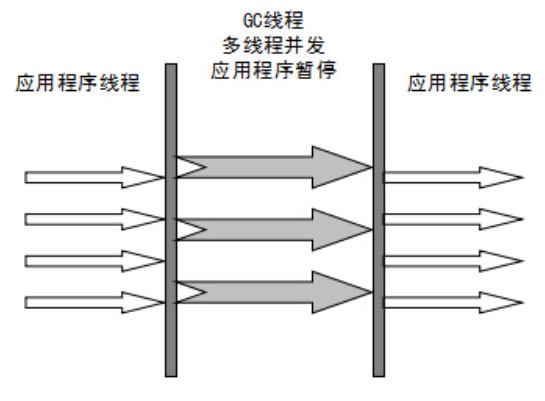
## ParNew收集器

### 基本思路

相当于多线程版本的Serial收集器

除了使用多线程进行垃圾收集以外，其他行为（控制参数、收集算法、回收策略等）都一致

只有ParNew收集器和Serial收集器可以和CMS配合工作



PS：

并行（Parallel）：多条垃圾收集线程同步进行，此时用户线程仍然等待

并发（Concurrent）：用户线程和垃圾收集线程同时执行（不一定是并行，可能是交替执行），用户程序在运行，垃圾收集器运行在另外一个CPU上

### 应用场景与对应的垃圾收集算法

新生代：复制

ParNew Young GC（多线程新生代GC）

### 优缺点

优点：是许多Server模式下虚拟机的首选

## Parallel Scavenge收集器

### 基本思路

多线程的复制算法收集器（看上去和ParNew很像）

区别在于Parallel Scavenge收集器关注吞吐量（CPU使用率）

Parallel Scavenge收集器提供了很多参数，供用户找到最合适的停顿时间或最大吞吐量

可以通过参数控制GC的时间上限或比例

### 应用场景与对应的垃圾收集算法

新生代：复制

Parallel Scavenge Young GC（并发多线程新生代GC）

### 优缺点

优点：收集器通过合适的参数配置可以使吞吐量提高

## Serial Old收集器

Serial收集的老年代版本

Serial Old GC（实际上是老年代+方法区，如果连上新生代的收集器可以视作Full GC）

一是用于JDK1.5之前与Parallel Scavenge收集器搭配使用

二是作为CMS收集器的后备方案。

标记整理算法

## Parallel Old收集器

Parallel Scavenge收集器的老年代版本（多线程，关注吞吐量）

Parallel Scavenge Old GC（实际上是老年代+方法区，如果连上新生代的收集器可以视作Full GC）

标记整理算法

## CMS （Concurrent Mark Sweep）收集器

### 基本思路

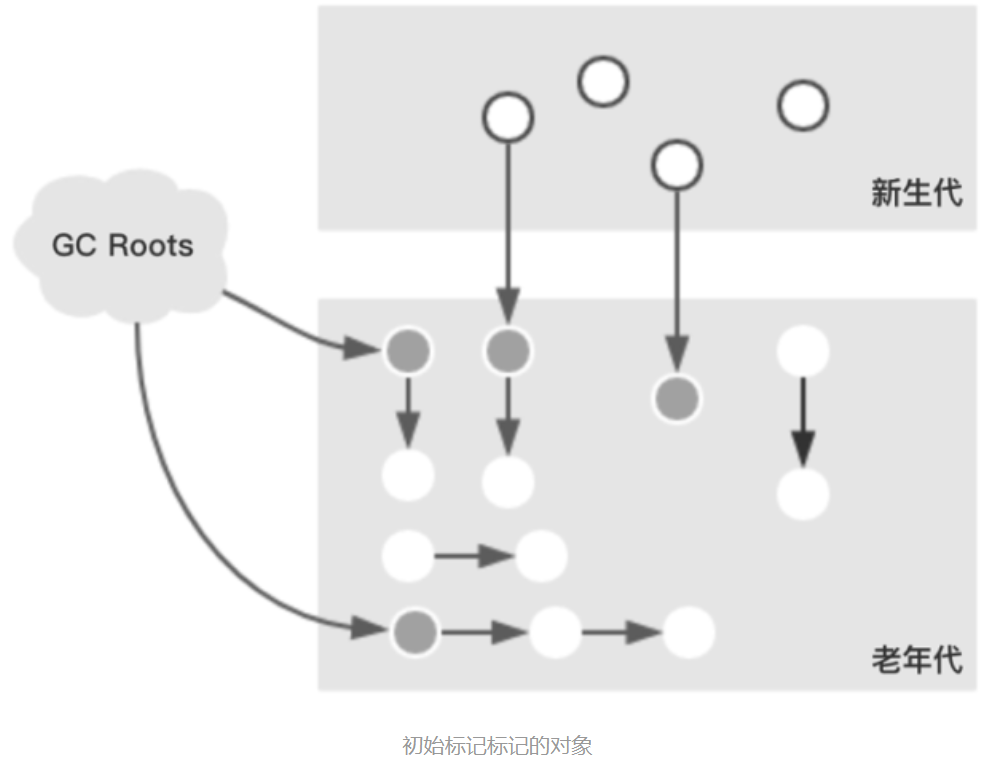
通过让GC步骤和用户线程并发执行，以提供最佳的用户体验

目标：获取最短的回收停顿（STW）时间

是HotSpot虚拟机第一款真正意义上的并发收集器，第一次实现了垃圾收集线程和用户线程（基本上）同时工作。

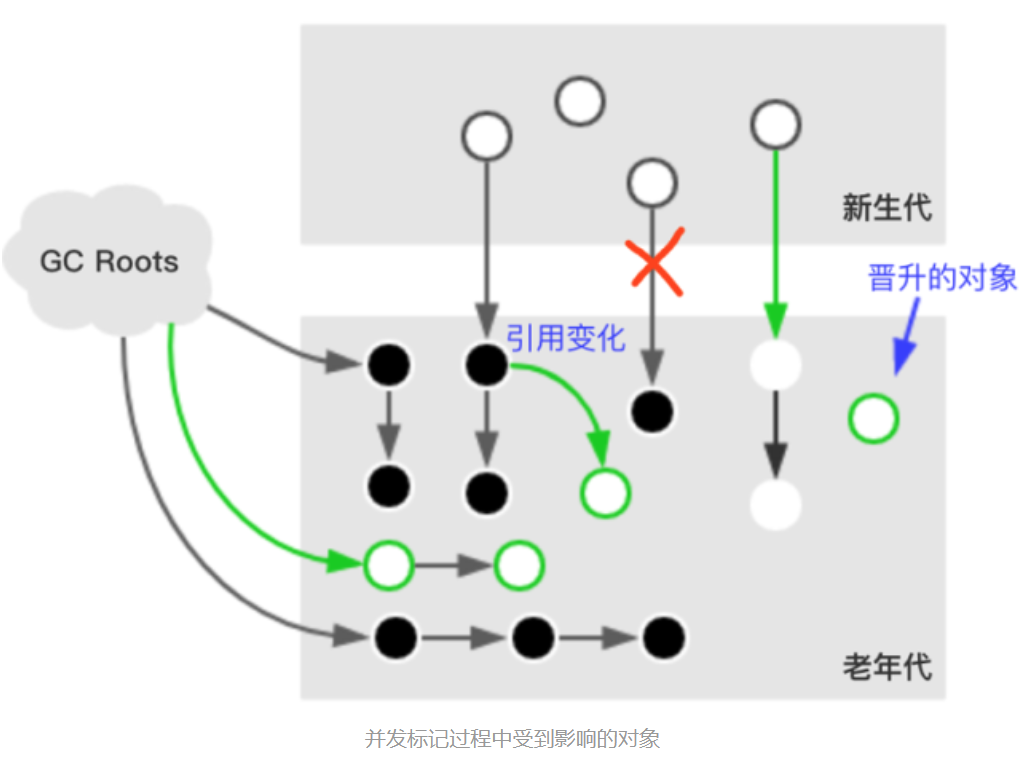
### 过程步骤

**初始标记（Initial Mark）**：目标是获取老年代中的存活对象，暂停其他所有线程（STW），获取GC Roots的直接引用，和新生代中存活对象的引用对象，耗时很短。



**并发标记（Concurrent Mark）**：根据初始标记的结果，进一步的进行GC Tracing，递归的标记老年代中所有的可达对象。这一步耗时较长，但是做到了和用户进程并发执行，不需要STW。

**预清理（Precleaning）**：目的是尽量减少并发标记时产生的新的引用变化带来的影响，给重新标记减轻负担。



本阶段是一个可选阶段，通过参数CMSPrecleaningEnabled调用（默认开启）。主要做两件事：1.判断新生代中对老年代引用情况的变化（需要扫描新生代）；2.扫描老年代中的Dirty Card区，整理老年代内部的新变化。整理完后刷新脏区。

PS（Card）：先将老年代空间逻辑划分为大小相等的区域（Card，512Byte），同时维护一张Card的索引表。如果老年代当中的引用关系发生变化，比如GC Roots对老年代的引用变化，老年代内部对象之间的引用关系的变化，新生代To区对象晋升到老年代，都会将对应对象所在的Card在卡表中标记为Dirty Card（脏区，意思该Card内部发生了变化，哪种变化不知道）

Card Table（卡表）：是Card的一个索引，用来寻找脏区。

**可中断的预清理（Abortable Preclean）**：目标和预清理的一致，减少并发阶段的变化对重新标记带来的压力。此外还有一个目标，就是争取时间，让重新标记开始前出现一次Young GC，还可以避免短时间内两次连续停顿的发生（如果并发标记的时间很短，那么初始标记和重新标记引起的中断就会时间上很接近）。

可中断预清理在Eden使用率达到50%的时候开始，也就是说距离下一次Young GC还有一半的时间。

本阶段不断循环的处理两件事：标记老年代的可达对象，扫描处理Dirty Card区域中的对象（其实就是不断的循环预清理的过程）。

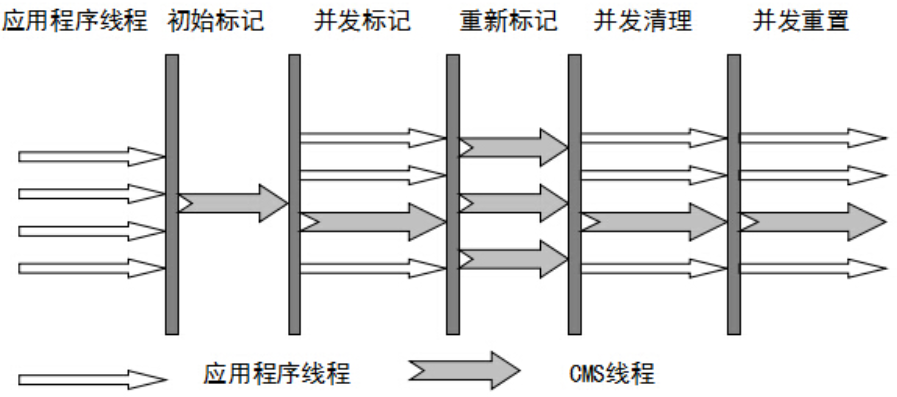
如果达到循环次数要求，或者到达循环时间阈值，则退出本阶段。

PS：如果在预清理阶段之后，Eden区的内存使用绝对值或者占比超出了某些参数的设置（可能意味着Young已经很接近了），那么会跳过本阶段直接进入重新标记阶段。

**重新标记/最终标记（Final Mark）**：第二次STW，修正在并发标记阶段因为用户程序继续运行所导致的引用状态的改变。这一阶段时间远比并发标记阶段短，比初始标记时间稍长。

**并发清除（Concurrent Sweep）**：开启用户线程，同时GC线程对标记的区域做清扫

**并发重置（Concurrent Reset）**：充值CMS算法内部相关数据，为下一次GC做准备



### 应用场景与对应的垃圾收集算法

老年代：标记清除算法

Old GC（并发清除是唯一的单纯的Old GC，但是可能会引起由Serial Old收集器执行的Full GC，还有可能要在Full GC的基础上进行压缩）

### 优缺点

优点：并发收集，低停顿

缺点与应对措施：

对CPU资源敏感：并发的设计模式导致在并发标记和并发清除两个阶段会抢占用户线程的CPU时间，导致CPU速度变慢，整体吞吐量降低。CMS默认的回收线程数是（CPU数量＋3）/4，也就是说当CPU数量在4个以上时使用不少于25%的资源。如果CPU数量不足两个，则可能对用户程序的影响变得很大。

无法处理浮动垃圾：并发清理阶段用户程序还在运行，这个时候产生的新垃圾无法在此次GC被清理，成为浮动垃圾，必须等待下一次的Old GC。为了应对这种情况，CMS不能等到老年代快满了才开始清理，必须预留有一定的空间（一般默认留8%，即老年代使用率达到92%时触发Old GC）给可能出现的浮动垃圾。这个预留比例可以通过参数设置，老年代使用达到阈值是会触发GC。如果预留的空间太大了可能会造成GC次数过多。如果预留的空间不够，则会触发Concurrent Mode Failure失败（并发失败）。

标记清除算法的固有缺陷：标记清除算法会逐渐使得可用空间碎片化（在老年代空间中尤为明显）。如果老年代还有较大的空间，但是由于太零碎了，则会频繁提前触发新的GC。为此CMS收集器提供了两个办法：一个是使用-XX:+UseCMSCpmpactAtFullCollection，在Full GC的时候完成一次内存整理，但是因为内存整理无法并发，会带来额外的时间开支。二是使用一个额外的参数-XX:UseCMSFullGCsBeforeCompaction，用于设置执行多少次不压缩的Full GC以后，来一次带压缩的。

### 可能出现的问题与对应的解决方案

**重新标记阶段停留时间过长的问题**。

原因：常见原因是新生代存在对老年代的无效引用。如果在上一阶段的可中断预清理中没有等到一次Young GC，那么新生代内可能会存在大量对老年代的无效引用，引起本阶段STW时间过长。

措施：为此可以使用CMSScavengeBeforeRemark参数，要求进入本阶段之前一定要引起一次Young GC。不过这个参数会使得，如果可中断的预清理中已经等到了Young GC，还会因为这个参数再进行一遍Young GC。

**并发失败（Concurrent Mode Failure失败）问题**

原因：并发回收阶段老年代空间预留不足。

措施：作为预留方案，此时CMS会临时启动Serial Old收集器（此时就是不单单是老年代了，还包括了方法区）完成对老年代的清理，但是这样可能会导致老年代的清理时间过长。作为预留方案，此时CMS会临时启动Serial Old收集器（此时就是不单单是老年代了，还包括了方法区）完成对老年代的清理，但是这样可能会导致老年代的清理时间过长。

**晋升失败（promotion failed）问题**

原因：老年代有足够的空间但是过于碎片化，新生代晋升到老年代没有足够连续可用空间。

措施：触发单线程的带有压缩的Full GC（由Serial Old收集器执行）。

## G1（Garbage First）收集器

### 设计目标

面向服务器的垃圾收集器，主要针对配备多核处理器和大容量内存的集器

同时支持新生代和老年代的垃圾收集

实现可预期及可配置的STW时间

最终替换CMS

### 优点/特点

并行与并发：G1能充分利用多CPU、多核环境的优势来缩短STW停顿的时间。部分其他收集器原本需要停顿java线程执行的GC动作，G1仍然可以使用并发的方式让程序继续运行。

分代收集：G1收集器可以独立管理整个Java堆，但是依然保留了新生代和老年代的概念，可以采取不同的方式去处理新创建的对象和存活时间较长的对象

空间整合（采用标记整理算法，不会产生内存碎片）；

可预测停顿：不但追求较低的STW时间，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间段内GC消耗时间不得超过N毫秒（几乎是实时java的垃圾收集器特征了）。

### 基本思想

**堆划分成Region**：将Java堆划分成很多个大小相等的独立区域（Region）。每个Region的大小可以是1~32M（必须是2的整数幂），默认是2M。Region是内存分配和回收的基本单位。此时新生代和老年代不再是物理隔离的内存空间，而分别是一部分（可以不连续的）Region的集合。（并不是所有的Region都已经拥有了老年代或者新生代的身份，可能会有不少Region处于没有身份的未分配状态，按需使用）Region的身份可能会在虚拟机运行时发生变化（原本是新生代，后来回收后分配给老年代使用）。Region的存在使得老年代被拆分成很多块，每次Mixed GC可以只回收垃圾最多的部分老年代Region（这也是G1的名称Garbage First的由来）。



**巨型对象**：如果有对象的size超过了单个Region 50%的大小，则该为对象分配单独的一个或几个连续Region（哪怕最后一个Region没有完全使用也可以），安放在巨型对象区（Humongous Region，H区，H区看作特殊的Old区，算作老年代）。

**Remembered Set（RSet）**：虽然堆被划分成Region，用作GC的最小单位，但是在管理上Region不是独立的，每一个Region中的对象都可以和其他任何一个Region中的对象发生引用关系。因此在对某个Region做可达性判断的时候应该需要遍历整个Java堆（这个并非是G1的独有问题，只是在G1中很突出。在其他的收集器中，为了对某个新生代做可达性判断，可能需要遍历整个老年代，导致Minor GC效率降低）。

为了避免全堆扫描，Region之间的对象引用使用了Remembered Set（其他收集器中，新生代和老年代之间的引用也使用了类似的机制）。每个Region持有一个Remembered Set，用于记录其他Region对本Region的引用。

虚拟机发现在对Reference类型的数据进行写操作的时候会产生一个Write Barrier暂时中断，检查Reference引用的对象是否处于不同的Region（在别的收集器中是检查是否老年代引用了新生代对象），如果是，就把相关引用信息加入到Remembered Set之中。当进行内存回收的时候，在根节点枚举的范围内加入Remembered Set就可以保证不对全堆扫描也不会产生遗漏。

Remembered Set作用概括：G1中存放了所有的跨Region引用，使得Young GC和Mixed GC的时候不需要全堆扫描，只需要扫描Region内部引用和Remembered Set即可。在其他收集器中存放了老年代对新生代的引用，在Young GC的时候只需要扫描新生代和Remembered Set。综上，省去了可达性判定的全堆扫描。

**Card**：先将老年代空间逻辑划分为大小相等的区域（Card，521Byte），同时维护一张Card的索引表。由于每个Region是整数倍的M，所以一个Region可以被完整的划分为多个card。

如果老年代当中的引用关系发生变化，比如GC Roots对老年代的引用变化，老年代内部对象之间的引用关系的变化，新生代To区对象晋升到老年代，都会将对应对象所在的Card在卡表中标记为Dirty Card（脏区，意思该Card内部发生了变化，哪种变化不知道）

Card Table（卡表）：是Card的一个索引，用来寻找脏区。

一个Region被划分成了多个Card。如果Region A中的某一个Card中的对象，引用了另一个Region B中的一个对象，那么就在B的RSet中记录A的那个Card的位置信息。当B被回收的时候，根据B的RSet中的信息，就可以知道A中有一个Card引用了它，这个时候就再去扫描A中的那个Card即可。

如果一个Card中的对象创建了一个对别人的引用，那么此时就在Card Table中把该Card标记为Dirty（发出引用的Card是脏的，不是被引用的）

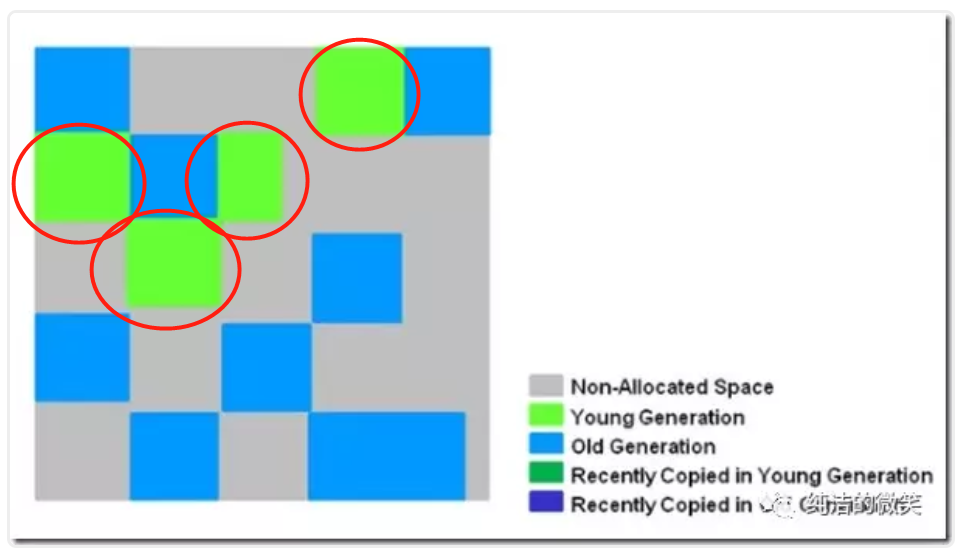
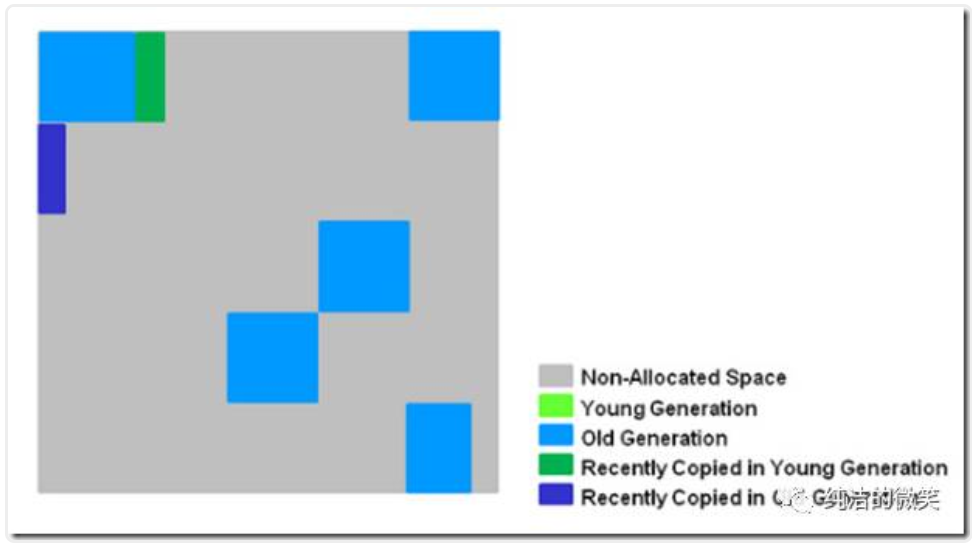
### GC的触发和处理

**Young GC**：新生代空间（Eden区）不足触发。

对新生代所在的Region进行GC，采用复制算法（一个Region中的部分数据被回收，和其他的Region中还存活的数据整合复制到新的Region去，原Region回收。所需的新Region可能不止一个，尽量连续分配）。此时Eden区和From Survivor区消失，为了复制新分配出来的地方现在是即将变成From区的To Survivor区。

给Eden和Survivor区分配新的Region

需要STW

**Mixed GC**：老年代空间使用达到阈值

G1收集器的最主要区别。此时回收新生代和部分老年代（对能产生最大效益的一些老年代Region进行回收，比如某个Region内部的对象全部失效，那么该Region的回收收益就很高）。控制对老年代回收哪些Region，回收多少个Region（剩下的这次不回收），就可以控制GC的开销，在极大概率上满足指定的GC运行时间。（为此需要对Region做回收价值排序）

**Full GC**（G1正常处理流程中没有Full GC）：老年代回收太慢或者新生代增长太快。具体的情况有疏散失败/转移失败（即新生代回收后没有分区接收存活的对象），巨型对象分配失败（空间不足或者没有足够的连续空间）

采用Serial Old GC（耗时很长，应该尽量避免，G1调优的重点就是尽量避免Full GC）。当出现Full GC时代表现有的参数配置出现了问题（Full GC被G1视作异常），可以增大新生代内存（防止新生代增长过快），增大堆内存（防止巨型对象分配失败）或者调用更多的线程进行GC（防止老年代回收过慢）

### Mixed GC过程步骤

初始标记（Initial Mark）：并行的标记GC Roots能直接关联到的对象。同时修改TAMS（Next Top at Mark Start）的值，让下一个阶段并发标记时能在正确的可用Region中创建对象。此过程需要STW。

当老年代使用率达到阈值触发Mixed GC时并不会启动初始标记，而是等到下一次Young GC的时候，利用Young GC的STW时间完成初始标记。这种方式称为借道（Piggy backing）

根区域扫描（Root Region Scan）：在上一阶段的STW之后（此时已经经历了一轮新的Young GC），把GC Roots直接引用到的对象视作根（Root，有别于GC Roots），根所在的Region（也就是Survivor区所使用的Region）视作根分区。（此时的根分区就是GC Roots直接引用的Region）。

获得根区域之后，扫描所有的Old区Region，把所有是根区域或者被根区引用的Region做标记，这些做标记的Region就是和GC Roots确定存在了间接引用的Region。下一步并发标记的时候就避免了全Old区扫描。

此阶段不能发生可能产生新生代收集的阶段（新生代收集可能会擦除当前部分Region，把里面的存活对象转移位置）。因此不能放在后续的并发标记过程中（虽然执行的任务并发标记也能完成，但是并发标记的过程种可能会出现不定次数的Young GC，所以放在并发标记之前）。

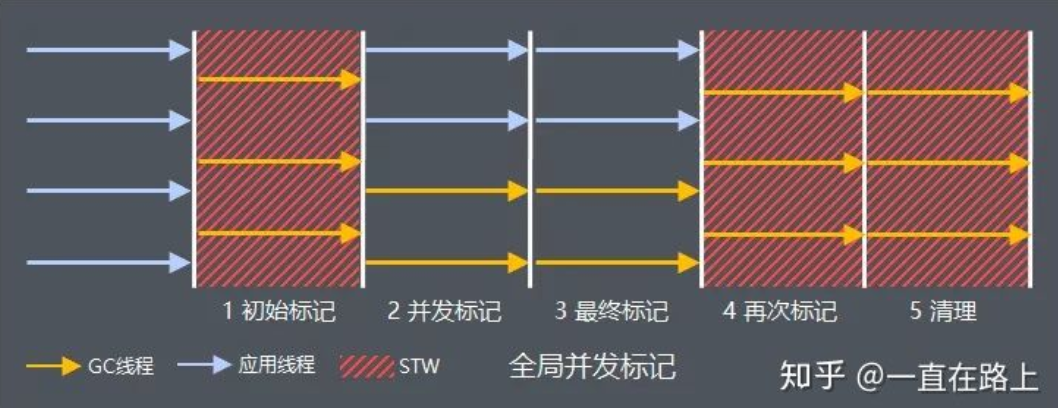
并发标记（Concurrent Mark）：多个标记线程并行，与用户线程之间可并发（当然也可并行）。这段时间耗时较长，在这个阶段如果产生了老年代对新生代的引用，会写入到RSet Log。

重新标记/最终标记（Remark）：重新标记阶段会STW，同时把RSet Log中的内容写入到RSet，然后去确定所有的引用变化情况。**可以并行执行**。和CMS中类似，这里也使用了卡表去寻找Dirty Card。（使用了SATB，比CMS速度更快）

SATB：Snapshot At The Beginning

在标记的初始阶段进行快照

清理（Cleanup）：对各个老年代Region按照回收价值和成本进行排序，在用户期望的GC开支内指定回收计划（可能只能回收部分的Region，导致回收不彻底，仍然有垃圾存在）。可以与用户程序并发执行，但是如果暂停用户程序STW会大幅提升收集效率。



### 应用场景与对应的垃圾收集算法

新生代：类似复制（但是没有保留一般的空间由于复制，所以可能会产生疏散失败的问题）

老年代：不完全的标记整理（对没有发生改变的老年代Region不移动，整理发生了变化的Region）

## GC收集器之间的应用范围和搭配

### 新生代收集器

Serial收集器，ParNew收集器，Parallel Scavenge收集器（都只能Young GC）

### 老年代收集器

CMS收集器（单纯老年代GC）

Serial Old收集器（老年代+方法区，可能用来在特殊时候完成Full GC的老年代和方法区GC）

Parallel Old收集器（老年代+方法区）

### 混合收集器

G1收集器（既能Young GC，也能Mixed GC）

### 搭配方式

CMS可以搭配Serial收集器，但是由于设计上的原因，不能和Parallel Scavenge收集器搭配，为了解决Serial收集器的单线程性能低问题，开发了ParNew收集器，可以和CMS搭配。

图中？是G1收集器，ZGC收集器和Shenandoah收集器。这三种都是跨代收集器。

ZGC收集器：Java11开始使用，使用了染色指针，STW非常短暂，但是带来了内存消耗的增加，适合超大堆。

Shenandoah收集器，RedHat开发

补充：当CMS老年代收集失败触发了Full GC，以及G1收集器使用中触发了Full GC时，使用Serial Old（这两种情况都是GC调优的主要针对对象）

