**Java程序内存模型**

**薪薪乐组技术分享-JellyLiu 刘骥**

目录

[一 JVM 内存区域 3](#_Toc51687719)

[1、虚拟机栈 3](#_Toc51687720)

[1.1 栈内存大小设置 4](#_Toc51687721)

[1.2虚拟机栈异常类型 6](#_Toc51687722)

[2 本地方法栈 7](#_Toc51687723)

[3 程序计数器 7](#_Toc51687724)

[3.1程序计数器的定义 7](#_Toc51687725)

[3.2程序计数器的作用 7](#_Toc51687726)

[3.3程序计数器的特点 8](#_Toc51687727)

[4.直接内存（堆外内存） 8](#_Toc51687728)

[**4.1操作直接内存** 8](#_Toc51687729)

[**4.2直接内存与堆内存比较** 8](#_Toc51687730)

[5 Java堆 8](#_Toc51687731)

[5.1默认配置 9](#_Toc51687732)

[5.2 Jvm配置参数 9](#_Toc51687733)

[二、Java内存模型Java Memory Model（JMM） 10](#_Toc51687734)

[2.1 Java内存模型定义 10](#_Toc51687735)

[2.2 happens-before 11](#_Toc51687736)

[三、Java对象模型 12](#_Toc51687737)

[3.1 oop-klass model 12](#_Toc51687738)

[（1）oop体系 12](#_Toc51687739)

[（2） klass 14](#_Toc51687740)

[（3）JVM实现 16](#_Toc51687741)

[四 垃圾回收 17](#_Toc51687742)

[1、如何识别垃圾 17](#_Toc51687743)

[1.1引用计数法 17](#_Toc51687744)

[1.2可达性算法 18](#_Toc51687745)

[(1)虚拟机栈中引用的对象 19](#_Toc51687746)

[(2)方法区中类静态属性引用的对象 19](#_Toc51687747)

[(3) 方法区中常量引用的对象 19](#_Toc51687748)

[(4)本地方法栈中 JNI 引用的对象 20](#_Toc51687749)

[2垃圾回收主要方法 20](#_Toc51687750)

[2.1标记清除算法 20](#_Toc51687751)

[2.2复制算法 20](#_Toc51687752)

[2.3标记整理法 21](#_Toc51687753)

[2.4分代收集算法 22](#_Toc51687754)

[3垃圾收集器种类 （自行阅读） 25](#_Toc51687755)

[3.1新生代收集器 26](#_Toc51687756)

[（1）Serial 收集器 26](#_Toc51687757)

[（2）ParNew 收集器 27](#_Toc51687758)

[（3）Parallel Scavenge 收集器 28](#_Toc51687759)

[3.2老年代收集器 28](#_Toc51687760)

[（1）Serial Old 收集器 28](#_Toc51687761)

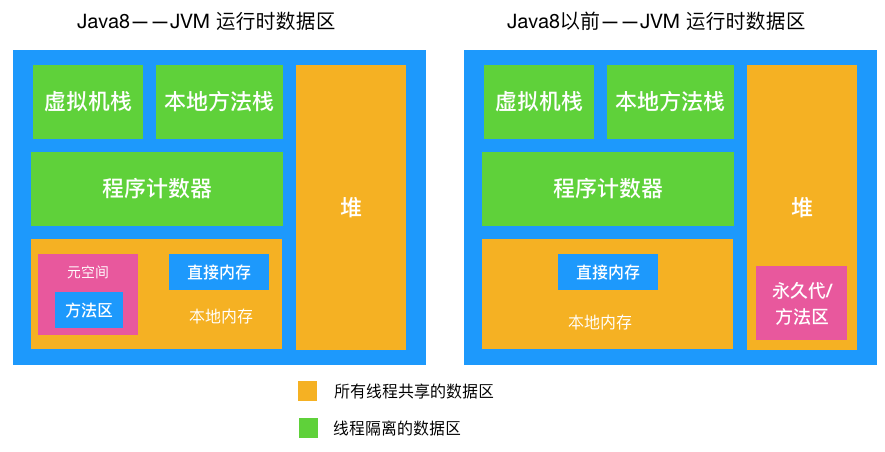
[（2）Parallel Old 收集器 29](#_Toc51687762)

[（3）CMS 收集器 29](#_Toc51687763)

[（4）G1（Garbage First） 收集器 30](#_Toc51687764)

[3.4垃圾回收总结 33](#_Toc51687765)

# 一 JVM 内存区域



（1）jdk1.7以及之前：方法区位于永久代(PermGen)，永久代和堆相互隔离，永久代的大小在启动JVM时可以设置一个固定值，不可变；

（2）元数据区（方法区的实现）： Java8中用元空间替代了永久代，元空间并不在虚拟机中，而是使用本地内存，并且大小可以是自动增长的，这样减少了OOM的可能性。元空间存储JIT即时编译后的native代码。

（3）永久代中的 class metadata 转移到了 native memory（本地内存，而不是虚拟机）；class metadata包括**类的版本、字段、方法、接口**等描述信息。

永久代中的 interned Strings 和 class static variables 转移到了 Java heap。

( 4）永久代参数 （PermSize MaxPermSize） -> 元空间参数（MetaspaceSize MaxMetaspaceSize）。

## 1、虚拟机栈



### 1.1 栈内存大小设置

栈内存为线程私有的空间，每个线程都会创建私有的栈内存。

栈空间内存设置过大，创建线程数量较多时会出现栈内存溢出StackOverflowError。

同时，栈内存也决定方法调用的深度，栈内存过小则会导致方法调用的深度较小，如递归调用的次数较少。

每个线程都有独立的栈空间。

**递归调用**

public class Stack {

private static int count = 0;

public static void recursion(long a, long b, long c) {

count++;

recursion(a, b, c);

}

public static void main(String args[]) {

try {

recursion(0L, 0L, 0L);

} catch (Throwable e) {

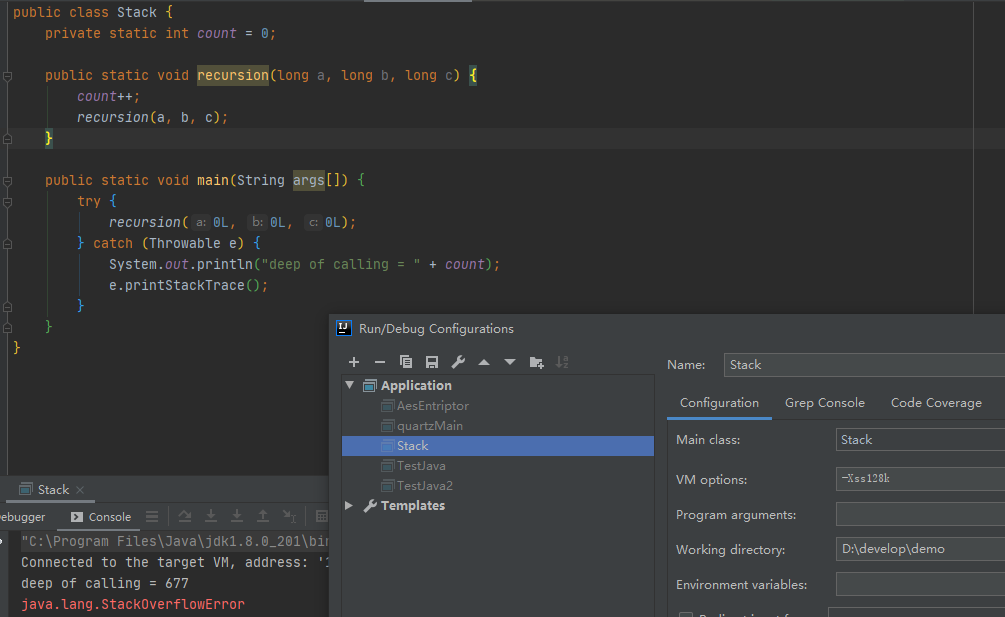
System.out.println("deep of calling = " + count);

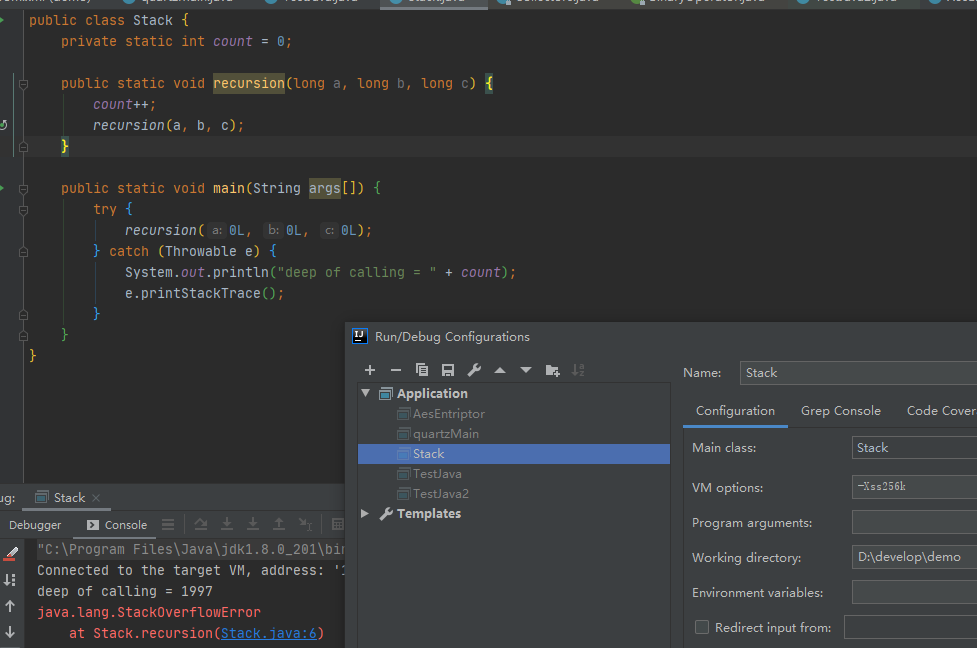
e.printStackTrace();

}

}

}





-Xss128k: deep of calling = 677

-Xss256k：deep of calling = 1997

### 1.2虚拟机栈异常类型

Java 虚拟机栈会出现两种异常：StackOverFlowError 和 OutOfMemoryError。

* StackOverFlowError 若 Java 虚拟机栈的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时，抛出 StackOverFlowError 异常。
* OutOfMemoryError 若允许动态扩展，那么当线程请求栈时内存用完了，无法再动态扩展时，抛出 OutOfMemoryError 异常。

出现 StackOverFlowError 时，内存空间可能还有很多。

## 2 本地方法栈

* 本地方法栈是为 JVM 运行 Native 方法准备的空间，由于很多 Native 方法都是用 C 语言实现的，所以它通常又叫 C 栈。它与 Java 虚拟机栈实现的功能类似，只不过本地方法栈是描述本地方法运行过程的内存模型。

## 3 程序计数器

### 3.1程序计数器的定义

程序计数器是一块较小的内存空间，是当前线程正在执行的那条字节码指令的地址。若当前线程正在执行的是一个本地方法，那么此时程序计数器为Undefined。

### 3.2程序计数器的作用

* 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。
* 在多线程情况下，程序计数器记录的是当前线程执行的位置，从而当线程切换回来时，就知道上次线程执行到哪了。

### 3.3程序计数器的特点

* 是一块较小的内存空间。
* 线程私有，每条线程都有自己的程序计数器。
* 生命周期：随着线程的创建而创建，随着线程的结束而销毁。
* 是唯一一个不会出现OutOfMemoryError的内存区域。

## 4.直接内存（堆外内存）

直接内存是除 Java 虚拟机之外的内存，但也可能被 Java 使用。

**4.1操作直接内存**

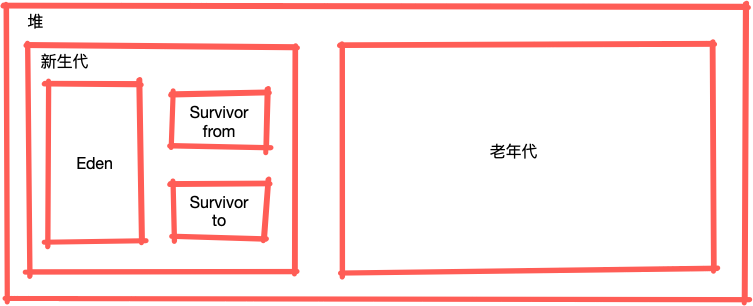
在 NIO 中引入了一种基于通道和缓冲的 IO 方式。它可以通过调用本地方法直接分配 Java 虚拟机之外的内存，然后通过一个存储在堆中的DirectByteBuffer对象直接操作该内存，而无须先将外部内存中的数据复制到堆中再进行操作，从而提高了数据操作的效率。

直接内存的大小不受 Java 虚拟机控制，但既然是内存，当内存不足时就会抛出 OutOfMemoryError 异常。

**4.2直接内存与堆内存比较**

* 直接内存申请空间耗费更高的性能
* 直接内存读取 IO 的性能要优于普通的堆内存。
* 直接内存作用链： 本地 IO -> 直接内存 -> 本地 IO
* 堆内存作用链：本地 IO -> 直接内存 -> 非直接内存 -> 直接内存 -> 本地 IO

## 5 Java堆



### 5.1默认配置

年轻代：老年代 = 1：2

eden区：survivor0 ： survivor1 = 8：1：1

### 5.2 Jvm配置参数

* -Xms：表示java虚拟机堆区内存初始内存分配的大小；
* -Xmx：表示java虚拟机堆区内存可被分配的最大上限
* -XX:newSize：表示新生代初始内存的大小，应该小于-Xms的值，128的倍数；
* -XX:MaxnewSize：表示新生代可被分配的内存的最大上限；当然这个值应该小于-Xmx的值，因为新生代占内存来自整个堆内存。为了优化GC(内存垃圾回收)，最好设置-XX:MaxnewSize值约等于-Xmx的1/3，值为128的倍数；
* -Xmn：这个参数则是对 -XX:newSize、-XX:MaxnewSize两个参数的同时配置，也就是说如果通过-Xmn来配置新生代的内存大小，那么-XX:newSize = -XX:MaxnewSize = -Xmn

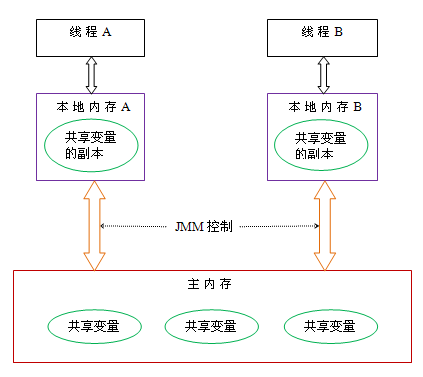
**通常会将** **-Xms** **与** **-Xmx两个参数的配置相同的值，其目的是为了能够在java垃圾回收清理完堆区后不需要重新分隔计算堆区的大小而浪费资源。**

# 二、[Java内存模型](https://mp.weixin.qq.com/s/ME_rVwhstQ7FGLPVcfpugQ)Java Memory Model（JMM）

## 2.1 Java内存模型定义

定义：JMM是一种规范，目的是解决由于多线程通过共享内存进行通信时，存在的本地内存数据不一致、编译器会对代码指令重排序、处理器会对代码乱序执行等带来的问题。目的是保证并发编程场景中的原子性、可见性和有序性. 其实JMM并不像JVM内存结构一样是真实存在的。他只是一个抽象的概念。

实现：volatile、synchronized、final、concurrent包等。其实这些就是Java内存模型封装了底层的实现后提供给程序员使用的一些关键字

[](http://www.hollischuang.com/wp-content/uploads/2018/06/11.png)

主内存：所有变量都保存在主内存中  
工作内存：每个线程的独立内存，保存了该线程使用到的变量的主内存副本拷贝，线程对变量的操作必须在工作内存中进行

每个线程都有自己的本地内存共享副本，如果A线程要更新主内存还要让B线程获取更新后的变量，那么需要：

1. 将本地内存A中更新共享变量
2. 将更新的共享变量刷新到主内存中
3. 线程B从主内存更新最新的共享变量

## 2.2 [happens-before](https://www.cnblogs.com/chenssy/p/6393321.html)

我们无法就所有场景来规定某个线程修改的变量何时对其他线程可见，但是我们可以指定某些规则，这规则就是happens-before。特别关注在多线程之间的内存可见性。

它是判断数据是否存在竞争、线程是否安全的主要依据，依靠这个原则，我们解决在并发环境下两操作之间是否可能存在冲突的所有问题。

1. 程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作；
2. 锁定规则：一个unLock操作先行发生于后面对同一个锁额lock操作；
3. volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作；
4. 传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C；
5. 线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作；
6. 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生；
7. 线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行；
8. 对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始；

# 三、[Java对象模型](https://www.cnblogs.com/qingshanli/p/9250491.html)

在内存中，一个Java对象包含三部分：对象头、实例数据和对齐填充（HotSpot要求对象起始地址必须是8字节的整数倍，比如byte 1字节 ，short 2字节,int 4字节）。而对象头中又包含锁状态标志、线程持有的锁等标志。

## ****3.1 oop-klass model****

OOP（Ordinary Object Pointer）指的是普通对象指针，而Klass用来描述对象实例的具体类型。

### （1）oop体系

[复制代码](javascript:void(0);)

//定义了oops共同基类

typedef class oopDesc\* oop;

//表示一个Java类型实例

typedef class instanceOopDesc\* instanceOop;

//表示一个Java方法

typedef class methodOopDesc\* methodOop;

//表示一个Java方法中的不变信息

typedef class constMethodOopDesc\* constMethodOop;

//记录性能信息的数据结构

typedef class methodDataOopDesc\* methodDataOop;

//定义了数组OOPS的抽象基类

typedef class arrayOopDesc\* arrayOop;

//表示持有一个OOPS数组

typedef class objArrayOopDesc\* objArrayOop;

//表示容纳基本类型的数组

typedef class typeArrayOopDesc\* typeArrayOop;

//表示在Class文件中描述的常量池

typedef class constantPoolOopDesc\* constantPoolOop;

//常量池告诉缓存

typedef class constantPoolCacheOopDesc\* constantPoolCacheOop;

//描述一个与Java类对等的C++类

typedef class klassOopDesc\* klassOop;

//表示对象头

typedef class markOopDesc\* markOop;

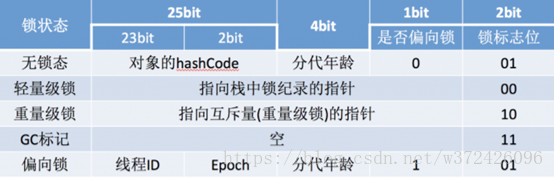
[复制代码](javascript:void(0);)

如上面代码所示, oops模块包含多个子模块, 每个子模块对应一个类型, 每一个类型的oop都代表一个在JVM内部使用的特定对象的类型。其中有一个变量oop的类型oopDesc是oops模块的共同基类型。而oopDesc类型又包含instanceOopDesc (类实例)、arrayOopDesc (数组)等子类类型。其中instanceOopDesc 中主要包含以下几部分数据：markOop \_mark和union \_metadata 以及一些不同类型的 field。

在java程序运行过程中, 每创建一个新的java对象, 在JVM内部就会相应的创建一个对应类型的oop对象来表示该java对象。而在HotSpot虚拟机中, 对象在内存中包含三块区域: 对象头、实例数据和对齐填充。其中对象头包含两部分内容：\_mark和\_metadata，而实例数据则保存在oopDesc中定义的各种field中。

\_mark:

\_mark这一部分用于存储对象自身的运行时数据, 如哈希码、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等, 这部分数据的长度在32位和64位的虚拟机(未开启压缩指针)中分别为32bit和64bit, 官方称它为 "Mark Word"。对象需要存储的运行时数据很多, 其实已经超出了32位和64位Bitmap结构所能记录的限度, 但是对象头信息是与对象自身定义的数据无关的额外存储成本, 考虑到虚拟机的空间效率, Mark Word被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息, 它会根据对象的状态复用自己的存储空间。



\_metadata:

\_metadata这一部分是类型指针, 即对象指向它的类元数据的指针, 虚拟机通过这个指针来确定这个对象是哪个类的实例。并不是所有的虚拟机实现都必须在对象数据上保留类型指针, 换句话说查找对象的元数据信息并不一定要经过对象本身, 其取决于虚拟机实现的对象访问方式。目前主流的访问方式有使用句柄和直接指针两种, 两者方式的不同这里先暂不做介绍。另外, 如果对象是一个Java数组, 那么在对象头中还必须有一块用于记录数组长度的数据, 因为虚拟机可以通过普通java对象的元数据信息确定java对象的大小, 但是从数组的元数据中却无法确定数组的大小。

### ****（2） klass****

Klass体系:

[复制代码](javascript:void(0);)

//klassOop的一部分，用来描述语言层的类型

class Klass;

//在虚拟机层面描述一个Java类

class instanceKlass;

//专有instantKlass，表示java.lang.Class的Klass

class instanceMirrorKlass;

//专有instantKlass，表示java.lang.ref.Reference的子类的Klass

class instanceRefKlass;

//表示methodOop的Klass

class methodKlass;

//表示constMethodOop的Klass

class constMethodKlass;

//表示methodDataOop的Klass

class methodDataKlass;

//作为klass链的端点，klassKlass的Klass就是它自身

class klassKlass;

//表示instanceKlass的Klass

class instanceKlassKlass;

//表示arrayKlass的Klass

class arrayKlassKlass;

//表示objArrayKlass的Klass

class objArrayKlassKlass;

//表示typeArrayKlass的Klass

class typeArrayKlassKlass;

//表示array类型的抽象基类

class arrayKlass;

//表示objArrayOop的Klass

class objArrayKlass;

//表示typeArrayOop的Klass

class typeArrayKlass;

//表示constantPoolOop的Klass

class constantPoolKlass;

//表示constantPoolCacheOop的Klass

class constantPoolCacheKlass;

[复制代码](javascript:void(0);)

和oopDesc是其他oop类型的父类一样，Klass类是其他klass类型的父类。

Klass向JVM提供两个功能:

* 实现语言层面的Java类（在Klass基类中已经实现）
* 实现Java对象的分发功能（由Klass的子类提供虚函数实现）

HotSpot JVM的设计者因为不想让每一个对象中都含有一个虚函数表, 所以设计了oop-klass模型, 将对象一分为二, 分为klass和oop。其中oop主要用于表示对象的实例数据, 所以不含有任何虚函数。而klass为了实现虚函数多态, 所以提供了虚函数表。所以，关于Java的多态，其实也有c++虚函数的影子在。

### （3）JVM实现

在Java中，JVM中的对象模型包含两部分：Oop和Klass，在类被加载的时候，JVM会给类创建一个instanceKlass，其中包含了类信息、常量、静态变量、即时编译器编译后的代码等，存储在方法区，用来在JVM层表示该Java类。而使用new一个对象后，JVM就会创建一个instanceOopDesc对象，该对象包含对象头和实例数据，对象头中保存的是锁的状态标志等信息，元数据则实际上是一个指针，指向instanceKlass。

我们来看看下面这段代码的存储结构。

[复制代码](javascript:void(0);)

class Model

{

public static int a = 1;

public int b;

public Model(int b) {

this.b = b;

}

}

public static void main(String[] args) {

int c = 10;

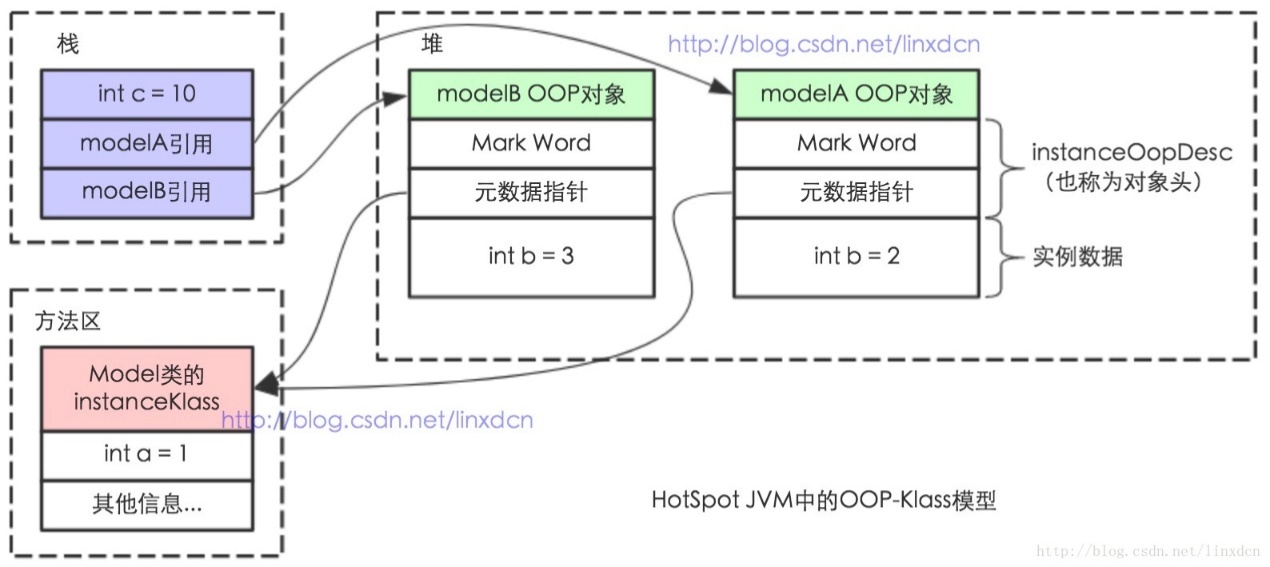
Model modelA = new Model(2);

Model modelB = new Model(3);

}

[复制代码](javascript:void(0);)

存储结构如下:



对象的实例（instantOopDesc)保存在堆上，

对象的元数据（instantKlass）保存在方法区，

对象的引用保存在栈上。

# 四 垃圾回收

## 1、如何识别垃圾

### 1.1引用计数法

最容易想到的一种方式是引用计数法，啥叫引用计数法，简单地说，就是对象被引用一次，在它的对象头上加一次引用次数，如果没有被引用（引用次数为 0），则此对象可回收。

**public class** **TestRC** {  
  
    TestRC instance;  
 **public** **TestRC**(String name) {  
    }  
  
 **public** **static** **void** **main**(String[] args) {  
 *// 第一步*  
 A a = **new** TestRC("a");  
 B b = **new** TestRC("b");  
  
 *// 第二步*  
 a.instance = b;  
 b.instance = a;  
  
 *// 第三步*  
 a = **null**;  
 b = **null**;  
 }  
}

虽然 a，b 都被置为 null 了，但是由于之前它们指向的对象互相指向了对方（引用计数都为 1），所以无法回收，也正是由于无法解决循环引用的问题，所以现代虚拟机都不用引用计数法来判断对象是否应该被回收。

### 1.2可达性算法

现代虚拟机基本都是采用这种算法来判断对象是否存活，可达性算法的原理是以一系列叫做  **GC Root**  的对象为起点出发，引出它们指向的下一个节点，再以下个节点为起点，引出此节点指向的下一个结点。。。（这样通过 GC Root 串成的一条线就叫引用链），直到所有的结点都遍历完毕,如果相关对象不在任意一个以 **GC Root** 为起点的引用链中，则这些对象会被判断为「垃圾」,会被 GC 回收。

从GC Roots搜索，那么这些 GC Roots 到底是什么东西呢，哪些对象可以作为 GC Root 呢，有以下几类

### (1)虚拟机栈中引用的对象

**publicclass** **Test** {  
 **public** **static** **void** **main**(String[] args) {  
 Test a = **new** Test();  
 a = **null**;  
 }  
}

### (2)方法区中类静态属性引用的对象

**public class** **Test** {  
**public static** Test s;  
 **public** **static** **void** **main**(String[] args) {  
 Test a = **new** Test();  
 a.s = **new** Test();  
 a = **null**;  
 }  
}

### (3) 方法区中常量引用的对象

如下代码所示，常量 s 指向的对象并不会因为 a 指向的对象被回收而回收

**public class** **Test** {  
 **public static final** Test s = **new** Test();  
     **public** **static** **void** **main**(String[] args) {  
     Test a = **new** Test();  
     a = **null**;  
     }  
}

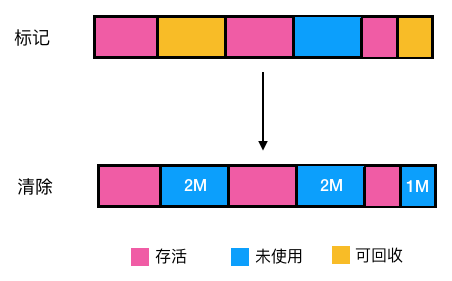
### (4)本地方法栈中 JNI 引用的对象

会在此本地方法执行完成后才会被释放。

## 2垃圾回收主要方法

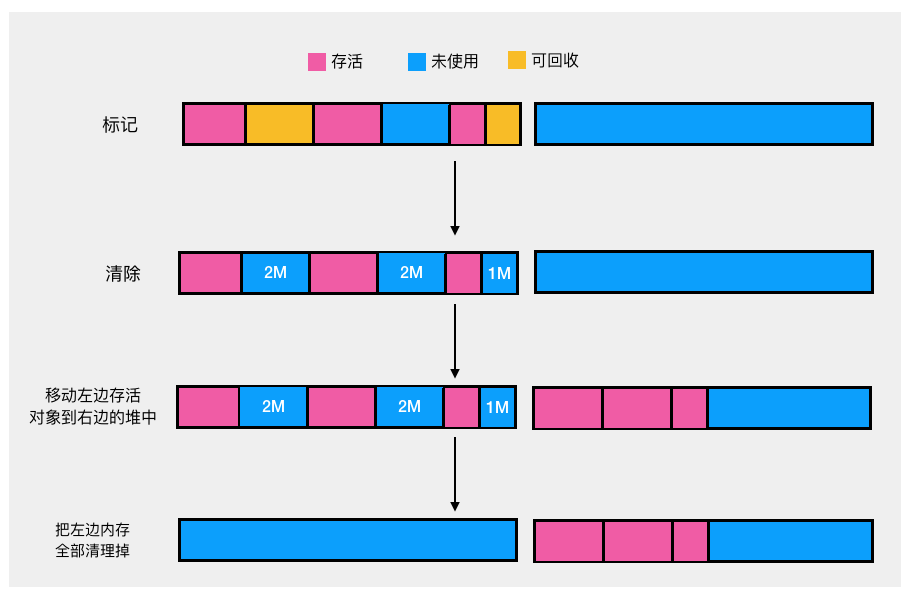
### 2.1标记清除算法

步骤很简单

1. 先根据可达性算法**标记**出相应的可回收对象（图中黄色部分）
2. 对可回收的对象进行回收
3. 操作起来确实很简单，也不用做移动数据的操作，那有啥问题呢？仔细看上图，没错，内存碎片！假如我们想在上图中的堆中分配一块需要**连续内存**占用 4M 或 5M 的区域，显然是会失败，怎么解决呢，如果能把上面未使用的 2M， 2M，1M 内存能连起来就能连成一片可用空间为 5M 的区域即可，怎么做呢?

### 2.2复制算法

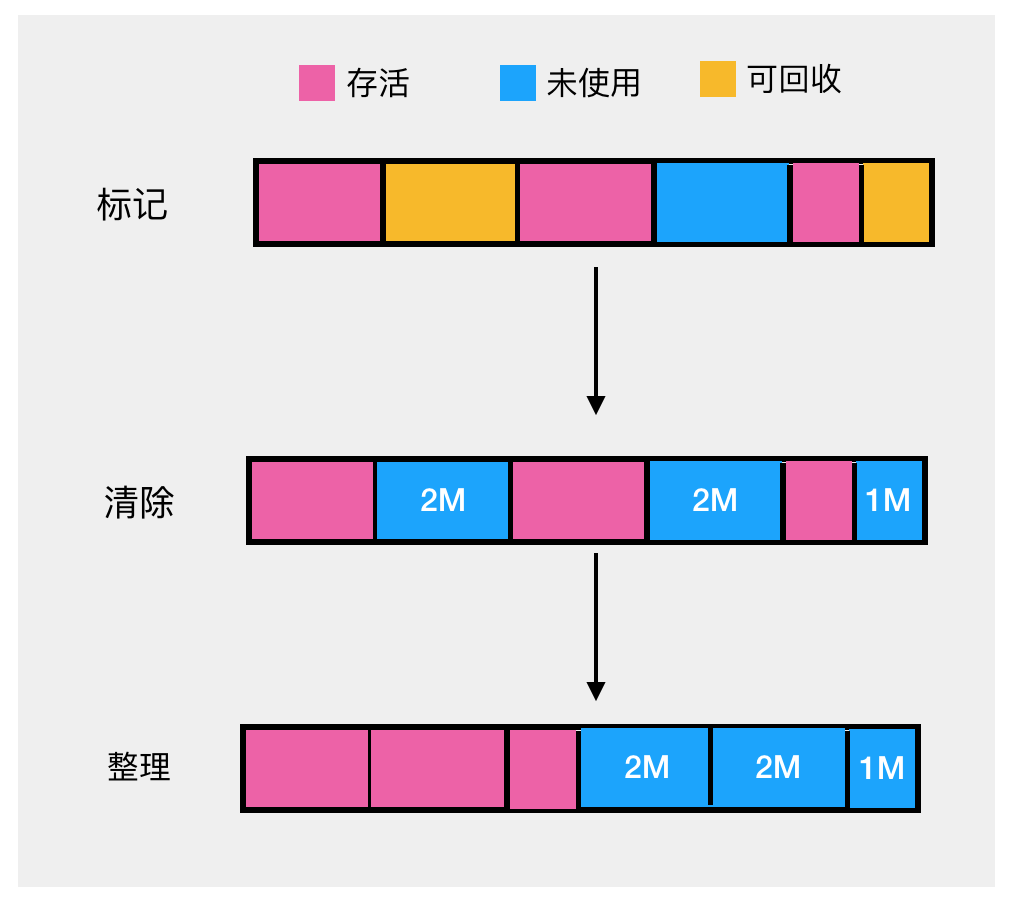
把堆等分成两块区域, A 和 B，区域 A 负责分配对象，区域 B 不分配, 对区域 A 使用以上所说的标记法把存活的对象标记出来（下图有误无需清除），然后把区域 A 中存活的对象都复制到区域 B（存活对象都依次**紧邻排列**）最后把 A 区对象全部清理掉释放出空间，这样就解决了内存碎片的问题了。



不过复制算法的缺点很明显，比如给堆分配了 500M 内存，结果只有 250M 可用，空间平白无故减少了一半！这肯定是不能接受的！另外每次回收也要把存活对象移动到另一半，效率低下（我们可以想想删除数组元素再把非删除的元素往一端移，效率显然堪忧）

### 2.3标记整理法

前面两步和标记清除法一样，不同的是它在标记清除法的基础上添加了一个整理的过程 ，即将所有的存活对象都往一端移动,紧邻排列（如图示），再清理掉另一端的所有区域，这样的话就解决了内存碎片的问题。



但是缺点也很明显：每进一次垃圾清除都要频繁地移动存活的对象，效率十分低下。

### 2.4分代收集算法

#### 2.4.1 分代收集原因

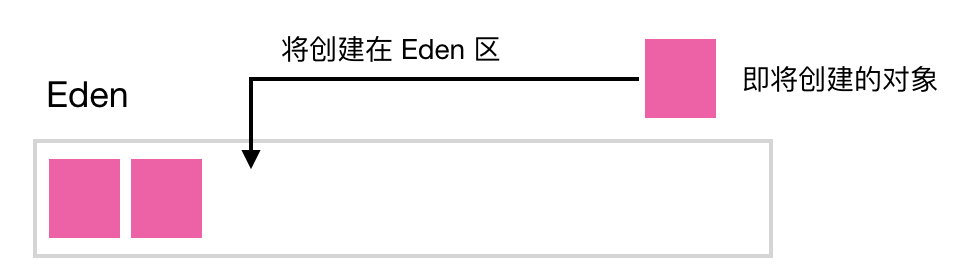
分代收集算法整合了以上算法，综合了这些算法的优点，最大程度避免了它们的缺点，所以是现代虚拟机采用的首选算法,与其说它是算法，倒不是说它是一种策略，因为它是把上述几种算法整合在了一起，为啥需要分代收集呢，来看一下对象的分配有啥规律。

大部分的对象都很短命，都在很短的时间内都被回收了（IBM 专业研究表明，一般来说，98% 的对象都是朝生夕死的，经过一次 Minor GC 后就会被回收），所以分代收集算法根据**对象存活周期的不同**将堆分成新生代和老生代（Java8以前还有个永久代）,默认比例为 1 : 2，新生代又分为 Eden 区， from Survivor 区（简称S0），to Survivor 区(简称 S1),三者的比例为 8: 1 : 1，这样就可以根据新老生代的特点选择最合适的垃圾回收算法，我们把新生代发生的 GC 称为 Young GC（也叫 Minor GC）,老年代发生的 GC 称为 Old GC（也称为 Full GC）。

#### 2.4.2分代收集工作原理

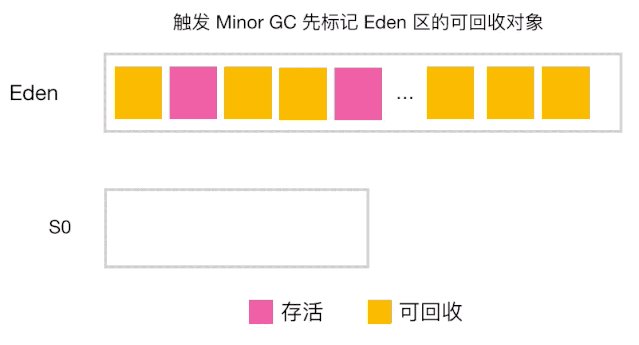
**（1）、对象在新生代的分配与回收**

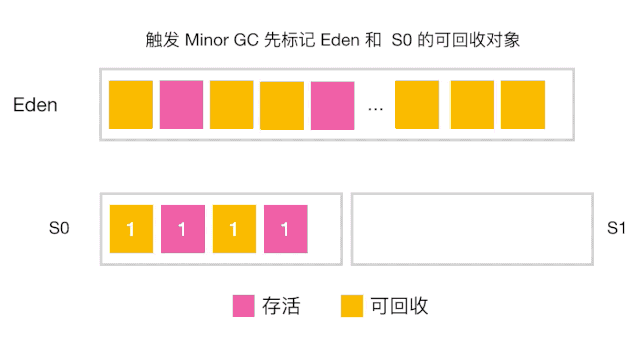
大部分对象在很短的时间内都会被回收，对象一般分配在 Eden 区



当 Eden 区将满时，触发 Minor GC

我们之前怎么说来着，大部分对象在短时间内都会被回收, 所以经过 Minor GC 后只有少部分对象会存活，它们会被移到 S0 区（这就是为啥空间大小  Eden: S0: S1 = 8:1:1, Eden 区远大于 S0,S1 的原因，因为在 Eden 区触发的 Minor GC 把大部对象（接近98%）都回收了,只留下少量存活的对象，此时把它们移到 S0 或 S1 绰绰有余）同时对象年龄加一（对象的年龄即发生 Minor GC 的次数），最后把 Eden 区对象全部清理以释放出空间,动图如下



当触发下一次 Minor GC 时，会把 Eden 区的存活对象和 S0（或S1） 中的存活对象（S0 或 S1 中的存活对象经过每次 Minor GC 都可能被回收）一起移到 S1（Eden 和 S0 的存活对象年龄+1）, 同时清空 Eden 和 S0 的空间。

若再触发下一次 Minor GC，则重复上一步，只不过此时变成了 从 Eden，S1 区将存活对象复制到 S0 区,每次垃圾回收, S0, S1 角色互换，都是从 Eden ,S0(或S1) 将存活对象移动到 S1(或S0)。也就是说在 Eden 区的垃圾回收我们采用的是**复制算法**，因为在 Eden 区分配的对象大部分在 Minor GC 后都消亡了，只剩下极少部分存活对象（这也是为啥 Eden:S0:S1 默认为 8:1:1 的原因），S0,S1 区域也比较小，所以最大限度地降低了复制算法造成的对象频繁拷贝带来的开销。

**（2）、对象何时晋升老年代**

* 当对象的年龄达到了我们设定的阈值，则会从S0（或S1）晋升到老年代
* 大对象 -XX:PretenureSizeThreshold 大于这个参数的对象直接在老年代分配
* 还有一种情况也会让对象晋升到老年代，即在 S0（或S1） 区相同年龄的对象大小之和大于 S0（或S1）空间一半以上时，则年龄大于等于该年龄的对象也会晋升到老年代。

**（3）、空间分配担保**

在发生 MinorGC 之前，虚拟机会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的总空间，如果大于，那么Minor GC 可以确保是安全的,如果不大于，那么虚拟机会查看 HandlePromotionFailure 设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用连续空间是否大于历次晋升到老年代对象的平均大小，如果大于则进行 Minor GC，否则可能进行一次 Full GC。

**（4）、Stop The World**

什么是 STW ？所谓的 STW, 即在 GC（minor GC 或 Full GC）期间，只有垃圾回收器线程在工作，其他工作线程则被挂起。

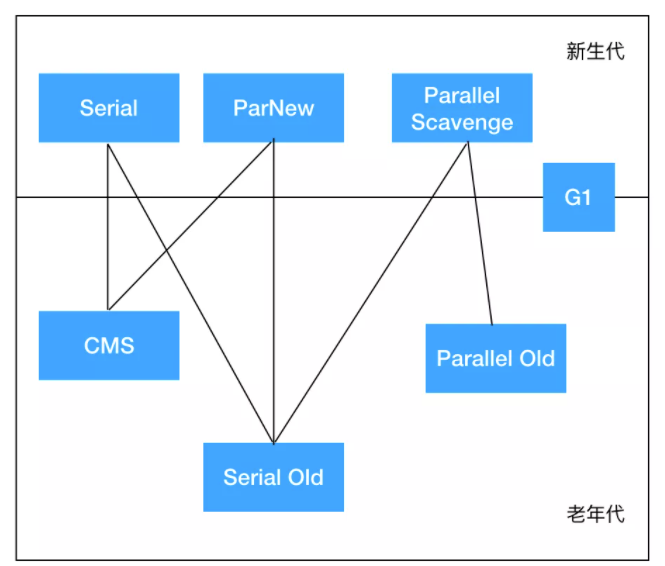
*画外音：为啥在垃圾收集期间其他工作线程会被挂起？想象一下，你一边在收垃圾，另外一群人一边丢垃圾，垃圾能收拾干净吗。*

Safe Point 主要指的是以下特定位置：

* 循环的末尾
* 方法返回前
* 调用方法的 call 之后
* 抛出异常的位置

## 3垃圾收集器种类 （自行阅读）

如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。Java 虚拟机规范并没有规定垃圾收集器应该如何实现，因此一般来说不同厂商，不同版本的虚拟机提供的垃圾收集器实现可能会有差别，一般会给出参数来让用户根据应用的特点来组合各个年代使用的收集器，主要有以下垃圾收集器



* 在新生代工作的垃圾回收器：Serial, ParNew, ParallelScavenge
* 在老年代工作的垃圾回收器：CMS，Serial Old, Parallel Old
* 同时在新老生代工作的垃圾回收器：G1

图片中的垃圾收集器如果存在连线，则代表它们之间可以配合使用，接下来我们来看看各个垃圾收集器的具体功能。

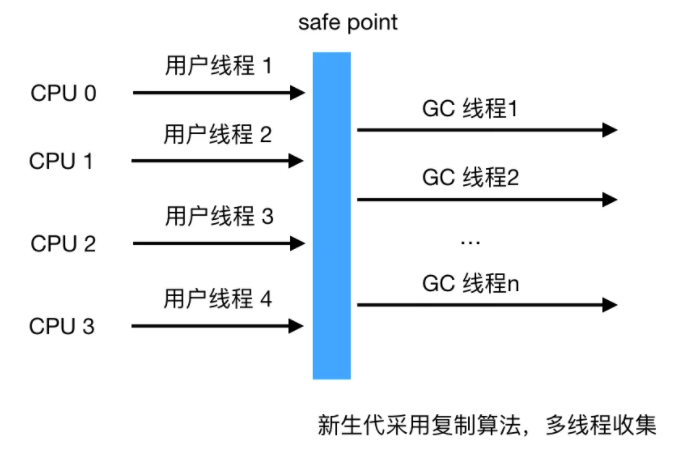
### 3.1新生代收集器

### （1）Serial 收集器

Serial 收集器是工作在新生代的，单线程的垃圾收集器，单线程意味着它只会使用一个 CPU 或一个收集线程来完成垃圾回收，在 **Client 模式**下，它简单有效（与其他收集器的单线程比），对于限定单个 CPU 的环境来说，Serial 单线程模式无需与其他线程交互，减少了开销，专心做 GC 能将其单线程的优势发挥到极致，另外在用户的桌面应用场景，分配给虚拟机的内存一般不会很大，收集几十甚至一两百兆（仅是新生代的内存，桌面应用基本不会再大了），STW 时间可以控制在一百多毫秒内，只要不是频繁发生，这点停顿是可以接受的，所以对于运行在 Client 模式下的虚拟机，Serial 收集器是新生代的默认收集器

### （2）ParNew 收集器

ParNew 收集器是 Serial 收集器的多线程版本，除了使用多线程，其他像收集算法,STW,对象分配规则，回收策略与 Serial 收集器完成一样，在底层上，这两种收集器也共用了相当多的代码，它的垃圾收集过程如下



ParNew 主要工作在 Server 模式，我们知道服务端如果接收的请求多了，响应时间就很重要了，多线程可以让垃圾回收得更快，也就是减少了 STW 时间，能提升响应时间，所以是许多运行在 Server 模式下的虚拟机的首选新生代收集器，另一个与性能无关的原因是因为除了 Serial  收集器，**只有它能与 CMS 收集器配合工作**，CMS 是一个划时代的垃圾收集器，是真正意义上的**并发收集器**，它第一次实现了垃圾收集线程与用户线程（基本上）同时工作，它采用的是传统的 GC 收集器代码框架，与 Serial,ParNew 共用一套代码框架，所以能与这两者一起配合工作，而后文提到的 Parallel Scavenge 与 G1 收集器没有使用传统的 GC 收集器代码框架，而是另起炉灶独立实现的，另外一些收集器则只是共用了部分的框架代码,所以无法与 CMS 收集器一起配合工作。

在多 CPU 的情况下，由于 ParNew 的多线程回收特性，毫无疑问垃圾收集会更快，也能有效地减少 STW 的时间，提升应用的响应速度。

### （3）Parallel Scavenge 收集器

Parallel Scavenge 收集器也是一个使用**复制算法**，**多线程**，工作于新生代的垃圾收集器，看起来功能和 ParNew 收集器一样，它有啥特别之处吗

**关注点不同**，CMS 等垃圾收集器关注的是尽可能缩短垃圾收集时用户线程的停顿时间，而 Parallel Scavenge 目标是达到一个可控制的吞吐量（吞吐量 = 运行用户代码时间 / （运行用户代码时间+垃圾收集时间）），也就是说 CMS 等垃圾收集器更适合用到与用户交互的程序，因为停顿时间越短，用户体验越好，而 Parallel Scavenge 收集器关注的是吞吐量，所以更适合做后台运算等不需要太多用户交互的任务。

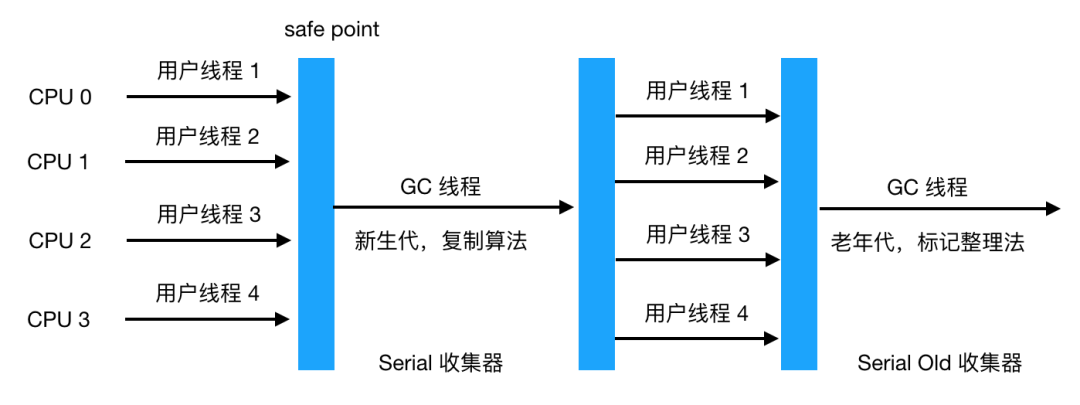
Parallel Scavenge 收集器提供了两个参数来精确控制吞吐量，分别是控制最大垃圾收集时间的 -XX:MaxGCPauseMillis 参数及直接设置吞吐量大小的 -XX:GCTimeRatio（默认99%）

除了以上两个参数，还可以用 Parallel Scavenge 收集器提供的第三个参数 -XX:UseAdaptiveSizePolicy，开启这个参数后，就不需要手工指定新生代大小,Eden 与 Survivor 比例（SurvivorRatio）等细节，只需要设置好基本的堆大小（-Xmx 设置最大堆）,以及最大垃圾收集时间与吞吐量大小，虚拟机就会根据当前系统运行情况收集监控信息，动态调整这些参数以尽可能地达到我们设定的最大垃圾收集时间或吞吐量大小这两个指标。自适应策略也是 Parallel Scavenge  与 ParNew 的重要区别！

### 3.2老年代收集器

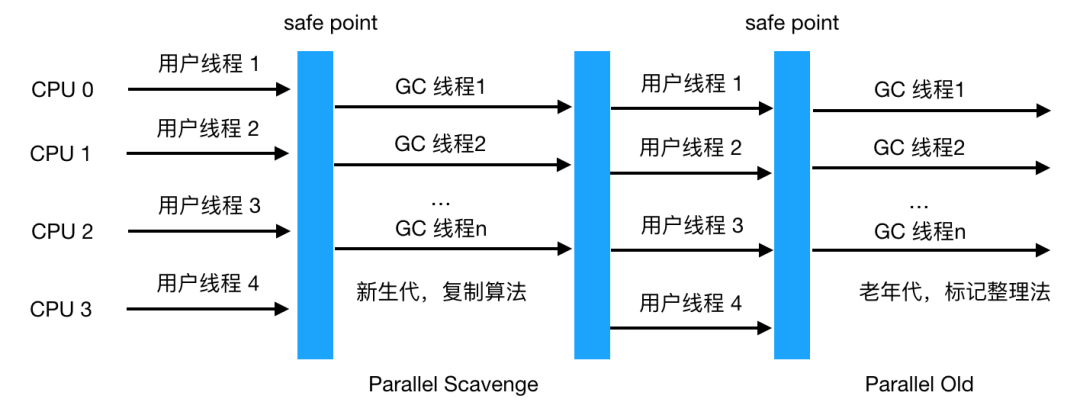
### （1）Serial Old 收集器

上文我们知道， Serial 收集器是工作于新生代的单线程收集器，与之相对地，Serial Old 是工作于老年代的单线程收集器，此收集器的主要意义在于给 Client 模式下的虚拟机使用，如果在 Server 模式下，则它还有两大用途：一种是在 JDK 1.5 及之前的版本中与 Parallel Scavenge 配合使用，另一种是作为 CMS 收集器的后备预案,在并发收集发生 Concurrent Mode Failure 时使用（后文讲述）,它与 Serial 收集器配合使用示意图如下



### （2）Parallel Old 收集器

Parallel Old 是相对于 Parallel Scavenge 收集器的老年代版本，使用多线程和标记整理法，两者组合示意图如下,这两者的组合由于都是多线程收集器，真正实现了「吞吐量优先」的目标

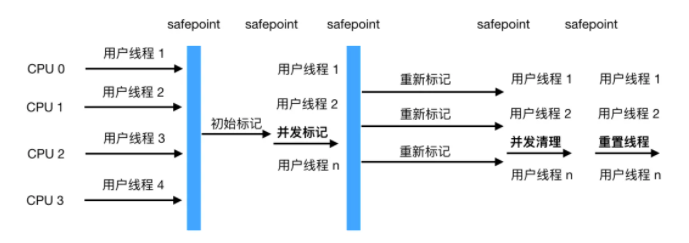


### （3）CMS 收集器

CMS 收集器是以实现最短 STW 时间为目标的收集器，如果应用很重视服务的响应速度，希望给用户最好的体验，则 CMS 收集器是个很不错的选择！

我们之前说老年代主要用标记整理法，而 CMS 虽然工作于老年代，但采用的是标记清除法，主要有以下四个步骤

1. 初始标记
2. 并发标记
3. 重新标记
4. 并发清除



从图中可以的看到初始标记和重新标记两个阶段会发生 STW，造成用户线程挂起，不过初始标记仅标记 GC Roots 能关联的对象，速度很快，并发标记是进行 GC Roots  Tracing 的过程，重新标记是为了修正并发标记期间因用户线程继续运行而导致标记产生变动的那一部分对象的标记记录，这一阶段停顿时间一般比初始标记阶段稍长，但**远比并发标记时间短**。

整个过程中耗时最长的是并发标记和标记清理，不过这两个阶段用户线程都可工作，所以不影响应用的正常使用，所以总体上看，可以认为 CMS 收集器的内存回收过程是与用户线程一起并发执行的。

但是 CMS 收集器远达不到完美的程度，主要有以下三个缺点

* CMS 收集器对 CPU 资源非常敏感  原因也可以理解，比如本来我本来可以有 10 个用户线程处理请求，现在却要分出 3 个作为回收线程，吞吐量下降了30%，CMS 默认启动的回收线程数是 （CPU数量+3）/ 4, 如果 CPU 数量只有一两个，那吞吐量就直接下降 50%,显然是不可接受的
* CMS 无法处理浮动垃圾（Floating Garbage）,可能出现 「Concurrent Mode Failure」而导致另一次 Full GC 的产生，由于在并发清理阶段用户线程还在运行，所以清理的同时新的垃圾也在不断出现，这部分垃圾只能在下一次 GC 时再清理掉（即浮云垃圾），同时在垃圾收集阶段用户线程也要继续运行，就需要预留足够多的空间要确保用户线程正常执行，这就意味着 CMS 收集器不能像其他收集器一样等老年代满了再使用，JDK 1.5 默认当老年代使用了68%空间后就会被激活，当然这个比例可以通过 -XX:CMSInitiatingOccupancyFraction 来设置，但是如果设置地太高很容易导致在 CMS 运行期间预留的内存无法满足程序要求，会导致 **Concurrent Mode Failure** 失败，这时会启用 Serial Old 收集器来重新进行老年代的收集，而我们知道 Serial Old 收集器是单线程收集器，这样就会导致 STW 更长了。
* CMS 采用的是标记清除法，上文我们已经提到这种方法会产生大量的内存碎片，这样会给大内存分配带来很大的麻烦，如果无法找到足够大的连续空间来分配对象，将会触发 Full GC，这会影响应用的性能。当然我们可以开启 -XX:+UseCMSCompactAtFullCollection（默认是开启的），用于在 CMS 收集器顶不住要进行 FullGC 时开启内存碎片的合并整理过程，内存整理会导致 STW，停顿时间会变长，还可以用另一个参数 -XX:CMSFullGCsBeforeCompation 用来设置执行多少次不压缩的 Full GC 后跟着带来一次带压缩的。

### （4）G1（Garbage First） 收集器

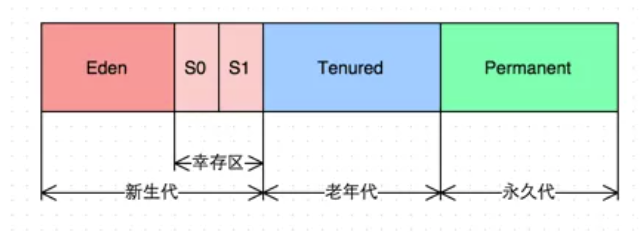
G1 收集器是面向服务端的垃圾收集器，被称为驾驭一切的垃圾回收器，主要有以下几个特点

* 像 CMS 收集器一样，能与应用程序线程并发执行。
* 整理空闲空间更快。
* 需要 GC 停顿时间更好预测。
* 不会像 CMS 那样牺牲大量的吞吐性能。
* 不需要更大的 Java Heap

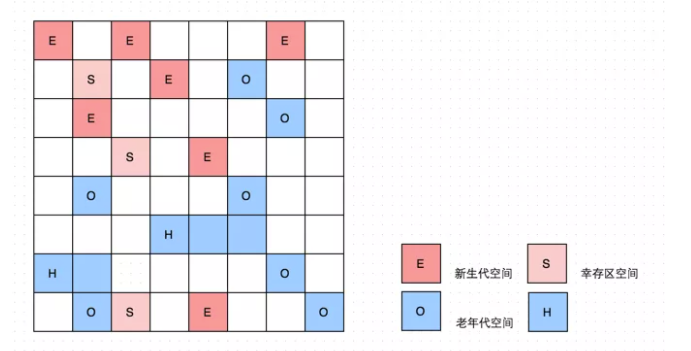
与 CMS 相比，它在以下两个方面表现更出色

1. 运作期间不会产生内存碎片，G1 从整体上看采用的是标记-整理法，局部（两个 Region）上看是基于复制算法实现的，两个算法都不会产生内存碎片，收集后提供规整的可用内存，这样有利于程序的长时间运行。
2. 在 STW 上建立了**可预测**的停顿时间模型，用户可以指定期望停顿时间，G1 会将停顿时间控制在用户设定的停顿时间以内。

为什么G1能建立可预测的停顿模型呢，主要原因在于 G1 对堆空间的分配与传统的垃圾收集器不一器，传统的内存分配就像我们前文所述，是连续的，分成新生代，老年代，新生代又分 Eden,S0,S1,如下



而 G1 各代的存储地址不是连续的，每一代都使用了 n 个不连续的大小相同的 Region，每个Region占有一块连续的虚拟内存地址，如图示

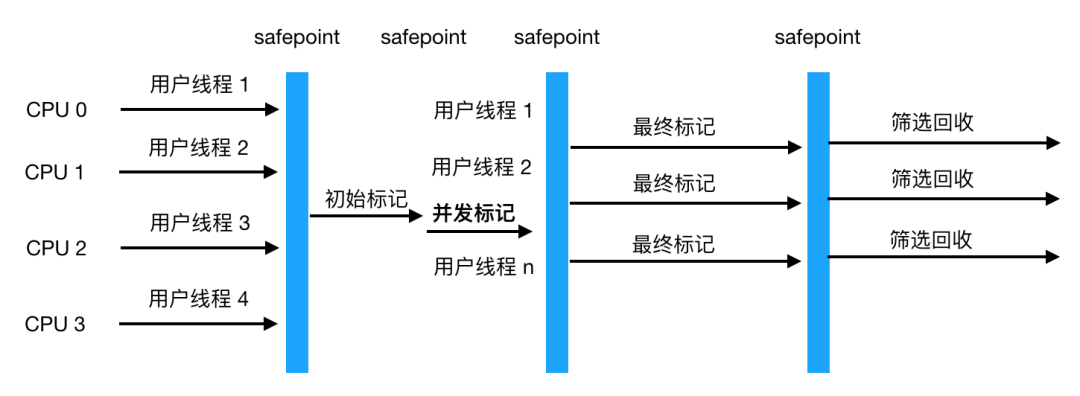


除了和传统的新老生代，幸存区的空间区别，Region还多了一个H，它代表Humongous，这表示这些Region存储的是巨大对象（humongous object，H-obj），即大小大于等于region一半的对象，这样超大对象就直接分配到了老年代，防止了反复拷贝移动。那么 G1 分配成这样有啥好处呢？

传统的收集器如果发生 Full GC 是对整个堆进行全区域的垃圾收集，而分配成各个 Region 的话，方便 G1 跟踪各个 Region 里垃圾堆积的价值大小（回收所获得的空间大小及回收所需经验值），这样根据价值大小维护一个优先列表，根据允许的收集时间，优先收集回收价值最大的 Region,也就避免了整个老年代的回收，也就减少了 STW 造成的停顿时间。同时由于只收集部分 Region,可就做到了 STW 时间的可控。

G1 收集器的工作步骤如下

1. 初始标记
2. 并发标记
3. 最终标记
4. 筛选回收



可以看到整体过程与 CMS 收集器非常类似，筛选阶段会根据各个 Region 的回收价值和成本进行排序，根据用户期望的 GC 停顿时间来制定回收计划。

### 3.4垃圾回收总结

本文简述了垃圾回收的原理与垃圾收集器的种类，相信大家有了更深刻的认识，在生产环境中我们要根据**不同的场景**来选择垃圾收集器组合，如果是运行在桌面环境处于 Client 模式的，则用 Serial + Serial Old 收集器绰绰有余，如果需要响应时间快，用户体验好的，则用 ParNew + CMS 的搭配模式，即使是号称是「驾驭一切」的 G1，也需要根据吞吐量等要求适当调整相应的 JVM 参数，没有最牛的技术，只有最合适的使用场景，切记！